Stateful Smart Contracts On Cardano

Part 1. Securing The Distributed State

Minting Unique Tokens With A Shared `PolicyId`

Lorenzo Fanton

887857@stud.unive.it

Problem

Our needs, at the lowest level:

1. We want to distribute a smart contract's state over mutiple UTxOs, *and*...
2. ... We must 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.

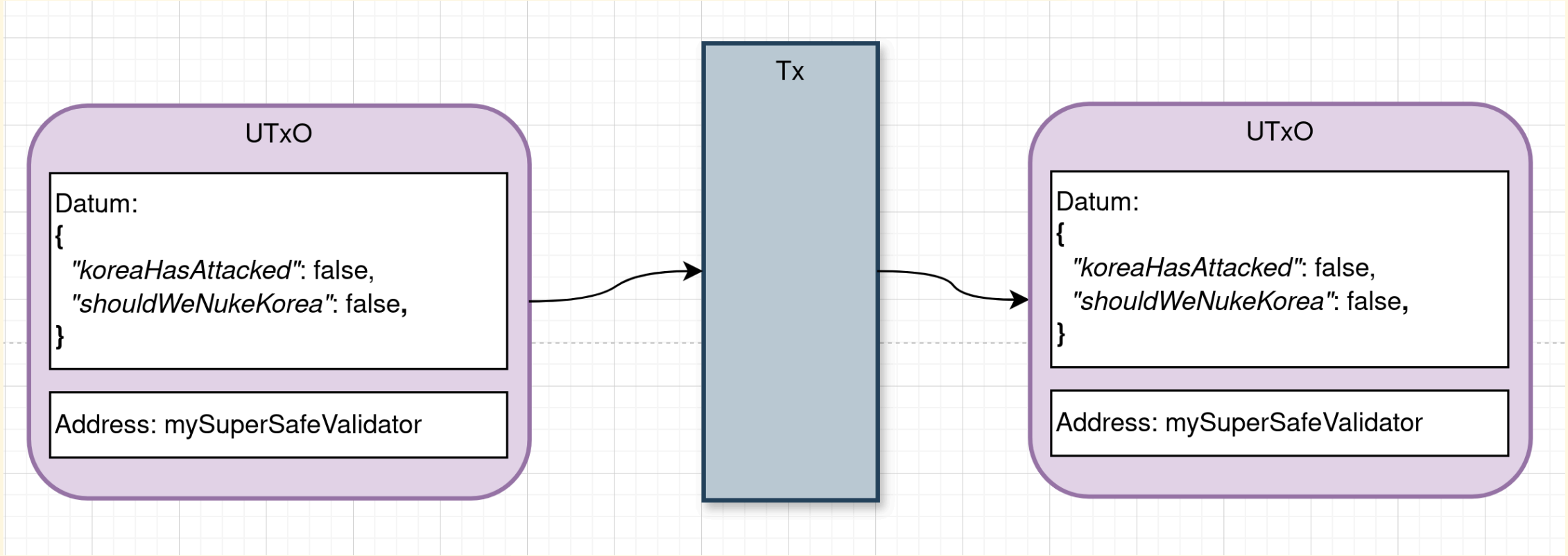
Problem

There are good news and bad news.

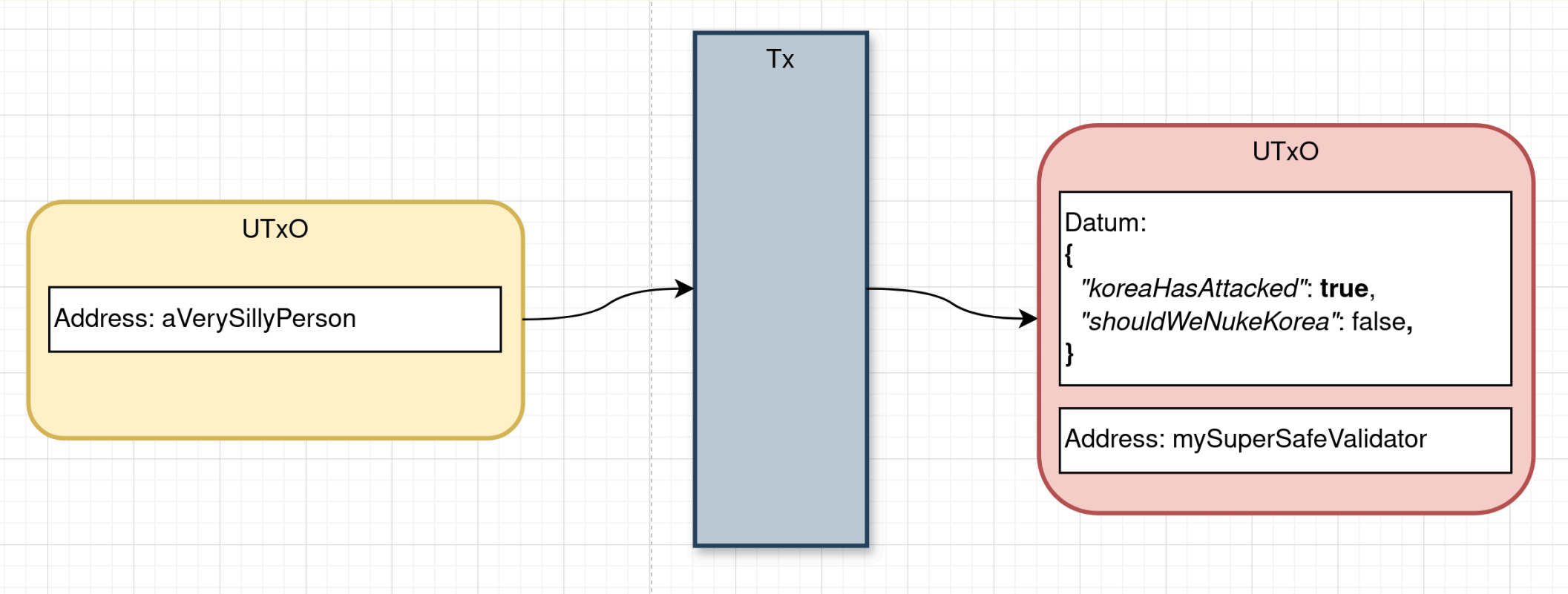
- **Good news**: Smart contracts (on-chain) can enforce the recipient(s) of the UTxOs being created, but...
- **Bad news**: ... They have **zero knowledge** about the sender of the UTxOs being spent.

Why is that a problem?

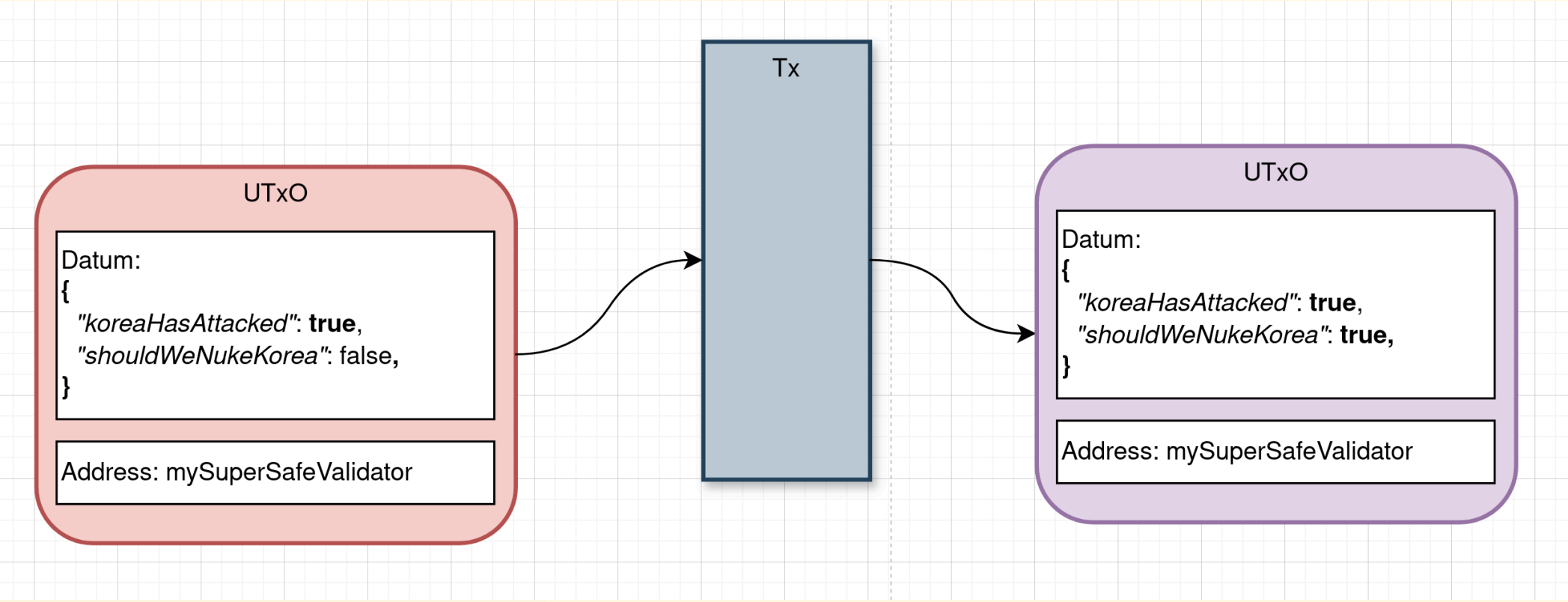
Example: Good State



Example: Attacker Setup



Example: (Very) Bad State



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

We need a way to authenticate the UTxOs that get consumed/produced by the transactions validated by our smart contract.

Idea: What if we had a set of unique "tokens", known by the contract, that could be propagated alongside each UTxO?

Congrats. We have just discovered NFTs.

Native Tokens

Native tokens, also called *assets* on Cardano, are defined as:

- $\mathrm{AssetId} := \mathrm{PolicyId} \times \mathrm{AssetName}$
- $\mathrm{PolicyId} := (\mathbb{F}_{256})^{28}$ (hash of the minting validator script)
- $\mathrm{AssetName} := (\mathbb{F}_{256})^{32}$ (arbitrary string)

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

Native Tokens

On Cardano, every asset is a Native Token. Even ADA can be considered as such (with an empty policy id, since there is no mint script for ADA).

In fact, UTxOs can carry Native Tokens, not just ADA.

Fungible, Non Fungible

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

Example: if I have an amount of `("0x1", "lore")`, then I have 2 tokens that fungible (one with each other).

If I have 1 of `("0x1", "hammad")` and 1 of `("0x1", "lore")`, the two are not fungible.

If I have 1 of `("0x1", "hammad")` and 1 of `("0x2", "hammad")`, the two are not fungible.

Spending Validators

We call V the type of a Spending Validator:

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

Minting Validators

We call M the type of a Minting Validator:

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

NB: Minting Validators do not take a Datum as inputs, because they do not spend UTxOs.

Minting The NFTs

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

We declare the following functions that produce smart contracts:

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

Minting The NFTs

$f_{start,mint}$ takes an $\mathrm{OutputReference}$ `o` and returns a Minting Validator that produces a single token with $\mathrm{AssetName}$ `"Auth"`.

Such script is a **one-time** minting validator, because it ensures that `o` is consumed in the transaction (which can happen only once).

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

```
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
  }
}
```

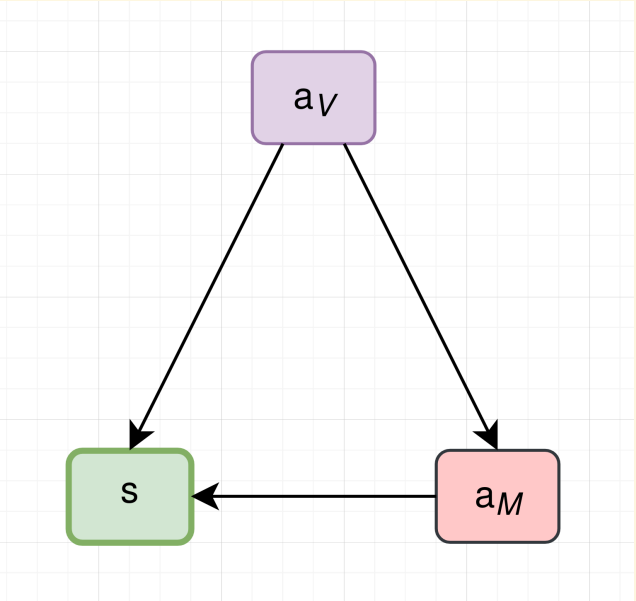
Minting The NFTs

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

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

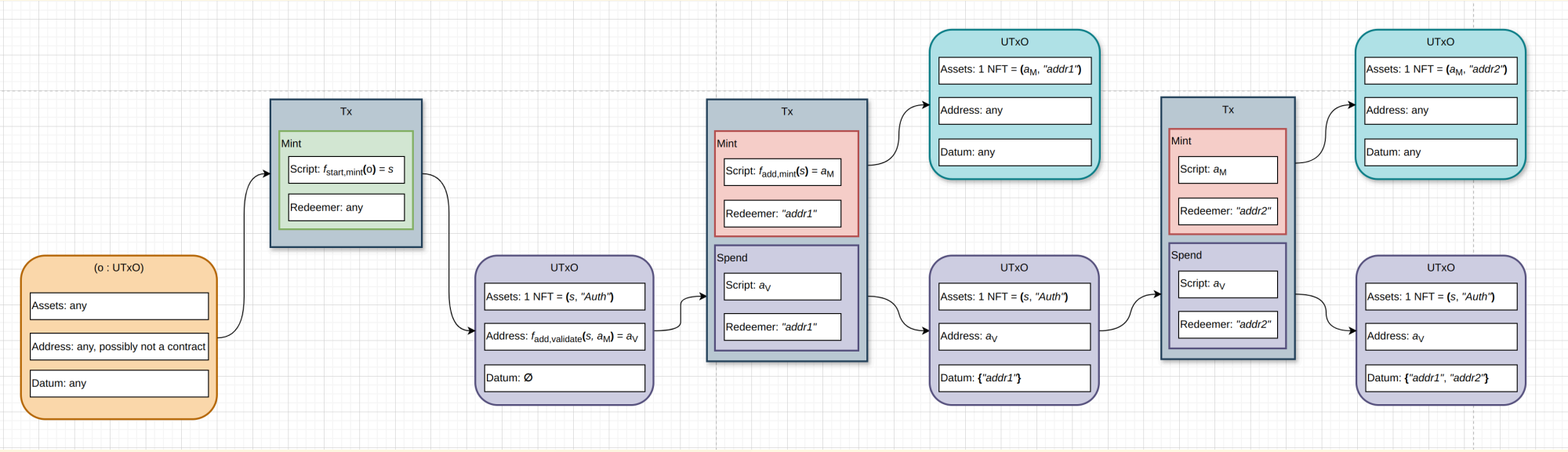
- $(s : M) = f_{start,mint}(o)$
- $(a_M : M) = f_{add,mint}(s)$
- $(a_V : V) = f_{add,validate}(s, a_M)$

Dependencies



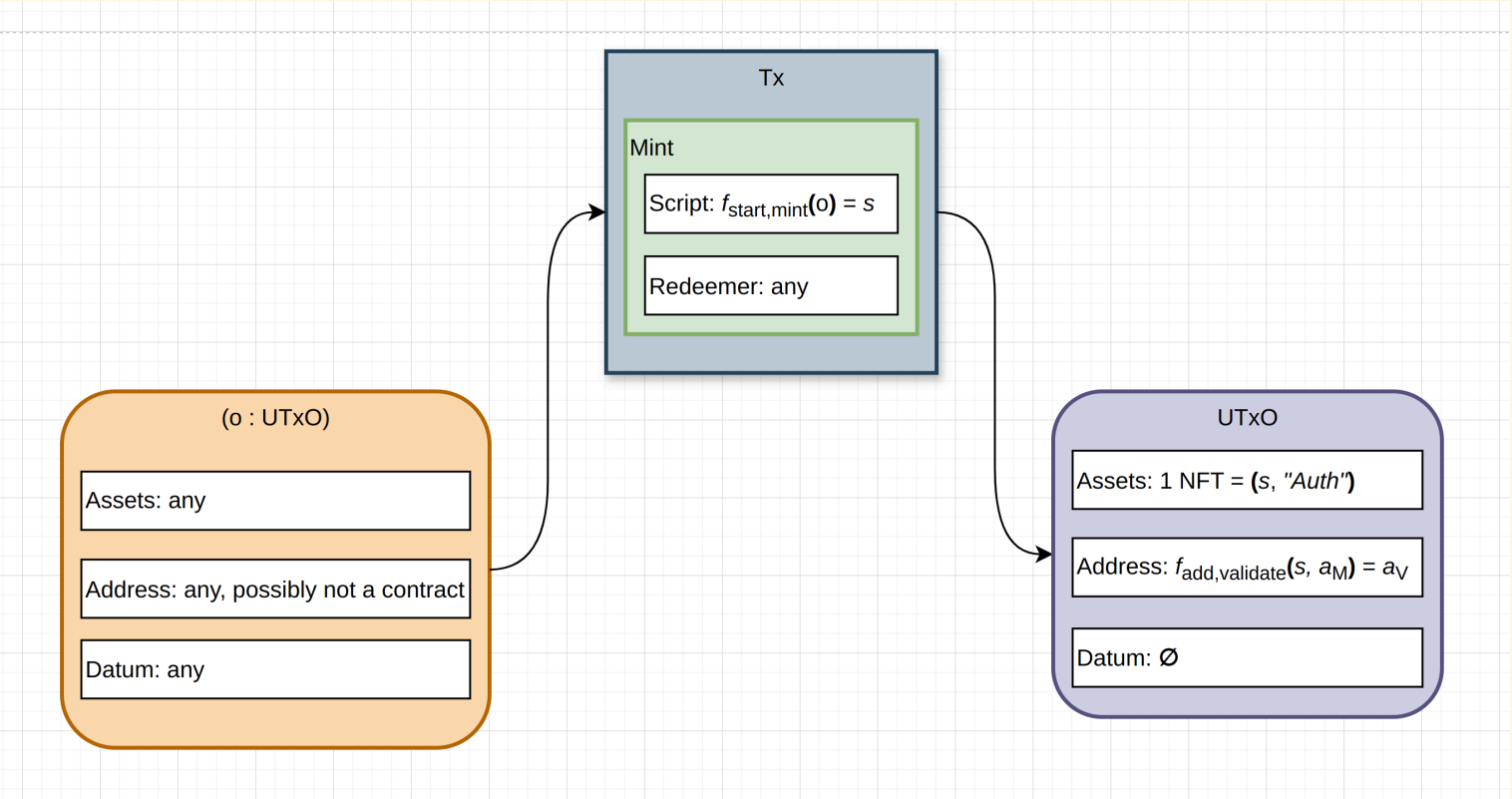
NB: "$\rightarrow$" means "depends on".

Example: Full Example

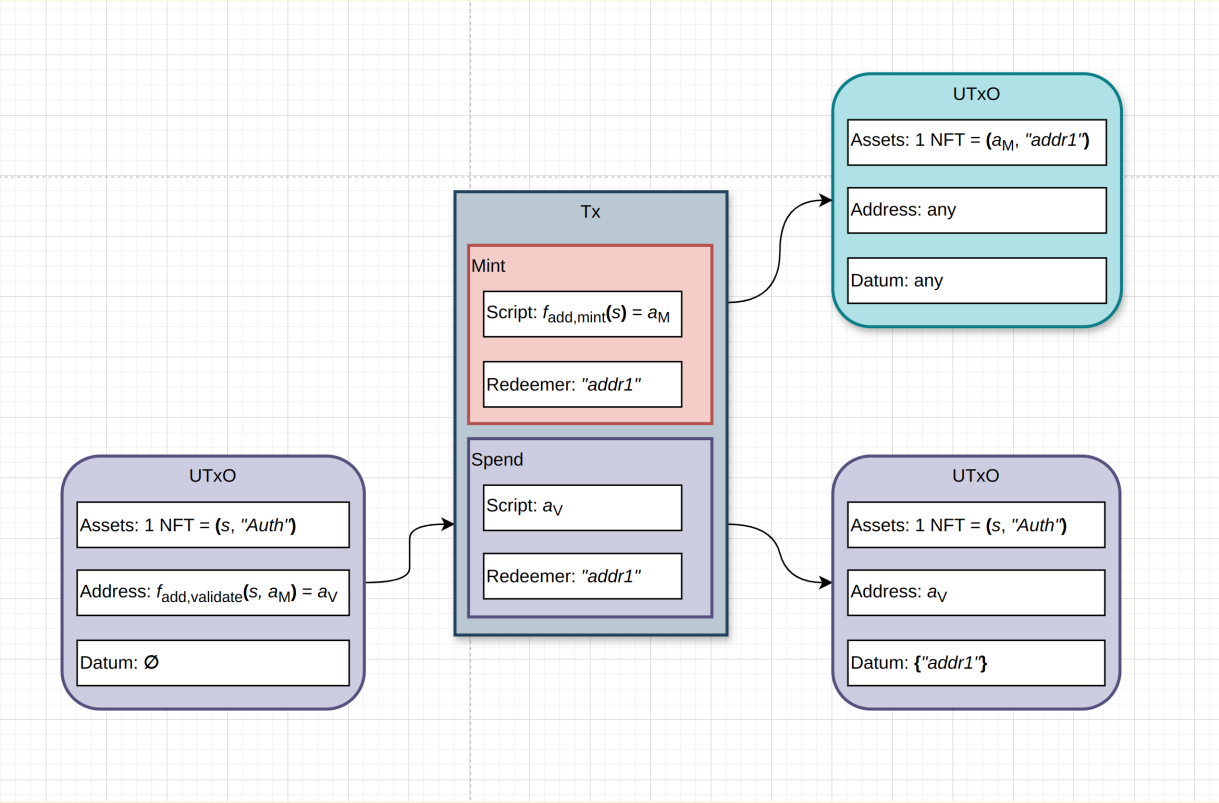


NB Cyan UTxOs contain the minted NFTs that we will use to authenticate our smart contract's state.

Example: Start The Minting Process



Example: Minting The 1st NFT



Example: Minting The 2nd NFT