Stateful Smart Contracts On Cardano

Part 1. Securing The Distributed State With NFTs

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.

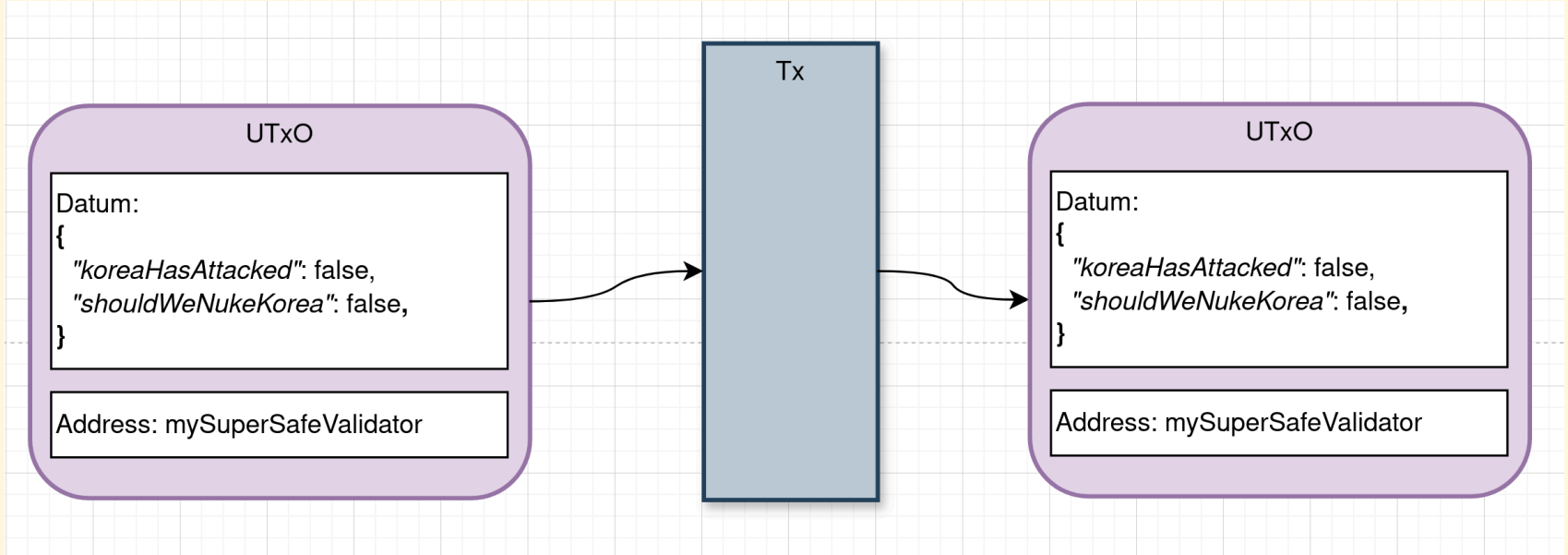
Problem

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.

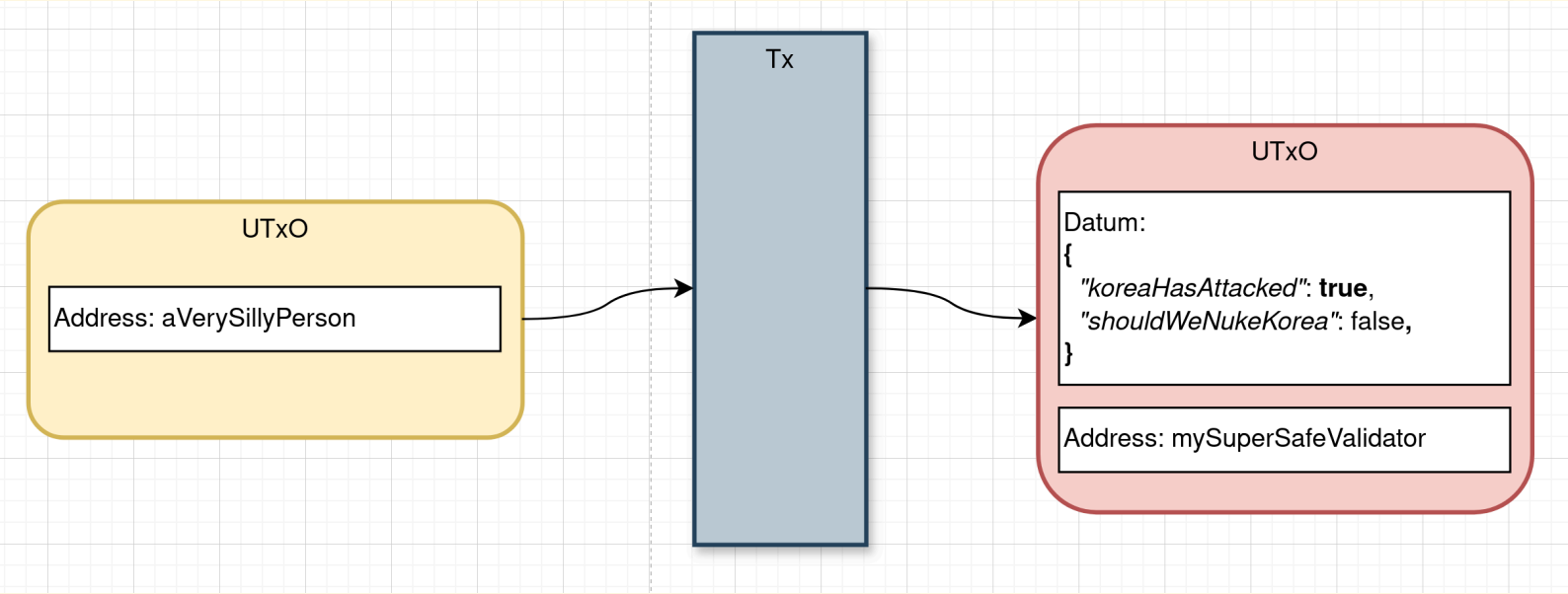
Why is that a problem?

Example: Good State



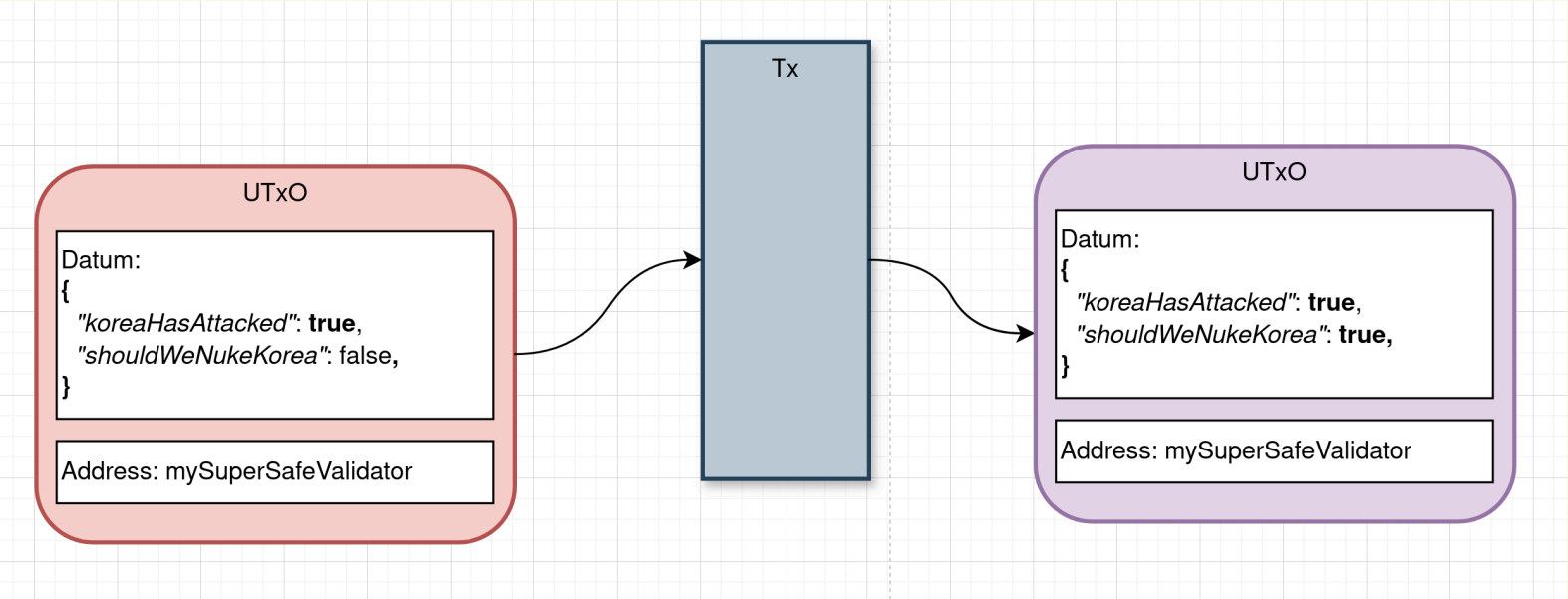
In a normal setup, a stateful smart contract would ensure that the state is propagated to itself.

Example: Attacker Setup



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



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 objects that represent value. Even ADA *can be thought of* as a special kind of native token. Native tokens are identified by their "asset id", of type $\mathrm{PolicyId} \times \mathrm{AssetName}$:

- $\mathrm{PolicyId} = (\mathbb{F}_{256})^{28}$ 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 `("0x10", "hammad")` and 1 of `("0xFF", "hammad")`, the two tokens **are not** fungible.

Minting vs Spending

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.

An instance of M identifies the $\mathrm{PolicyId}$ of the tokens that can be minted by it.

Minting vs Spending

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

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

Outlining Properties

We said earlier that we want to utilize unique tokens, to guarantee that each piece of the smart contract's state is indeed valid. We want to find a minting technique T such that:

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 . $t_1 \sim t_2$ where the relation $\sim$ means "t_1 and t_2 have the same policy id".

Achieving Properties

In particular, if we manage to mint NFTs we get (1) basically for free. However, the (2) is tougher. Usual NFT minting techniques on Cardano fall into two broad categories:

1. Use a script with a predefined timer for its validity (problem: you cannot truly enforce an amount to mint, as long as you mint multiple times before expiration)
2. Use a script parametrized with a UTxO reference. By making sure the UTxO is spent during the minting transaction, the script effectively becomes a one-time minting policy (problem: different UTxOs will produce different policy ids).

Achieving Properties

In the next slides, we will explore a minting technique capable of minting tokens with:

- The **same policy id** (thus, from the same script), but...
- ... Each with a *different asset name* (and, as such, **different asset ids**)
- **Without knowing in advance** the number of tokens we want to generate!!

The Minting Technique

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

We declare the following functions:

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

NB: With abuse of notation, we use the actual script instead of its hash when using it as argument. Obviously, in real code the hash (e.g. policy id) is used.

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

$f_{start,mint}$, when applied to an UTxO reference, returns a one-time minting policy that:

- Gets called only once
- Produces a single NFT "Auth" that is propagated throughout the rest of the process

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

```
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
  }
}✨
```

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

$f_{add,mint}$ is applied to the hash of the start script and returns a normal minting policy:

- In principle, it could be called an indefinite number of times to mint arbitrary tokens.
- However, it checks that the "Auth" NFT is present in the inputs of the transaction.

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

```
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
  }
}
```

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

$f_{add,validate}$ is applied to the start script and the add minting script. It returns a spending validator that:

- Propagates the "Auth" NFT
- Checks that its minting "counterpart" is called in the same transaction
- Propagates (via the Datum) the set of already minted assets, failing the transaction if someone is trying to mint something pre-existent (with the same asset name).

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

```
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
  }
}✨
```

The Minting Technique

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

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

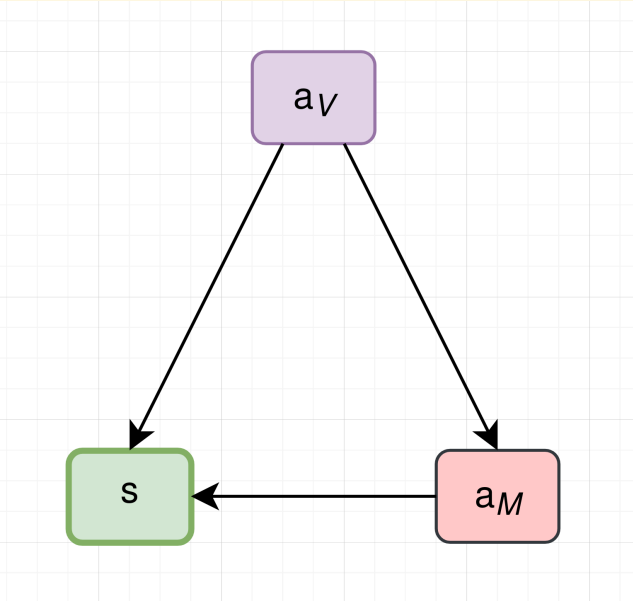
- $(s : M) = f_{start,mint}(o)$
- $(a_M : M) = f_{add,mint}(s)$
- $(a_V : V) = f_{add,validate}(s, a_M)$

The Minting Technique

- $o_1 \neq o_2 \iff f_{start,mint}(o_1) \neq f_{start,mint}(o_2)$
- $s_1 \neq s_2 \iff f_{add,mint}(s_1) \neq f_{add,mint}(s_2)$
- $(s_1, a_{M,1}) \neq (s_1, a_{M,2}) \iff f_{add,validate}(s_1, a_{M,1}) \neq f_{add,validate}(s_2, a_M$

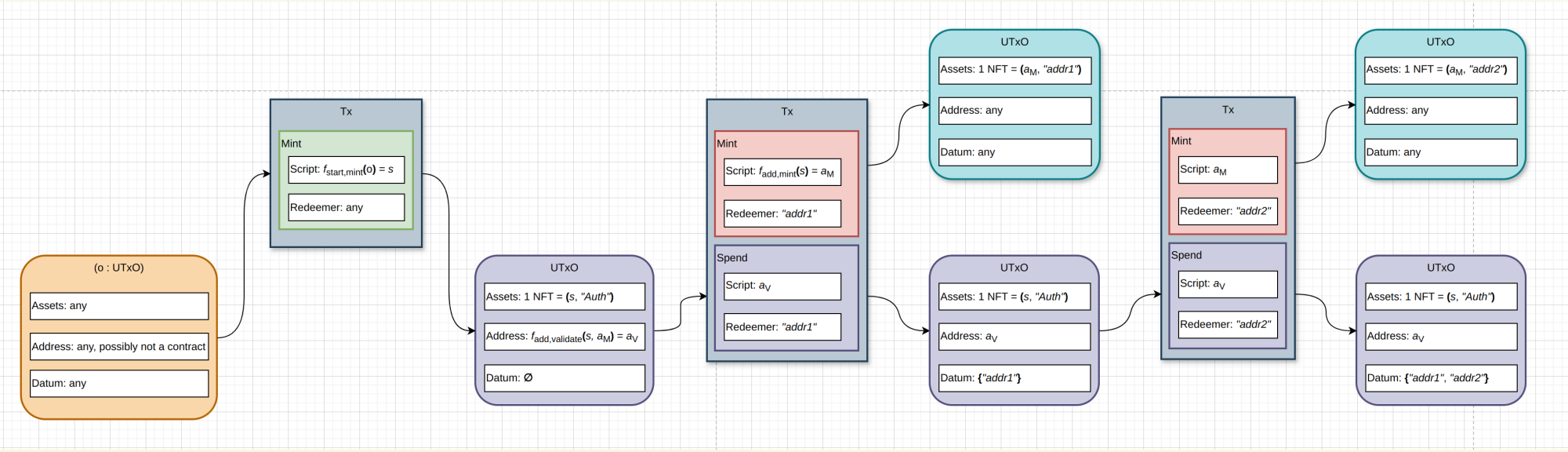
The Minting Technique

Script dependencies:



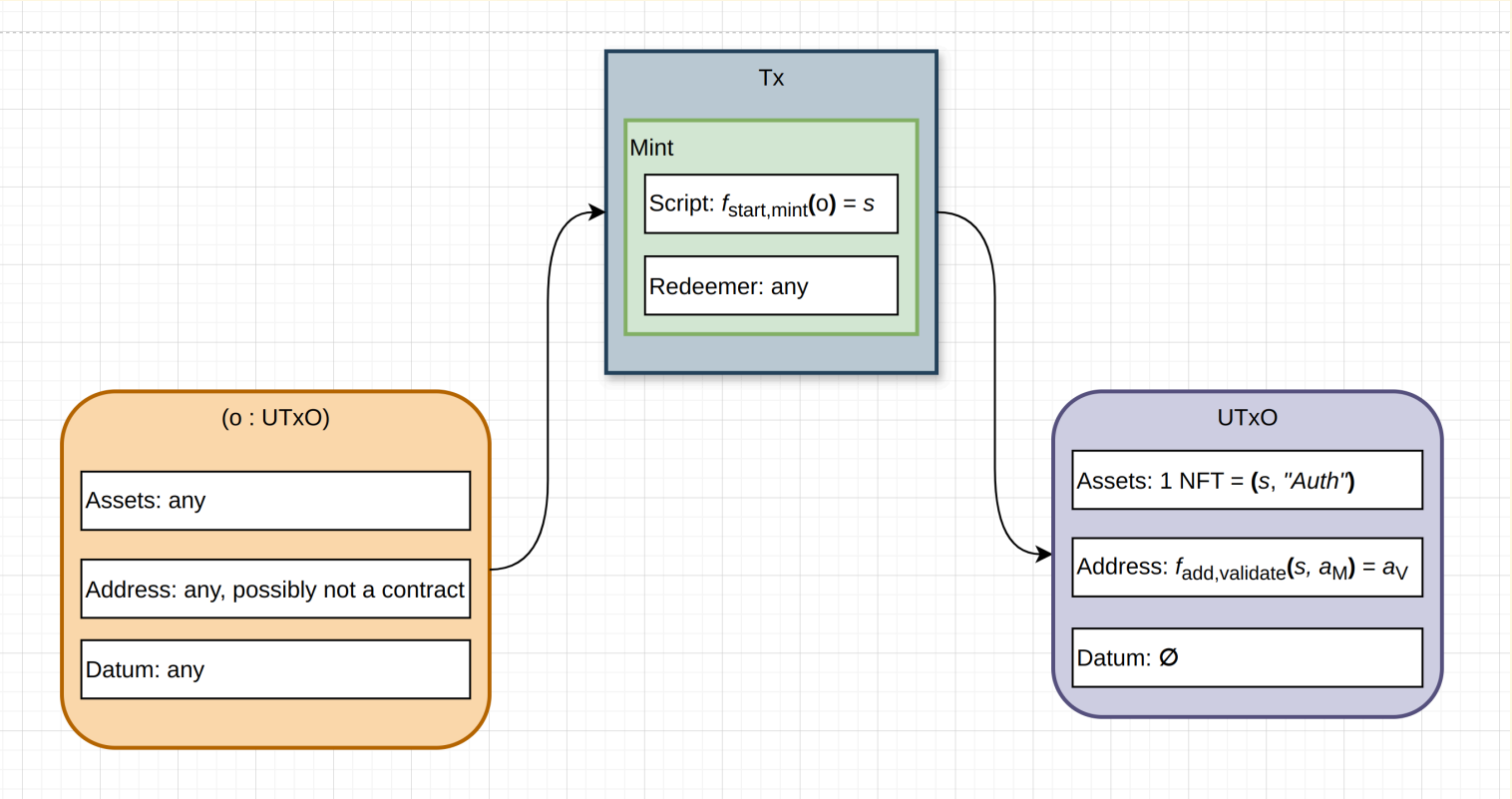
NB: "$\rightarrow$" means "depends on".

Example: Full Minting Process

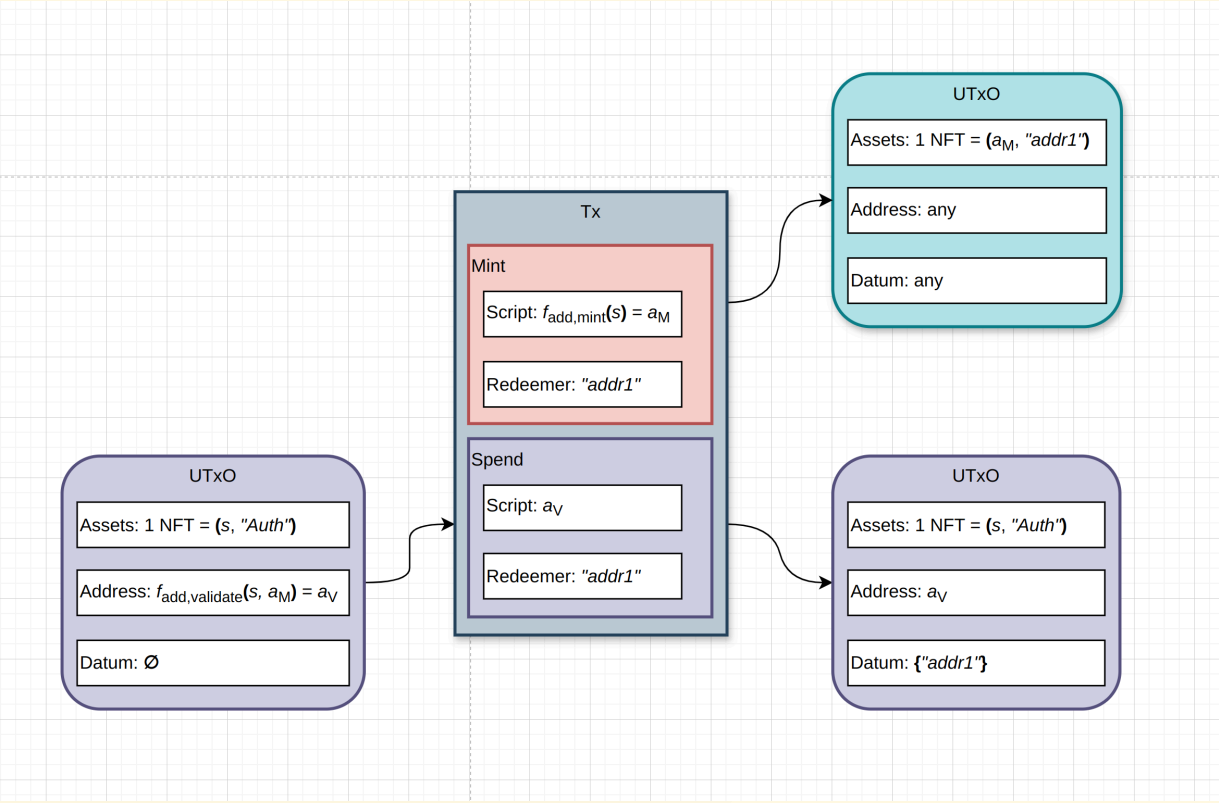


NB: UTxOs coloured cyan contain the minted NFTs that we will use to authenticate our smart contract's state.

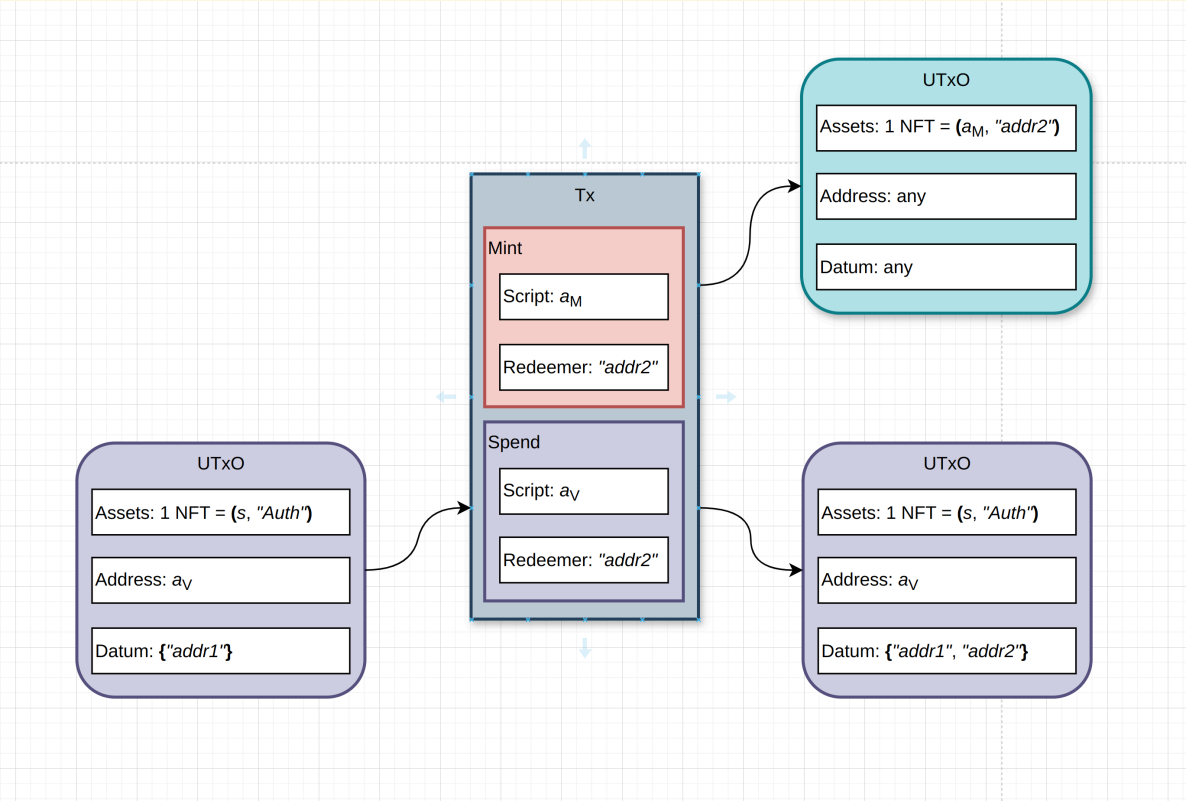
Example: Start The Minting Process



Example: Minting The 1st NFT



Example: Minting The 2nd NFT



Using The NFTs

A smart contract would just need to know the hash of a_M (the policy id of the NFTs).

An UTxO carrying a "piece" of the contract's state would be valid only if associated to one of the minted NFTs.

The smart contract would have to make sure to pay the NFT back to itself, together with the (updated) piece of state.

Critical Points

- The "Auth" NFT must be immediately paid to a_V in the same transaction it is minted. This can be enforced with a helper validator which would know about a_V (not shown here).
- The minted NFTs should be immediately paid to the smart contract. A quick solution would be to add the address of the smart contract to the parameters of $f_{add,validate}$ or better $f_{add,mint}$.

References

1. Official Cardano Documentation on Native Tokens
2. Piefayth's Blog for the original idea of using one NFT to authenticate a *single* Datum (here extended to multiple UTxOs)