

# Lecture Seven

"Constraints Liberate, Liberties Constrain"  
— Runar Bjarnason

## 1. Introduction

*Note: You always write your introduction last.*

## 2. Committal Schemes

We see these types of techniques used within network protocols quite a bit. I believe (if I remember correctly from my Cryptography module, six years ago) the Diffie Hellman protocol uses a committal scheme, or some kind of assumption. Anyway, what we're essentially doing is communicating with a potential untrustworthy party in a manner where they cannot read what we send them until the appropriate point within the protocol.

For example, consider the following game where the rules are: if you select the same value as your opponent, you win, else, you loose. However, this game must be played synchronously, which means you have to pick a value first, *but if your opponent could see what you selected, that would be a problem*. Plus, you both cannot trust one another. So, how do we get around this little dilemma?

There are a lot of variants of committal schemes, but the simplest to explain is as follows:

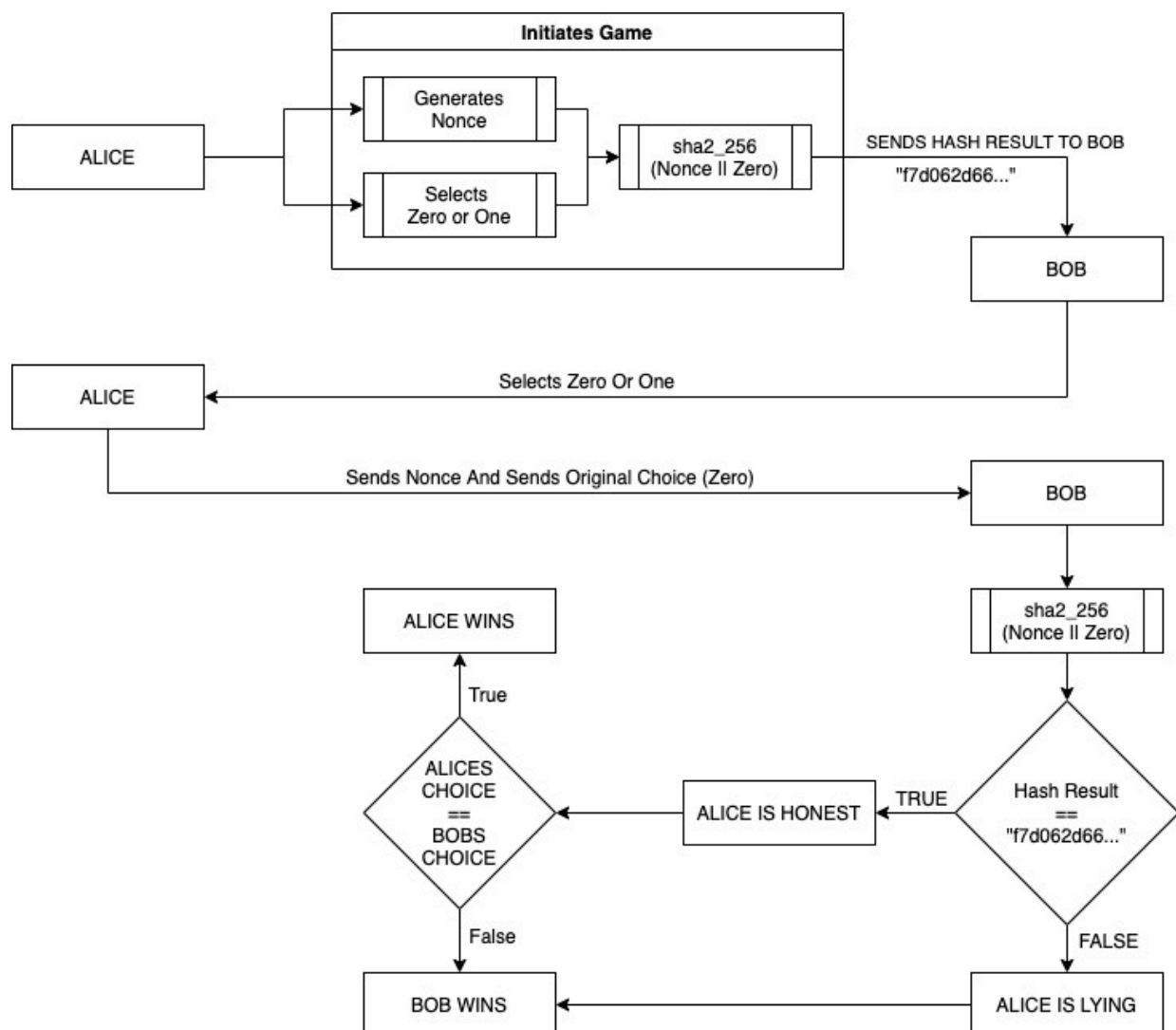
*Note: You are called Alice, and your opponent is called Bob.*

1. You attempt to initiate a game (by betting 500 ADA).
2. An opponent matches your bet of 500 ADA before a pre-defined clock runs out.
3. You are both now playing one another, since you initiated the game, you go first.
4. You generate a pseudo-random number, let's call this Q.
5. You select a value from the set of {0, 1}, let's call this Y.
6. You use a hashing algorithm (say: sha2\_256) to encode Q concatenated with Y:
7. `Initial Selection <- sha2_256(Q ++ Y)`
8. `Initial Selection = "3Ef7d062d66..."`
9. You send Bob the value of the initial selection.
10. Bob now sends his selected value to you.
11. Now, you send Q and Y to Bob.

12. IFF Bob runs the `sha2_256(Q ++ Y)` and sees the result you previously sent him, he knows you are acting honestly (provably fair).
13. Bob then knows if he has won or lost.
14. You also know if you have won or lost.

*Note: a point failed to be addressed within the lecture is: what if Bob decided to pretend Alice is lying, even when she is not?" I understand that is regardless in relation to running a smart contract, because it will do as you tell it to. But, if this was a simple network protocol, couldn't Bob use something like Wireshark to modify the contents of the payload in the package he may send back to Alice to say that he had lost?*

Please see the below image for additional details.



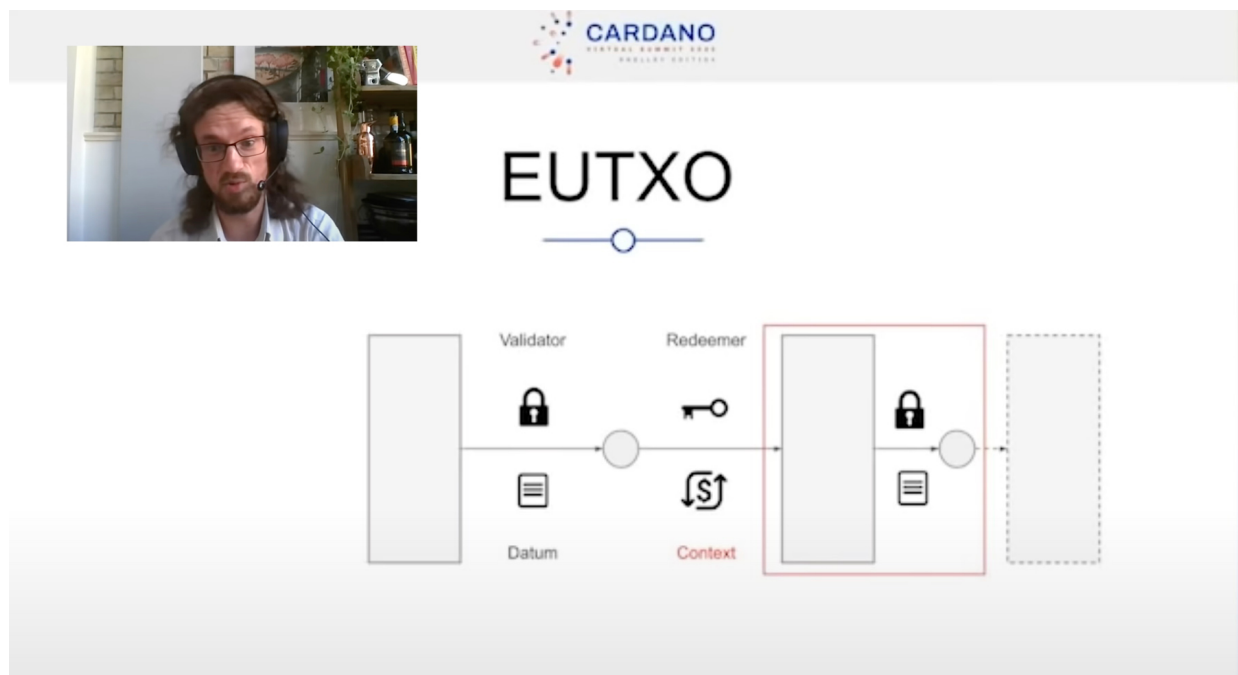
### 3. Applications: State Machines

What is an application? Usually it is something that runs and applies itself to change the state of something in a useful manner. Haskell wasn't much good for that, up until recently.

Plutus-core which is implemented by compiling it from Haskell is what is used to trigger the validate transactions (of many kinds) on the Cardano blockchain.

Instead of planning to build arbitrary applications, in this instance, although arbitrary logic is executed by any given script, it's very much a case of freedom within limitations. When you are building safety-critical systems or financial systems, you really do want to think extremely carefully about how you're going to implement any given system (and what tools you're going to use).

The fundamental concept of the Blockchain Ledger is that transactions have outputs with AssetClasses tied to them. So, we have a bunch of transactions on-chain (txId) that have (potentially) a bunch of outputs (known as Unspent Transaction Outputs), until somebody comes along and spends it. But not just anybody, it has to be the wallet that holds the private key in order to produce a digital signature AND satisfy the limited arbitrary logic implemented by that script (in order to validate). **Right, so nothing new here, it's the same as Bitcoin Script, having a validator, having a redeemer, you've heard it all before.** Except, you do also understand that we have a datum output (because, well - it will become apparent as to why this may come in useful in a moment). Furthermore, we are able to access the contexts, which provides us with all kinds of information about the Tx.



As is demonstrated in the above image, you can essentially initiate a chain of transactions using the datum output and the state. Further, since

	A :	A:0	A:1
B :	S0	-	-
B:0	-	S1	S2
B:1	-	S2	S1

### Quick Explanation!

Right, so for some reason I had a bit of a brain fart when implementing this table for the first time. It's been **quite a while** since I've had to draw out a state transition table. But, what it's essentially saying is that you:

- Can only start at S0
- S1 is a final state (ALICE WINS)
- S2 is a final state (BOB WINS)
- Assuming that Bob is (as the last player to 'roll') is always going to tell the truth. All we would have to do to ensure he wasn't pretending to win and causing a dispute would be to apply the same hashing technique to his choice as Alice did to hers.

### Some Code! Let's work through the EvenOdd Example

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE DeriveAnyClass      #-}
{-# LANGUAGE DeriveGeneric       #-}
{-# LANGUAGE FlexibleContexts    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoImplicitPrelude   #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell     #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}

module Week07.EvenOdd
( Game (..)
, GameChoice (..)
, FirstParams (..)
, SecondParams (..)
, GameSchema
, endpoints
) where

import           Control.Monad           hiding (fmap)
import           Data.Aeson              (FromJSON, ToJSON)
import qualified Data.Map                 as Map
import           Data.Text                (Text)
import           GHC.Generics            (Generic)
import           Ledger                   hiding (singleton)
import           Ledger.Constraints       as Constraints
import qualified Ledger.Typed.Scripts     as Scripts
import           Ledger.Ada               as Ada
import           Ledger.Value
import           Playground.Contract      (ToSchema)
import           Plutus.Contract         as Contract
import qualified PlutusTx
import           PlutusTx.Prelude        hiding (Semigroup(..), unless)
import           Prelude                  (Semigroup (..), Show (..), String)
import qualified Prelude

```

What we're essentially doing is implementing that state transition table / DFA as a game where player one rolls a two sided die, or chooses a binary value after the game has been init'd. Then player two does the same. If player one has picked the same value as player two, player one wins. If Player two has picked a different value, then player two wins! Fairly straight forwards...

So, this is a parameterised smart contract. We would like to inject the Game data structure

and importing a bunch of stuff - this is just the top of the module, standard imports, module name and data structure.

```

data Game = Game
  { gFirst      :: !PubKeyHash
  , gSecond     :: !PubKeyHash
  , gStake      :: !Integer
  , gPlayDeadline :: !POSIXTime
  , gRevealDeadline :: !POSIXTime
  , gToken      :: !AssetClass
  } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)

PlutusTx.makeLift ''Game

data GameChoice = Zero | One
  deriving (Show, Generic, FromJSON, ToJSON, ToSchema, Prelude.Eq, Prelude.Ord)

instance Eq GameChoice where
  {-# INLINABLE (==) #-}
  Zero == Zero = True
  One  == One  = True
  _    == _    = False

PlutusTx.unstableMakeIsData ''GameChoice

data GameDatum = GameDatum ByteString (Maybe GameChoice)
  deriving Show

instance Eq GameDatum where
  {-# INLINABLE (==) #-}
  GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')

PlutusTx.unstableMakeIsData ''GameDatum

data GameRedeemer = Play GameChoice | Reveal ByteString | ClaimFirst | ClaimSecond
  deriving Show

PlutusTx.unstableMakeIsData ''GameRedeemer

{-# INLINABLE lovelaces #-}
lovelaces :: Value -> Integer
lovelaces = Ada.getLovelace . Ada.fromValue

{-# INLINABLE gameDatum #-}
gameDatum :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe GameDatum
gameDatum o f = do
  dh      <- txOutDatum o
  Datum d <- f dh
  PlutusTx.fromBuiltinData d

```

## **Creating a data type: Game:**

Within the game data structure, we're assigning names to the two players. We're also providing times (for timeouts, as are important to avoid funds getting locked within a contract. gToken is an NFT that enables us to allow this script to live at a unique address.

*Working on Catalyst Proposal - Will be back to this soon.*

*Working on Alonzo Purple Test Net - Will be Back To This Soon!*