CSE 597-07 Spring ‘25

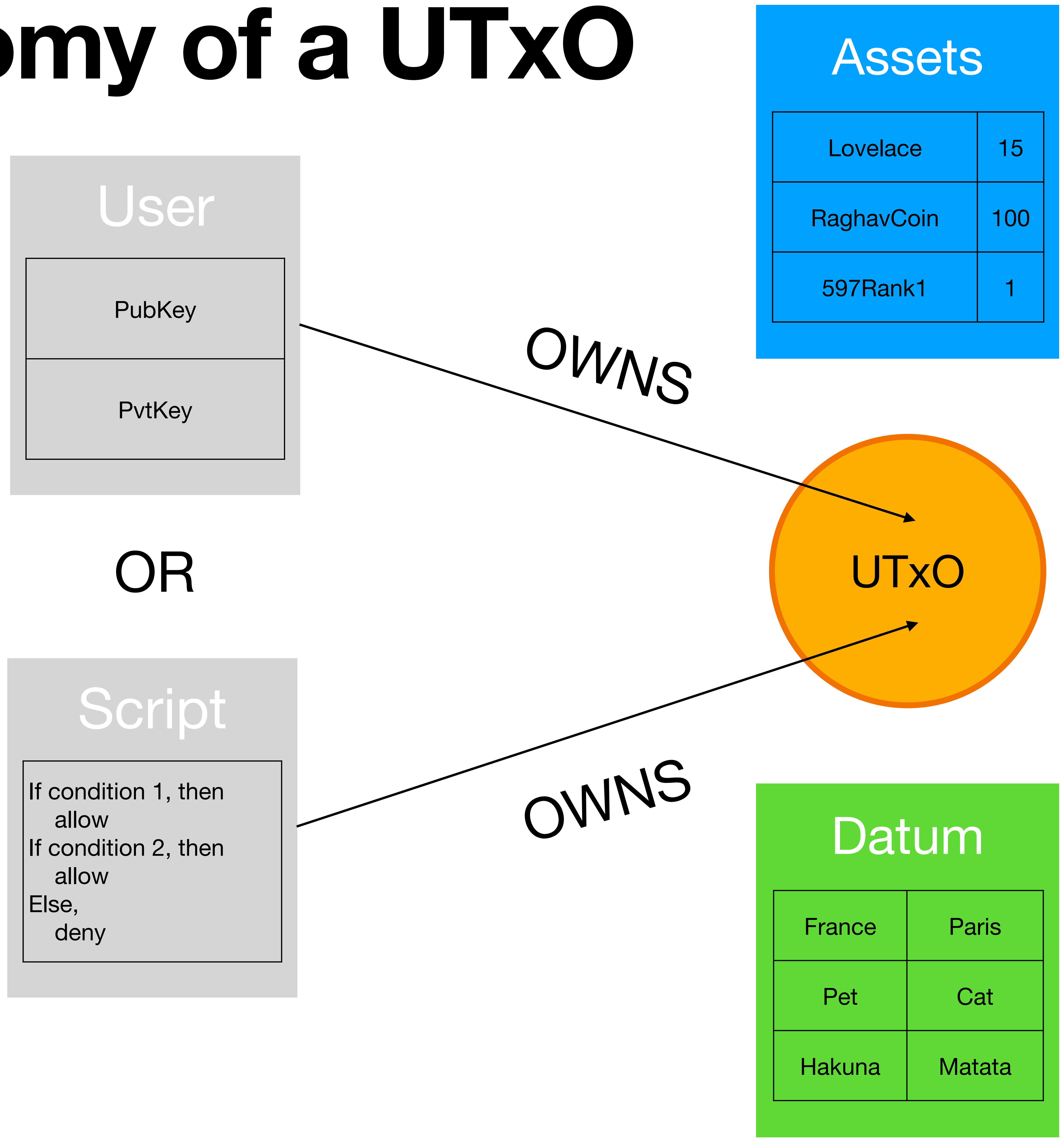
The Pennsylvania State University, University Park

Vulnerable Cardano Contracts

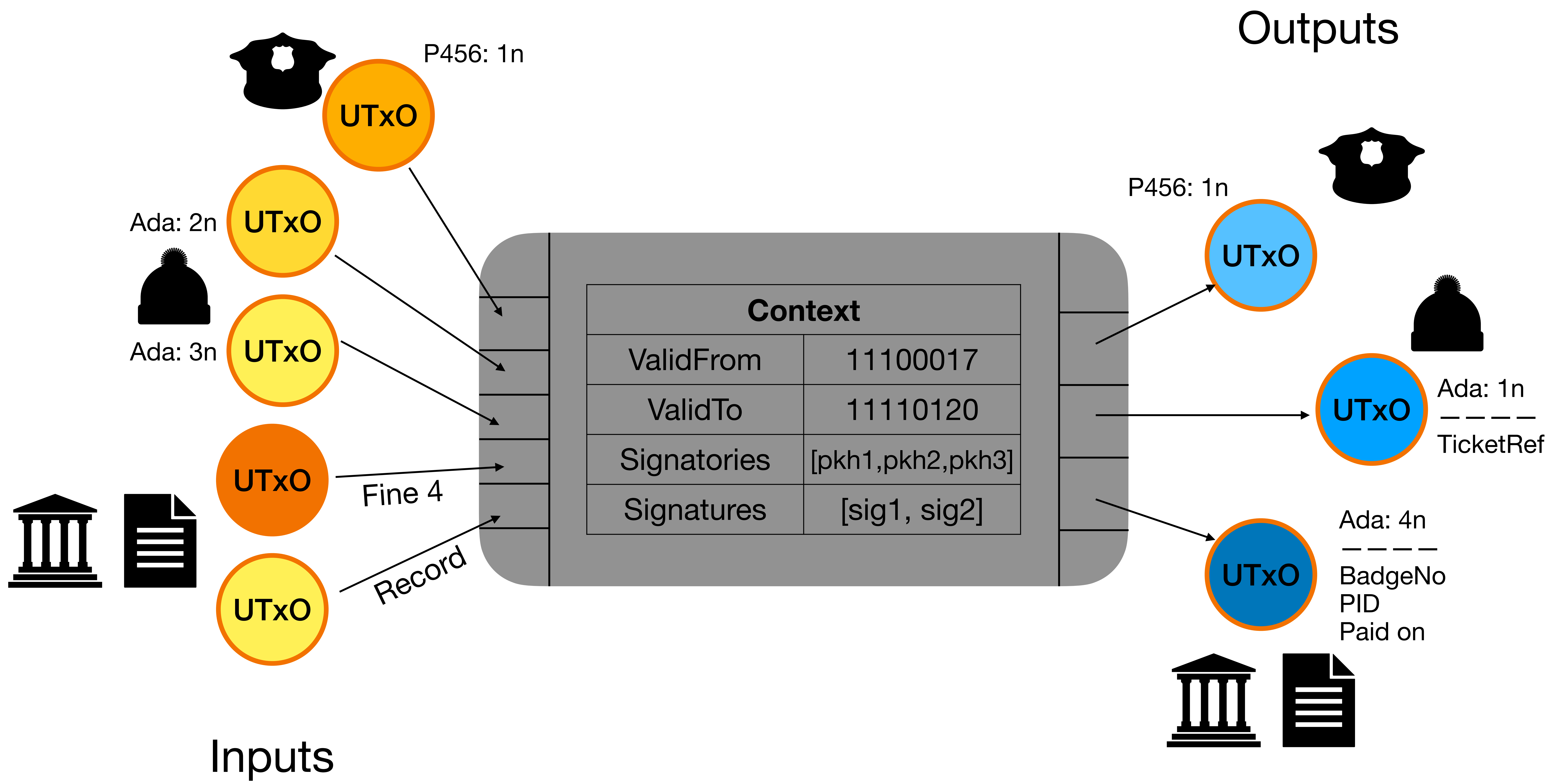
Are protocols reified in Cardano free of the unexpected?

Aakash Wagle, Raghav Kumar

Anatomy of a UTxO



Anatomy of a transaction



What all should hold?

- The number of assets (e.g. ADA, P456) used in inputs must be greater than number of assets generated in output. (Except in case of Minting)
- The validation script of every UTxO in input must evaluate to True.
  - Validation Script of a UTxO owned by a “user” just verifies that their signature is present in the transaction’s context. User’s address is generated by a “wallet” software.
  - Validation Script of a UTxO owned by a “contract” is the program whose hash is equal to the address of the “contract”
- Transaction must be within the period between ValidFrom and ValidTo
- Signatures of all “Signatories” must be present in the transaction

What’s minting?

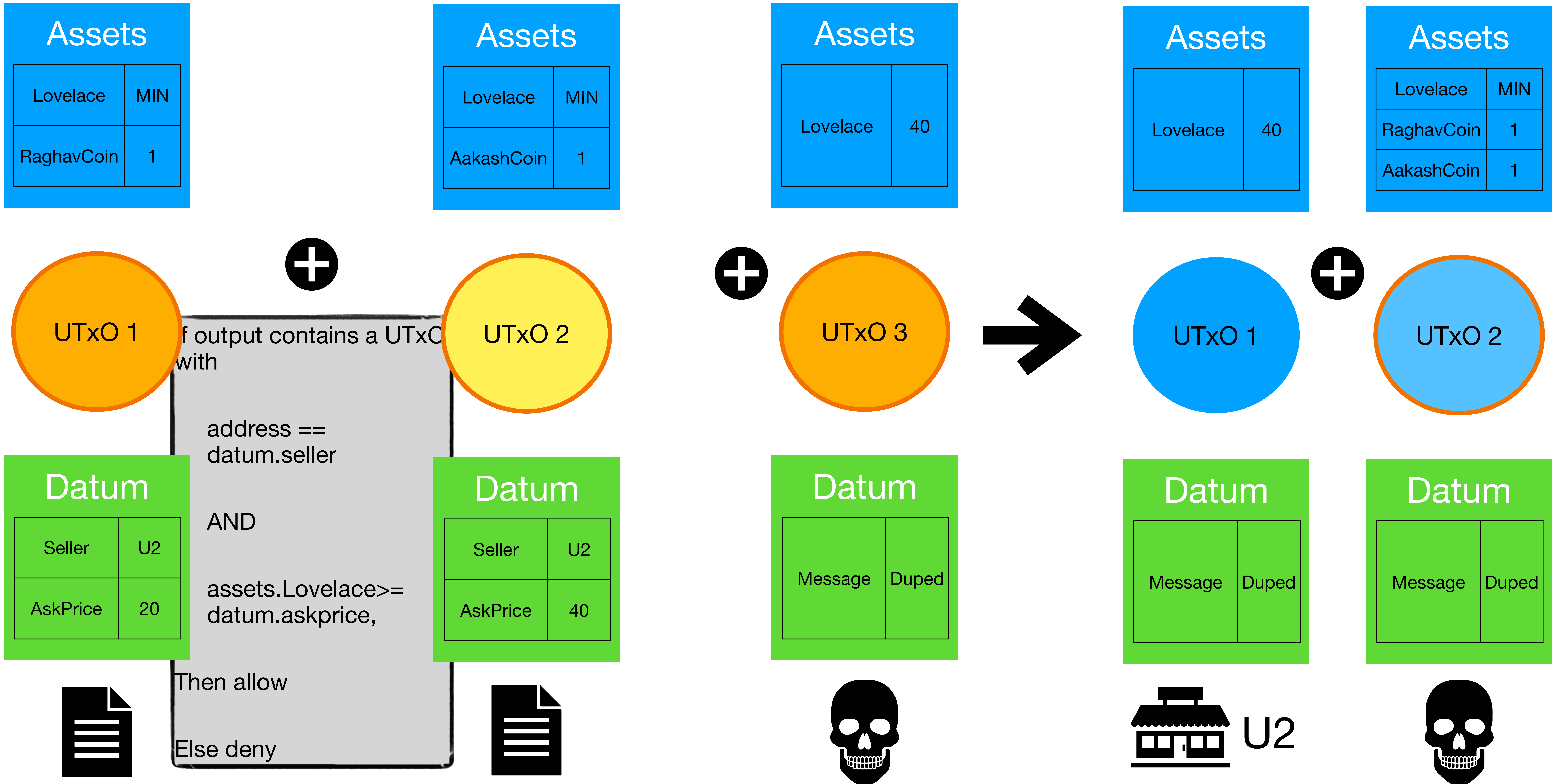
- Ada (Lovelace) is the native currency of Cardano, but we can create other first class currencies too (Think NFTs!). However, Lovelace stays first among equals.
- Any amount of a new currency can be “minted” in a transaction subject to a “minting policy” given that
  - The currency that will be minted will take a name as Hash of minting policy program + a token name
  - The minting policy are also contracts, can look at all input and output UTxOs of the transaction to allow or deny minting any amount of a token
- It has a logical counterpart “burning”, which is to delete any amount of tokens, including ADA!

What could go wrong?

Major classes we saw in the CTF

- Double satisfaction
- Denial of service
- Token stealing
- Trusting the datum
- Address has two parts
- Time handling

Example of Double Spending



```
type Datum {
  seller: Address,
  price: Int,
}

validator {
  fn buy(datum: Datum, _redeemer: Void, ctx: ScriptContext) -> Bool {
    expect Spend(_output_reference) = ctx.purpose
    expect Some(_seller_output) =
      list.find(
        ctx.transaction.outputs,
        fn(output) {
          (output.address == datum.seller)? && (lovelace_of(output.value) >= datum.price)?
        },
      )
    True
  }
}
```

Picture 1: No bugs, no typos, no incorrect logic, yet unsafe

What could go wrong?

In principle

- Any entity can come up with a “protocol” to represent their business process and use “Validators”, “Minting Policies” and “Assets” to represent state and manage state transition.
- The state transition logic for Cardano is written in a Functional paradigm (Haskell sits at it’s bedrock)
- Unlike Ethereum (which is the most popular rival, we’re not just picking on it), in Cardano network whether a transaction will succeed or fail can be discovered even before it is published on network.
- As validators and minting policies grow in a system, unexpected interactions may result in stolen assets or unusable states

What we did in part 1?

Also, what did we learn?

- We solved 10 level of VaccumLabs Cardano CTF
- From level 1 we learnt about how basic double satisfaction works
- From level 2 we learnt that Cardano contracts can't see current time and using the lower or upper limits of transaction validity can lead to losses
- From level 3 we learnt that the datum of any UTxO cannot be trusted. Any one can create a UTxO with any datum and send it to any address
- From level 4 we learnt that UTxOs with large datum are too big to use, leading to Denial of Service (DoS)

What we did in part 1?

Also, what did we learn?

- From level 5 we learnt that addresses in Cardano have two parts, and while only the first one controls authentication and logic, simply checking that only one UTxO owned by a contract's address is present in the transaction input is not sufficient to prevent double satisfaction
- From level 6 we learnt that UTxOs with too many tokens are too big to use, leading to DoS
- From level 7 we learnt that if validators are not carefully written one can transfer tokens, that signify a state in a protocol, to any address and reuse them to gain unauthorised access
- From level 8 we learnt that the given lending protocol can be used to hold the borrower's collateral as hostage without paying them the loan amount as a result of trusting the datum
- From level 9 we learnt that not only can two validators be susceptible to double satisfaction, but minting policies can also fall prey to double satisfaction allowing attacker to mint new tokens

What we did in part 1?

Also, what did we learn?

- From level 10 we learnt that addresses in Cardano have two parts, and the second part can be malformed thus preventing payments to oneself while still controlling a UTxO From level 6 we learnt that UTxOs with too many tokens are too big to use, leading to DoS

- Obviously we knew all the validators had “some” flaw and required us to manually probe the code and make sense of it.

What could go wrong?

Major classes we saw in the CTF

- Double satisfaction - level 1,5 & 9
- Denial of service - level 4 & 6
- Token stealing - level 7
- Trusting the datum - level 3, 8
- Address has two parts - level 5 & 10
- Time handling - level 2

Questions to ask

Are these good, important and unanswered questions?

- Is the Plutus Core platform, that runs the validator and minting scripts, and the functionalities it provides free of bugs?
  - we didn’t find a CVE related to the Plutus Core platform.

- Given a protocol that specifies the states and state transitions of a business process, expressed in Cardano’s framework, can we enumerate all possible states and weed out unacceptable states? What is the maximal complexity of a system for which all states can be enumerated before the number explodes?

- Given a protocol that specifies the states and state transitions of a business process, expressed in Cardano’s framework, as well as known undesired states

  - How can we encode the undesirable states?

  - How can we detect their presence in the enumerated states of the given protocol without enumerating all the states?

How to detect the unexpected?

- Plutus Core (Cardano's on-chain smart contract system) is functional, this
  - Makes a set of smart contracts amenable to formal verification
  - Can give sense of security against unknown unknowns
- In addition to formal verification, can we use techniques such as fuzzing to generate test cases in the negative space left by designers
- Cardano Smart Contracts can be written in many languages - Aiken, Python, Rust, but all of them are compiled to Haskell.
- We did it manually, but can we automate detecting the unexpected?

Security Analysis Techniques

PennState

- Testing/Fuzzing (Dynamic Analysis)
- Symbolic Execution
- Concolic Execution
- Static Analysis
- Formal Verification

Automatic test case generation

Static analysis

Program verification

→

Fuzzing

Dynamic symbolic execution

Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

What we plan to do next?

- Find out how a system made up of validators, minting policy and tokens can be represented formally
- Design a fuzzer that, given a system, generates transactions by mutating inputs and outputs
    - Maybe we can use LLMs

Lit review

- Functional Blockchain Contracts
- Formal Specification of the Cardano Blockchain Ledger, Mechanized in Agda
- LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models
- VaccumLabs Cardano CTF
- VaccumLabs Cardano CTF Hints and Solutions
- VaccumLabs Blog
- VaccumLabs Audit Reports
- Translation Certification for Smart Contracts
- Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection
- Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives
- VulnScan GPT: a new framework for smart contract vulnerability detection combining vector database and GPT model