Stateful Smart Contracts On Cardano

Part 1. Securing The Distributed State.

Lorenzo Fanton

887857@stud.unive.it

Problem

At the lowest level, we need to:

1. Distribute a smart contract's state over *mutiple* UTxOs, *and*...
2. ... Ensure that it cannot be compromised by an attacker.

Otherwise, it would be pointless to use whatever abstractions we might come up with.

Problem

An informal statement of the problem might be the following.

> A smart contract's state, distributed over multiple of UTxOs, should be modified only by transactions validated by the smart contract itself.

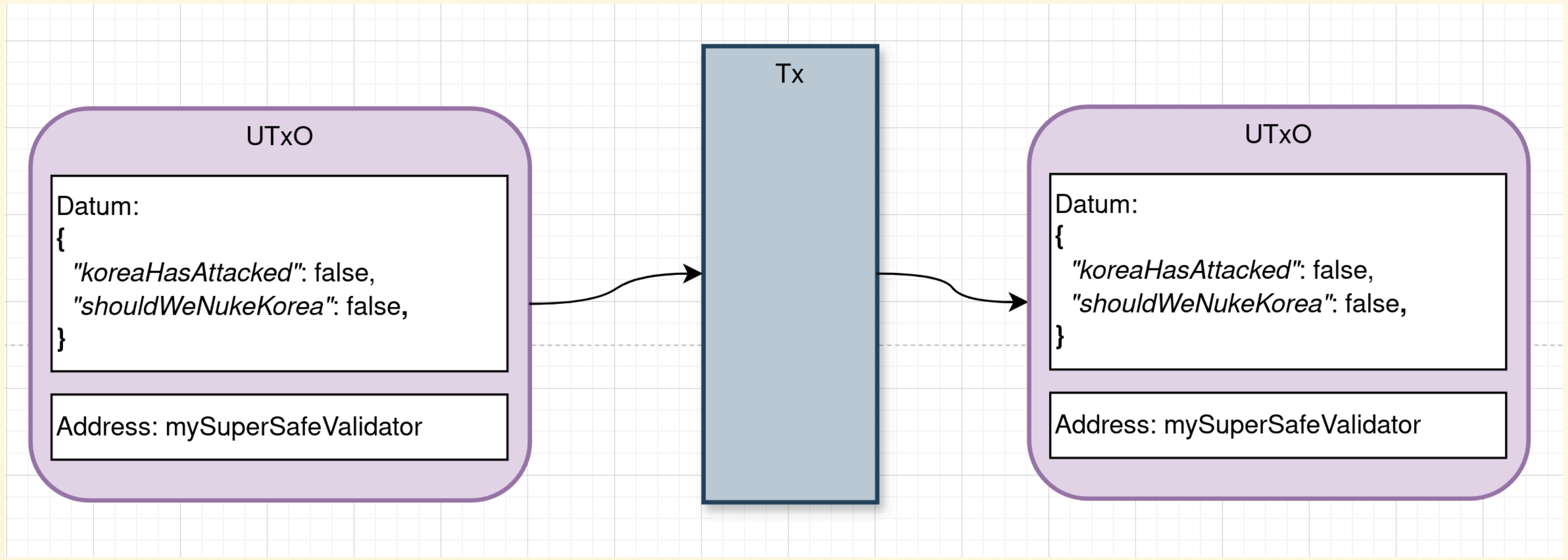
Problem

There are good news and bad news.

- **Good news**: Smart contracts (on-chain) can enforce the recipient(s) of the UTxOs being created, but...
- **Bad news**: ... They have **zero knowledge** about the sender of the UTxOs being spent.

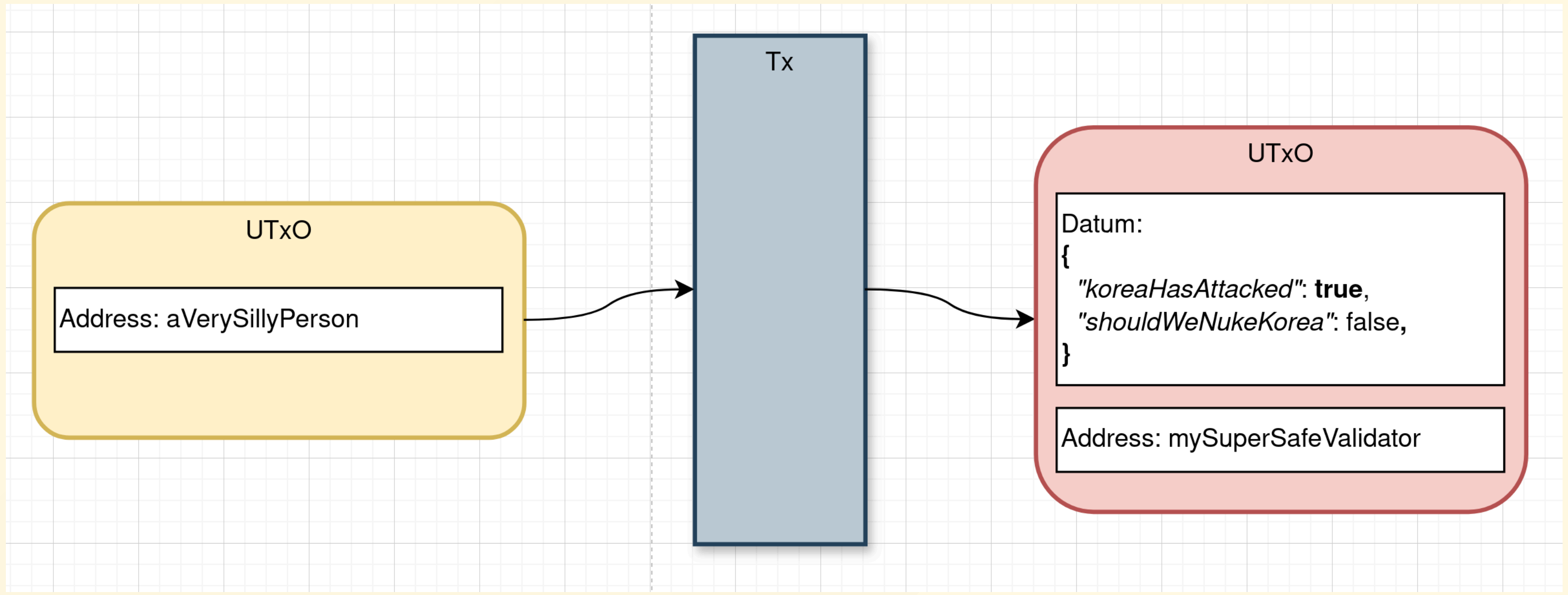
Why is that a problem?

Example: Good State


UTxO
Datum:
{
"koreaHasAttacked": false,
"shouldWeNukeKorea": false,
}
Address: mySuperSafeValidator
Tx
UTxO
Datum:
{
"koreaHasAttacked": false,
"shouldWeNukeKorea": false,
}
Address: mySuperSafeValidator

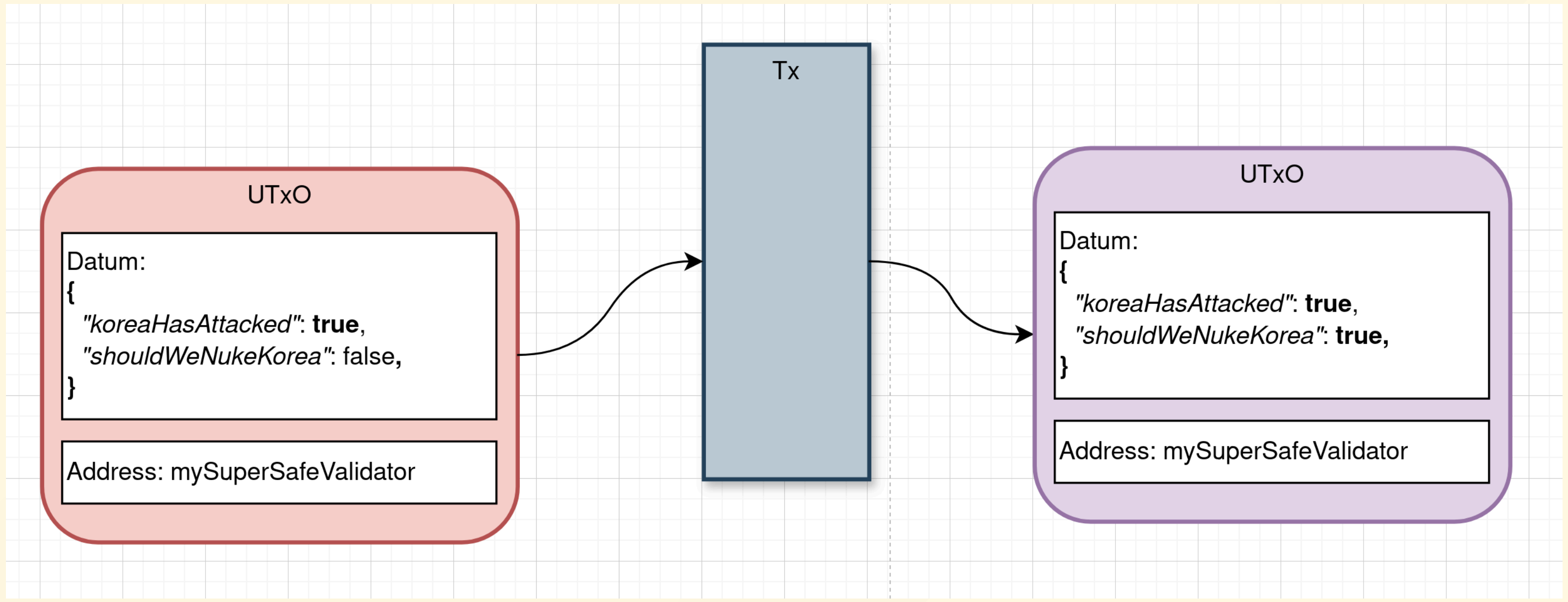
In a normal setup, a stateful smart contract would ensure that the state is propagated to itself.

Example: Attacker Setup


UTxO
Address: aVerySillyPerson
Tx
UTxO
Datum:
{
"koreaHasAttacked": true,
"shouldWeNukeKorea": false,
}
Address: mySuperSafeValidator

However, an attacker might craft a malicious piece of state and "pay" it to the smart contract. Nothing would prevent them to do so.

Example: (Very) Bad State



Datum:
{
  "koreaHasAttacked": **true**,
  "shouldWeNukeKorea": false,
}

Datum:
{
  "koreaHasAttacked": **true**,
  "shouldWeNukeKorea": **true,**
}

Now the smart contract would mindlessly use that piece of state to validate whatever transaction. **The contract's state is now compromised.**

Problem

Compromised state could be the source of other, more serious issues.

What could be worse than WW3? Well, for example:

> “ *You* could be losing money!! ”

While total thermonuclear annihilation might be somewhat tolerable and even fun, losing money is not and we must avoid that at all costs.

Solution

How can we authenticate the UTxOs that get consumed/produced by the transactions validated by our smart contract?

For example, by using a set of unique "tokens" attached to each UTxO, such that:

- They are easily recognizable by the smart contract
- They can be extended without modifying the contract's source code

Enter NFTs.

Native Tokens And NFTs

Native tokens, also called *assets* on Cardano, are (loosely) objects that represent value. Technically, ADA is a (very special) type of native token.

Native tokens are idenfied by their "asset id", of type $\mathrm{AssetId} = \mathrm{PolicyId} \times \mathrm{AssetName}$:

- $\mathrm{PolicyId} = (\mathbb{F}_{256})^{28}$ hash of the minting script;
- $\mathrm{AssetName} = (\mathbb{F}_{256})^{32}$ arbitrary 32-byte string.

Where $\mathbb{F}_q := \{0, \ldots, q-1\}$.

Fungible, Not Fungible

Two tokens are said to be *fungible* if they have the same asset id.

Example:

- If I have 2 of `("0x1", "lorenzo")`, then I have 2 tokens that **are** fungible.
- If I have 1 of `("0x1", "hammad")` and 1 of `("0x1", "lorenzo")`, the two tokens **are not** fungible.
- If I have 1 of `("0x1", "hammad")` and 1 of `("0x2", "hammad")`, the two tokens **are not** fungible.

Minting Native Tokens

Minting is the process of creating new tokens. A minting validator is just a smart contract with the following signature M:

$$M = (\mathrm{Redeemer} \times \mathrm{ScriptContext}) \to \mathbb{B}$$

Minting validators do not take a Datum as argument, because they do not spend UTxOs.

Spending UTxOs

By contrast, spending validators are smart contracts with the following signature V:

$$V = (\mathrm{Datum} \times \mathrm{Redeemer} \times \mathrm{ScriptContext}) \to \mathbb{B}$$

Properties Of Tokens

We said earlier that we want to utilize unique tokens, to guarantee that each piece of the smart contract's state is indeed valid. Firstly, we have to discuss two desired properties of such tokens.

In particular, let T the minting process detailed in the following slides.

1. $\forall t_1, t_2$ tokens from $T : t_1 \neq t_2$ using the asset id to test for equality.
2. $\forall t_1, t_2$ tokens from $T : \exists \sim$ such that $t_1 \sim t_2$. We will show that one suitable $\sim$ is precisely "t_1 and t_2 have the same policy id".

Properties Of Tokens

In particular, if we manage to mint NFTs we get (1) basically for free.

The (2) is tougher. Usually on Cardano NFTs are produced from a single script either setting timers for the validity of the contract, or parametrizing the contract with a UTxO reference and making sure to spend it in the transaction (in order to achieve a one-time mint validator).

Properties Of Tokens

However, this means that in order to produce new NFTs you would need a different contract (using different UTxO references changes the actual compiled code). In the context of native tokens, different minting validators yield different policy ids.

What we are going to achieve is a "minting process" to have multiple NFTs from the same policy id.

Multiple NFTs From The Same PolicyId

Let $\mathrm{OutputReference}$ represent an on-chain reference to an UTxO.

We declare the following functions:

- $f_{start,mint} : \mathrm{OutputReference} \to M$
- $f_{add,mint} : M \to M$
- $f_{add,validate} : (M \times M) \to V$

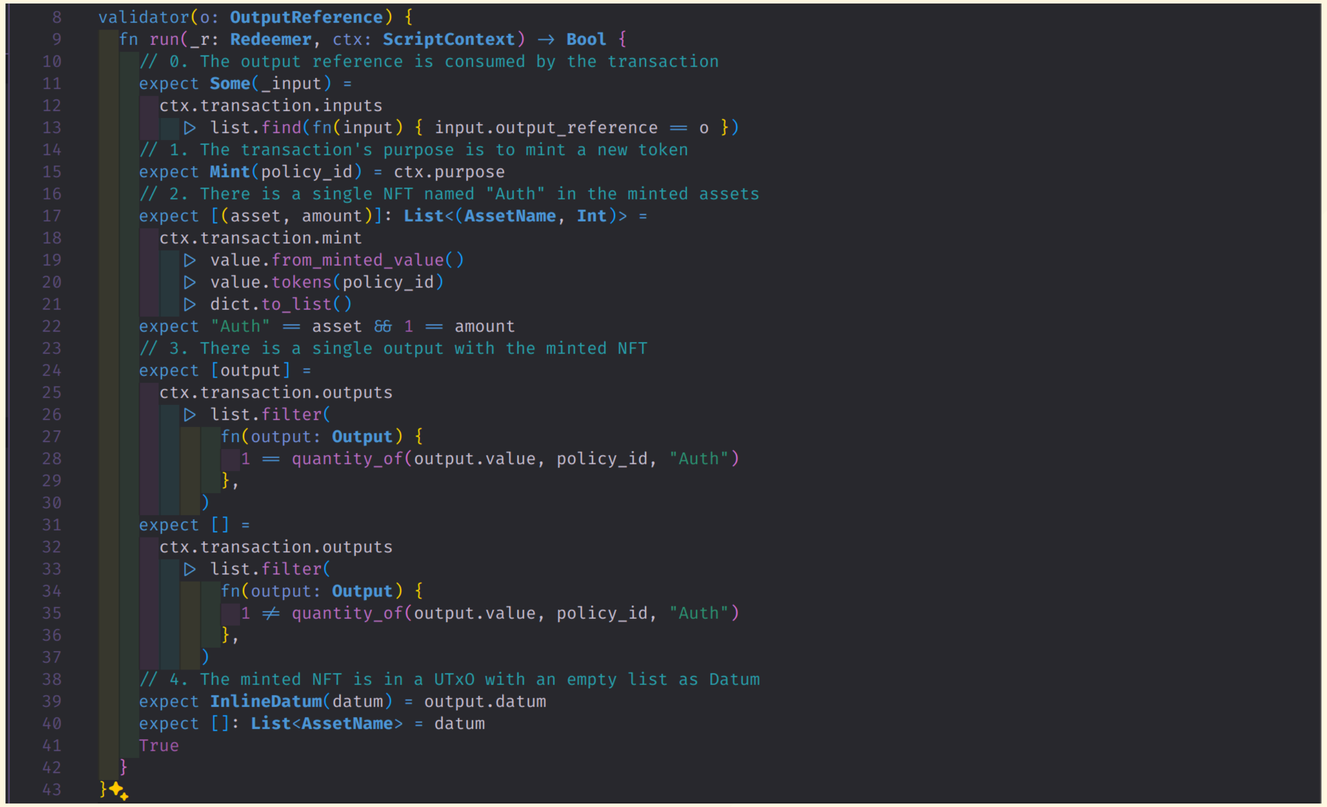
Where M is "minting validator" and V is "spending validator". **NB**: With abuse of notation, we use the actual validator instead of its hash (when used as argument). Obviously, in real code the hash is used.

Multiple NFTs From The Same PolicyId

Some key properties:

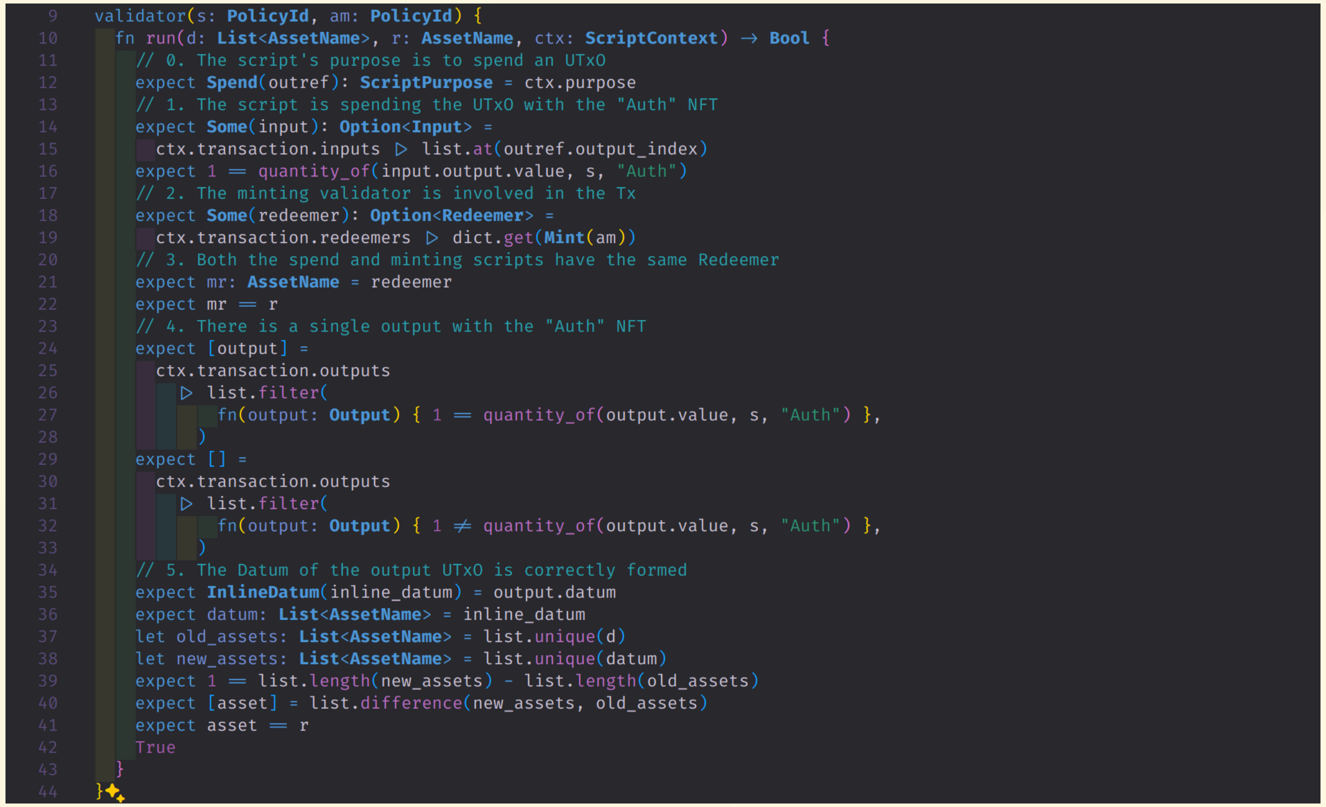
- $o_1 \neq o_2 \iff f_{start,mint}(o_1) \neq f_{start,mint}(o_2)$
- $s_1 \neq s_2 \iff f_{add,mint}(s_1) \neq f_{add,mint}(s_2)$
- $(s_1, a_1) \neq (s_1, a_2) \iff f_{add,validate}(s_1, a_1) \neq f_{add,validate}(s_2, a_2)$

Code: $f_{start,mint} : \mathrm{OutputReference} \to M$



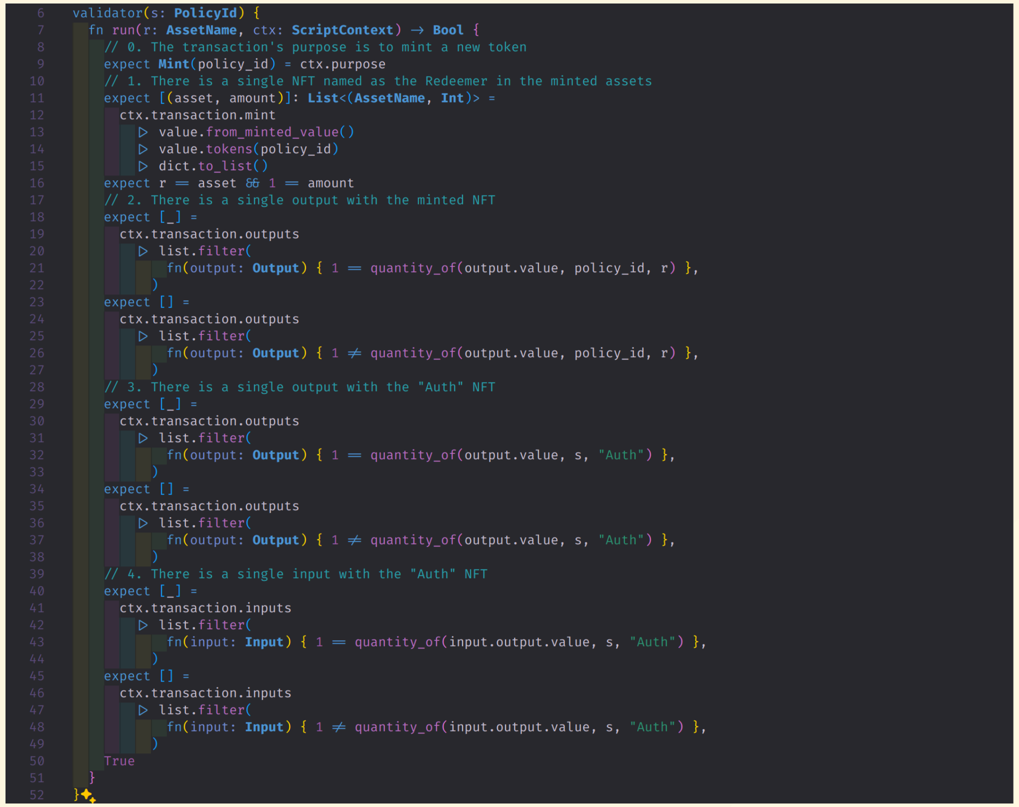
```
validator(o: OutputReference) {
  fn run(_r: Redeemer, ctx: ScriptContext) → Bool {
    // 0. The output reference is consumed by the transaction
    expect Some(_input) =
      ctx.transaction.inputs
        ▷ list.find(fn(input) { input.output_reference == o })
    // 1. The transaction's purpose is to mint a new token
    expect Mint(policy_id) = ctx.purpose
    // 2. There is a single NFT named "Auth" in the minted assets
    expect [(asset, amount)]: List<(AssetName, Int)> =
      ctx.transaction.mint
        ▷ value.from_minted_value()
        ▷ value.tokens(policy_id)
        ▷ dict.to_list()
    expect "Auth" == asset && 1 == amount
    // 3. There is a single output with the minted NFT
    expect [output] =
      ctx.transaction.outputs
        ▷ list.filter(
            fn(output: Output) {
              1 == quantity_of(output.value, policy_id, "Auth")
            },
          )
    expect [] =
      ctx.transaction.outputs
        ▷ list.filter(
            fn(output: Output) {
              1 ≠ quantity_of(output.value, policy_id, "Auth")
            },
          )
    // 4. The minted NFT is in a UTxO with an empty list as Datum
    expect InlineDatum(datum) = output.datum
    expect []: List<AssetName> = datum
    True
  }
}
```

Code: $f_{add,validate} : (M \times M) \to V$



```
validator(s: PolicyId, am: PolicyId) {
  fn run(d: List<AssetName>, r: AssetName, ctx: ScriptContext) -> Bool {
    // 0. The script's purpose is to spend an UTxO
    expect Spend(outref): ScriptPurpose = ctx.purpose
    // 1. The script is spending the UTxO with the "Auth" NFT
    expect Some(input): Option<Input> =
      ctx.transaction.inputs |> list.at(outref.output_index)
    expect 1 == quantity_of(input.output.value, s, "Auth")
    // 2. The minting validator is involved in the Tx
    expect Some(redeemer): Option<Redeemer> =
      ctx.transaction.redeemers |> dict.get(Mint(am))
    // 3. Both the spend and minting scripts have the same Redeemer
    expect mr: AssetName = redeemer
    expect mr == r
    // 4. There is a single output with the "Auth" NFT
    expect [output] =
      ctx.transaction.outputs
        |> list.filter(
            fn(output: Output) { 1 == quantity_of(output.value, s, "Auth") },
          )
    expect [] =
      ctx.transaction.outputs
        |> list.filter(
            fn(output: Output) { 1 != quantity_of(output.value, s, "Auth") },
          )
    // 5. The Datum of the output UTxO is correctly formed
    expect InlineDatum(inline_datum) = output.datum
    expect datum: List<AssetName> = inline_datum
    let old_assets: List<AssetName> = list.unique(d)
    let new_assets: List<AssetName> = list.unique(datum)
    expect 1 == list.length(new_assets) - list.length(old_assets)
    expect [asset] = list.difference(new_assets, old_assets)
    expect asset == r
    True
  }
}
```

Code: $f_{add,mint} : M \to M$



```
validator(s: PolicyId) {
  fn run(r: AssetName, ctx: ScriptContext) -> Bool {
    // 0. The transaction's purpose is to mint a new token
    expect Mint(policy_id) = ctx.purpose
    // 1. There is a single NFT named as the Redeemer in the minted assets
    expect [(asset, amount)]: List<(AssetName, Int)> =
      ctx.transaction.mint
        |> value.from_minted_value()
        |> value.tokens(policy_id)
        |> dict.to_list()
    expect r == asset && 1 == amount
    // 2. There is a single output with the minted NFT
    expect [_] =
      ctx.transaction.outputs
        |> list.filter(
             fn(output: Output) { 1 == quantity_of(output.value, policy_id, r) },
           )
    expect [] =
      ctx.transaction.outputs
        |> list.filter(
             fn(output: Output) { 1 != quantity_of(output.value, policy_id, r) },
           )
    // 3. There is a single output with the "Auth" NFT
    expect [_] =
      ctx.transaction.outputs
        |> list.filter(
             fn(output: Output) { 1 == quantity_of(output.value, s, "Auth") },
           )
    expect [] =
      ctx.transaction.outputs
        |> list.filter(
             fn(output: Output) { 1 != quantity_of(output.value, s, "Auth") },
           )
    // 4. There is a single input with the "Auth" NFT
    expect [_] =
      ctx.transaction.inputs
        |> list.filter(
             fn(input: Input) { 1 == quantity_of(input.output.value, s, "Auth") },
           )
    expect [] =
      ctx.transaction.inputs
        |> list.filter(
             fn(input: Input) { 1 != quantity_of(input.output.value, s, "Auth") },
           )
    True
  }
}
```

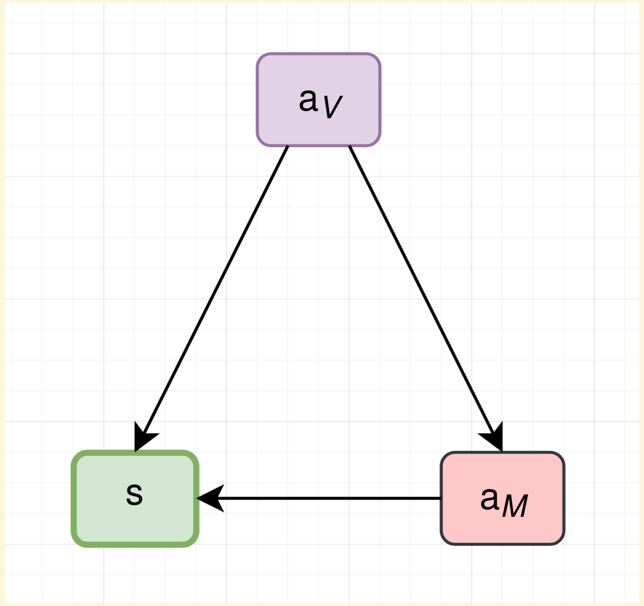
Multiple NFTs From The Same PolicyId

In particular, let $o : \mathrm{OutputReference}$.

We build the following three contracts that will take part in the generation of our NFTs:

- $(s : M) = f_{start,mint}(o)$ is the "start script".
- $(a_M : M) = f_{add,mint}(s)$ is the script for creating a new NFT.
- $(a_V : V) = f_{add,validate}(s, a_M)$ is the script for co-validating most of the minting.

Dependencies

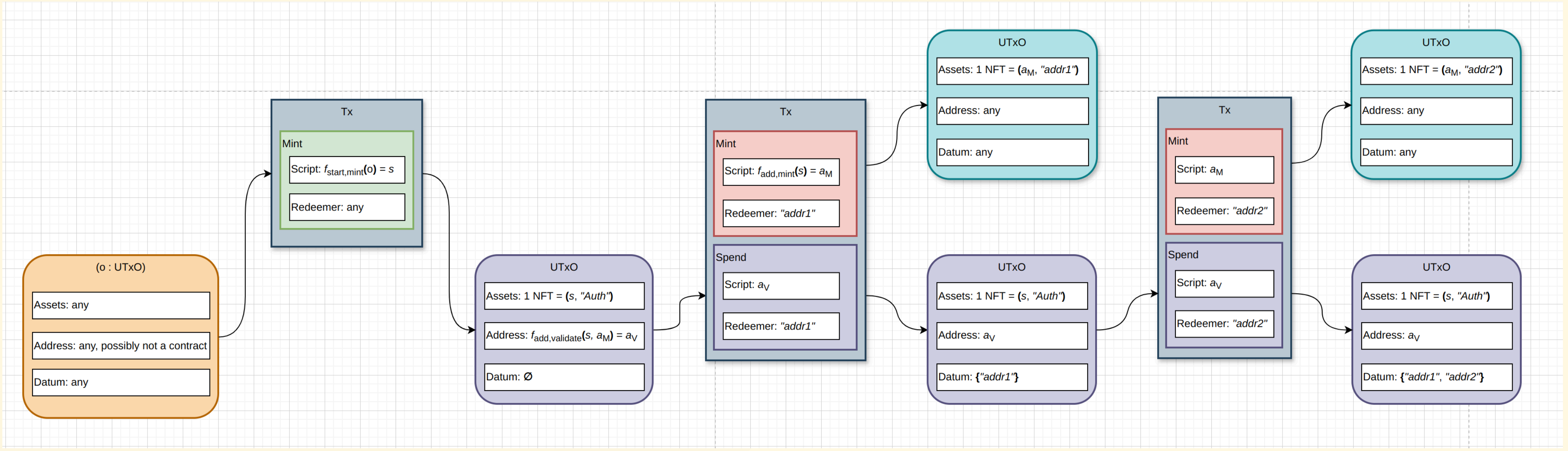

$$\begin{tikzcd} & a_V \arrow[dl] \arrow[dr] & \\ s & & a_M \arrow[ll] \end{tikzcd}$$

NB: "$\to$" means "depends on".

Multiple NFTs From The Same PolicyId

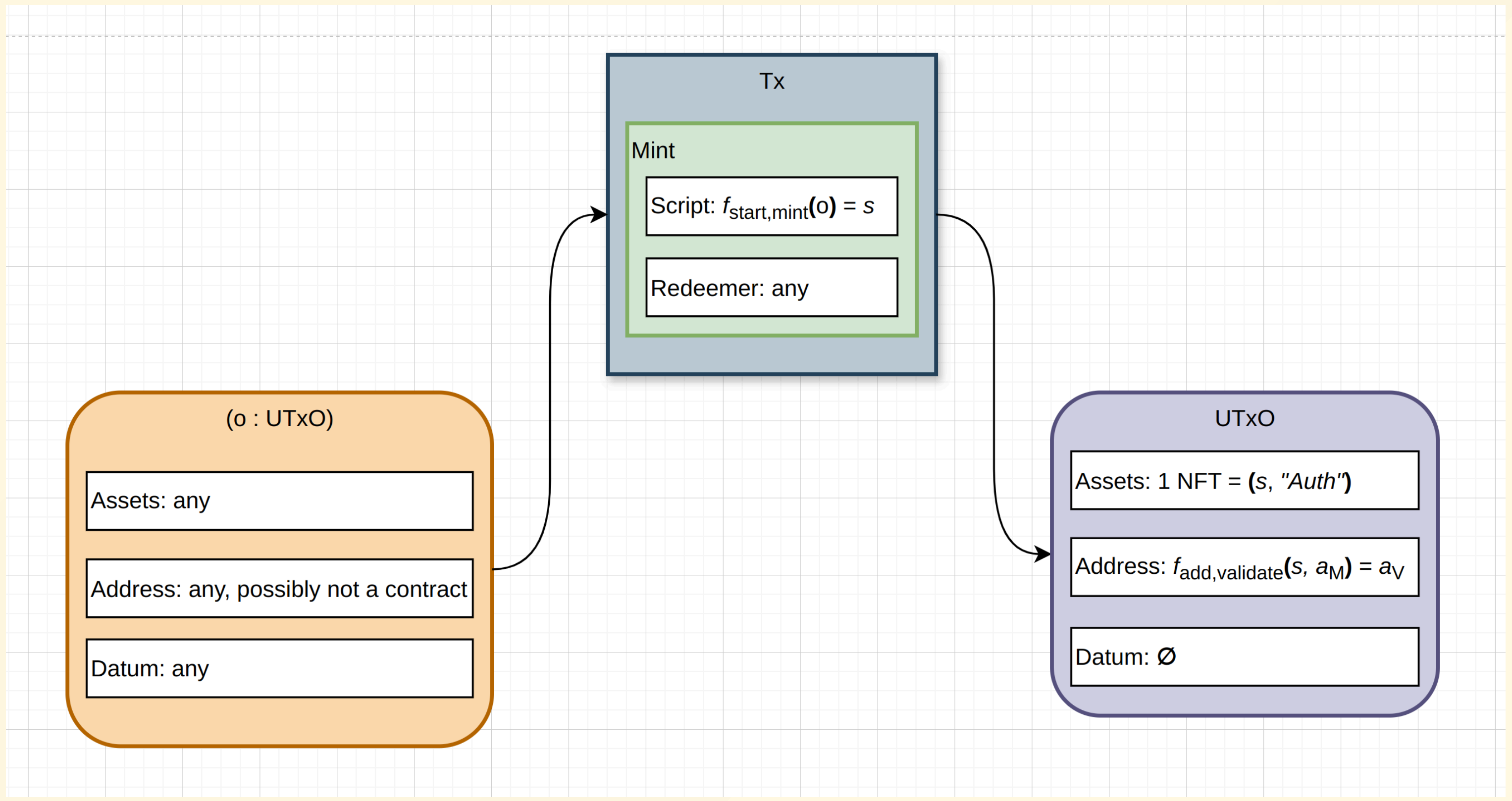
TODO: simple description of the schema.

Example: Full Example

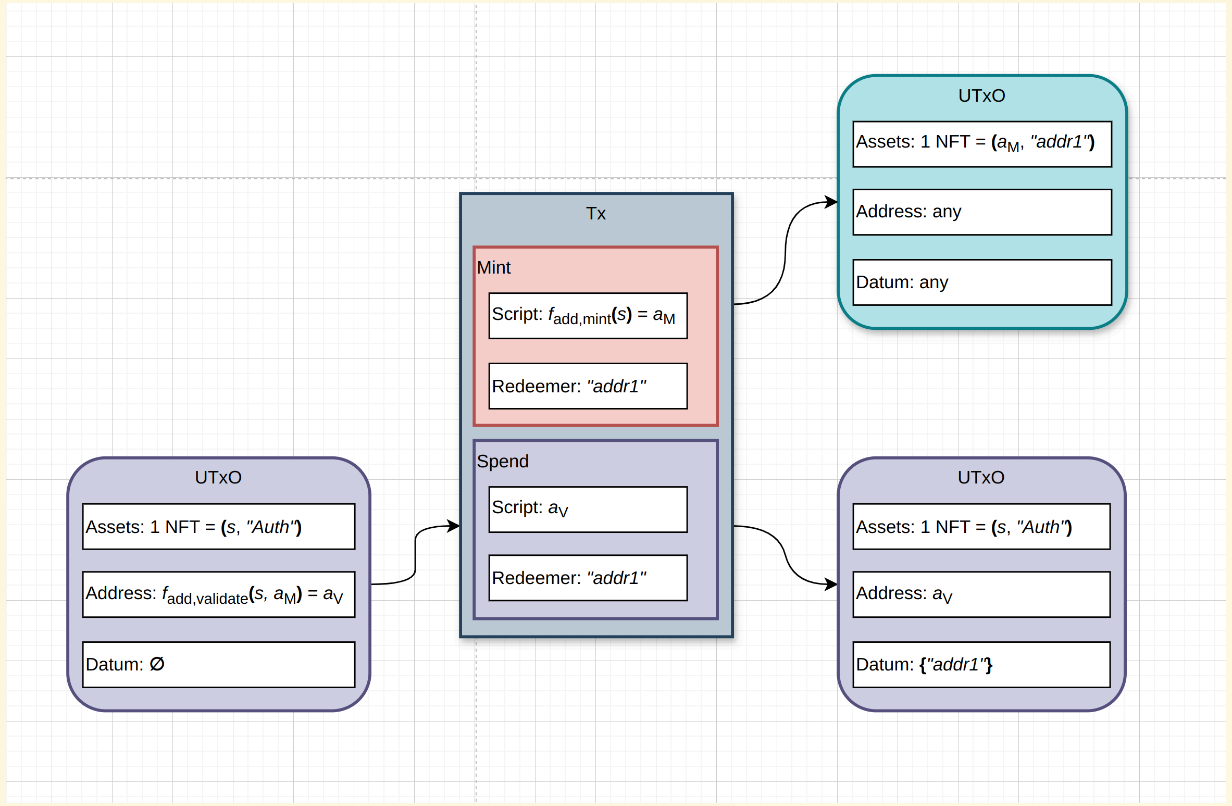

(o : UTxO)
Assets: any
Address: any, possibly not a contract
Datum: any
Tx
Mint
Script: $f_{\text{start,mint}}(o) = s$
Redeemer: any
UTxO
Assets: 1 NFT = (s, "Auth")
Address: $f_{\text{add,validate}}(s, a_M) = a_V$
Datum: ∅
Tx
Mint
Script: $f_{\text{add,mint}}(s) = a_M$
Redeemer: "addr1"
Spend
Script: a_V
Redeemer: "addr1"
UTxO
Assets: 1 NFT = (a_M, "addr1")
Address: any
Datum: any
UTxO
Assets: 1 NFT = (s, "Auth")
Address: a_V
Datum: {"addr1"}
Tx
Mint
Script: a_M
Redeemer: "addr2"
Spend
Script: a_V
Redeemer: "addr2"
UTxO
Assets: 1 NFT = (a_M, "addr2")
Address: any
Datum: any
UTxO
Assets: 1 NFT = (s, "Auth")
Address: a_V
Datum: {"addr1", "addr2"}

NB: UTxOs coloured cyan contain the minted NFTs that we will use to authenticate our smart contract's state.

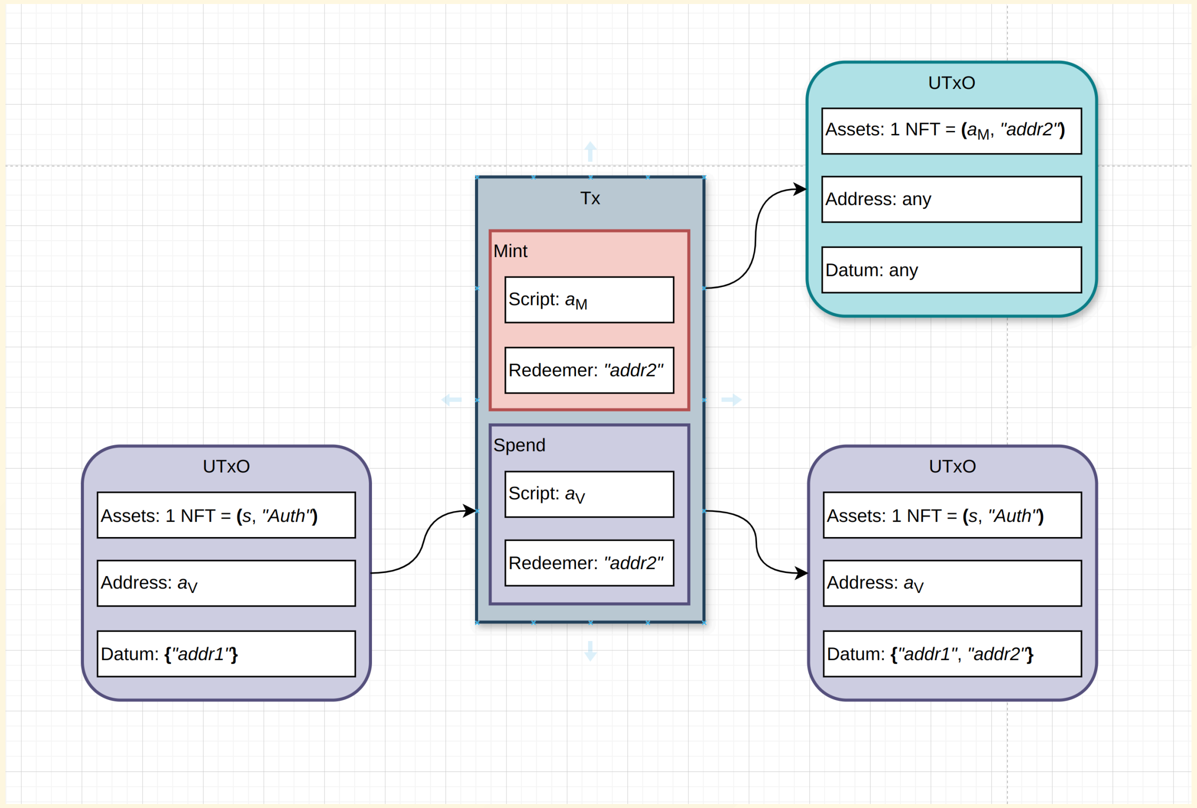
Example: Start The Minting Process


Tx
Mint
Script: $f_{start,mint}(o) = s$
Redeemer: any
(o : UTxO)
Assets: any
Address: any, possibly not a contract
Datum: any
UTxO
Assets: 1 NFT = $(s, \textit{"Auth"})$
Address: $f_{add,validate}(s, a_M) = a_V$
Datum: Ø

Example: Minting The 1st NFT


UTxO
Assets: 1 NFT = (s, "Auth")
Address: $f_{add,validate}(s, a_M) = a_V$
Datum: Ø
Tx
Mint
Script: $f_{add,mint}(s) = a_M$
Redeemer: "addr1"
Spend
Script: a_V
Redeemer: "addr1"
UTxO
Assets: 1 NFT = (a_M, "addr1")
Address: any
Datum: any
UTxO
Assets: 1 NFT = (s, "Auth")
Address: a_V
Datum: {"addr1"}

Example: Minting The 2nd NFT



Final Notes

Finally, we can use them.

A smart contract would just need to know the hash of a_M (the policy id of the NFTs). The uniqueness of them is guaranteed by the validators. Lastly, a valid "piece of state" would be one associated to the NFT. The smart contract would have to make sure to pay the NFT back to itself.

Critical Points

- The "Auth" NFT must be paid to a_V in the same transaction it is minted
- The data NFTs must be paid to the smart contract, giving the correct initial datum.