Lab: Step by Step towards Programming a Safe Smart Contract

May 12, 2015

Contents

1 Introduction

Cryptocurrencies, including Bitcoin, Ethereum, and many others, are an exciting new technology. They are experimental distributed systems that allow users to manipulate virtual currency. Actual stored wealth and monetary value are at stake! Ethereum is the first embodiment of the more general idea: it provides an expressive and flexible programming environment for controlling and interacting with money.

This tutorial is intended for instructors who wish to conduct a smart contract programming lab, or students/developers who want to learn about smart contract programming.

The first part of this lab consists of step-by-step examples illustrating basic design of functional smart contracts. We highly recommend you take a hands-on approach, and interact with these smart contract examples using the Ethereum simulator! The accompanying materials to this guide contain everything you need to get started with experimenting, including a virtual machine image, basic instructions, and a language reference.

The second part of this lab focuses on designing smart contracts that achieve their intended goals, and are robust to attacks. Although our lab makes us of a simulator, the smart contracts you write can also be used in the live Ethereum network[1] The basic concepts we discuss apply to other cryptocurrencies as well (including Bitcoin), so most of the skills you learn will be transferable.

Smart contract design is inherently security-oriented. Contracts are "play-for-keeps, since virtual currencies have real value. If you load money into a buggy smart contract, you will likely lose it. Unlike other hands-on labs in cryptography (e.g., sending encrypted emails with GPG), where actual attacks are unlikely or hard to observe, the attackers in a cryptocurrency are much more apparent.

Smart contract design also requires economic thinking. We use a running example about a rock-paper-scissors game. To help keep incentives in focus, we reward the winner with a monetary prize, so both participants have a stake in the outcome. Other, more clearly useful applications include derivative financial instruments, for example, that allow people to buy or sell insurance based on the price of another cryptocurrency, or based on other events that can be logged by the network. Smart contracts can also be used to raise crowdfunding money with a Kickstarter-like assurance contract, that gives contributors a refund if a donations target isnt reached. In all of these applications, we will want to guarantee that the smart contracts are "fair" and difficult and unprofitable to exploit.

2 Basic Smart Contract Design

In this section, we demonstrate basic concepts of smart contract design by discussing several working examples. We assume the reader has read the introduction of the accompanying programming tutorial and knows where to find the language reference.

2.1 Simple Serpent Contract Example - Namecoin

As four first example, we will make a contract that is normally called "namecoin". Essentially, it allow for us to create a basic "write-once" key-value store. A key value store is a data storage structure that allows for us to associate a key with a value, and look up values based on their keys. This contract will have two different functions to call. The first is the

[1] At the time of this writing, the only live Ethereum network is a test network, since the main network has not yet launched.

key-value registration function and the second is a function that retrieves a value associated with a provided key.

The first function we will look at is `register(key, value)`, which takes a key and value and associates them with each other:

```
def register(key, value):
        if not self.storage[key]:
                self.storage[key] = value
                return(1)
        else:
                return(-1)
```

Lets break this down. This contract essentially consists of an if-else statement. First, we check to see if the key-value is already in storage. We can use the not statement to check if nothing is stored. So if nothing is stored, we will store the value in the persistent key-value store `self.storage[]`. However, what if the key is already taken? We can't just overwrite someone else's key! So, we just return -1.

Now that we know how to store values, we need to be able to retrieve them. For that we use the `get(key)` function:

```
def get(key):
        if not self.storage[key]:
                return(-1)
        else:
                return(self.storage[key])
```

This function will simply return the value associated with the key. This function is very similar to our storage function. However, this time we don't store anything. If there is nothing associated with the key, we return -1. Otherwise, we return the value that is associated with the key.

The complete code for namecoin is below:

```
def register(key, value):
        if not self.storage[key]:
                self.storage[key] = value
                return(1)
        else:
                return(-1)

def get(key):
        if not self.storage[key]:
                return(-1)
```

```
        else:
                return(self.storage[key])
```

2.2 Basic Serpent Contract Example - Easy Bank

Let's take a quick look at an Easy Bank example from KenK's first tutorial. [1] A contract like this allows for a fully transparent bank to function with an open ledger that can be audited by any node on the network (an ideal feature for ensuring banks aren't laundering money or lending to enemies of the state.)

Before looking at the code for the contract, let's define our "easy bank" further. Our bank will be using its own contractual currency and not Ether (which we will discuss and implement in a later contract example). So, creating the currency is done within our contract. Now that we know what our bank does (creates and sends a currency that is exclusive to the contract), let's define what the contract must be capable of doing:

1. Setup at least one account with an initial balance of our contract-exclusive currency
2. Take funds from one account and send our currency to another account

```
def init():
        #Initialiaze the contract creator with 10000 fake dollars
        self.storage[msg.sender] = 10000

def send_currency_to(value, destination):
        #If the sender has enough money to fund the transaction, complete it
        if self.storage[msg.sender] >= value:
                self.storage[msg.sender] = self.storage[msg.sender]  - value
                self.storage[destination] = self.storage[destination] + value
                return(1)
        return(-1)

def balance_check(addr):
        #Balance Check
        return(self.storage[addr])
```

So what's going on in this contract? Our contract is divided into two methods, let's take a look at the first method:

```
def init():
        #Initialiaze the contract creator with 10000 fake dollars
        self.storage[msg.sender] = 10000
```

Our `init` method, from a general perspective, initializes the contract creator's account with a balance of 10,000 dollars. In our Ethereum contract, storage is handled with key value pairs. Every contract has their own storage which is accessed by calling `self.storage[key]`. So in our example the easy bank's contract storage now has a value of 10,000 at key msg.sender (we'll identify what this is in a moment).

Awesome. So who is `msg.sender`? `msg.sender` is the person who is sending the specific message to the contract - which in this case is us. `msg.sender` is unique and assigned and verified by the network. Now we have a heightened understanding of `init`, so lets look at our send method.

```
def send_currency_to(value, destination):
        #If the sender has enough money to fund the transaction, complete it
        if self.storage[msg.sender] >= value:
                self.storage[msg.sender] = self.storage[msg.sender]  - value
                self.storage[destination] = self.storage[destination] + value
                return(1)
        return(-1)
```

The `send_currency_to` function takes in two parameters. The first is the value in Wei that we are sending. The second is the public key of the address we are sending it to.

First, we check that the person trying to transfer money has enough in their account to successfully complete the transfer. If they do, we complete the transaction by removing the value from the sender's account and adding to the destination's account, and we return 1. If they do not have enough money, we simply return -1, denoting that the transaction failed.

The `balance_check` function simply returns the value currently stored in the provided public key's account.

Great! We have officially worked our way through a very basic contract example! Try to think of ways that you could improve this contract, here are some things to consider:

- What happens when the value exceeds the amount setup in the $from$ account?
- What happens when the value is negative?
- What happens when value isn't a number?

2.3 Moderate Serpent Contract Example - Bank

Let's take a quick look at a smart contract that implements a bank. A contract like this allows for a fully transparent bank to function with an open ledger that can be audited by any node on the network (an ideal feature for ensuring banks aren't laundering money or lending to enemies of the state.)

Before looking at the code for the contract, let's define our bank further. Our bank will allow users to store Ether in units of Wei. It must be capable of the following actions allowing users to:

1. Deposit money into their account.
2. Transfer money from their account to another account.
3. Withdraw their money.
4. Check their balance.

```
#Deposit
def deposit():
        self.storage[msg.sender] += msg.value
        return(1)

#Withdraw the given amount (in wei)
def withdraw(amount):
        #Check to ensure enough money in account
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                #If there is enough money, complete with withdraw
                self.storage[msg.sender] -= amount
                send(0, msg.sender, amount)
                return(1)

#Transfer the given amount (in wei) to the destination's public key
def transfer(amount, destination):
        #Check to ensure enough money in sender's account
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                #If there is enough money, complete the transfer
                self.storage[msg.sender] -= amount
                self.storage[destination] += amount
                return(1)

#Just return the sender's balance
def balance():
        return(self.storage[msg.sender])
```

So what's going on in this contract? Our contract is divided into four methods, let's take a look at the first method:

```
#Deposit
def deposit():
        self.storage[msg.sender] += msg.value
        return(1)
```

This method is a relatively simple method. It allows for a user to deposit funds into their account. Similar to our Namecoin example, we are using `self.storage[]` so we can associate the address of the person who owns the account with the value of the ether they are storing in their account. We do this on the third and fourth lines, where we use `msg.sender` as the key. `msg.sender` stores the address of whomever sent the command. The other built-in variable reference we use is `msg.value`. This stores the amount of ether (measured in wei) that is sent with the transaction. When ether is sent with a command to a contract, it is stored by the contract. Therefore, we just need to account for how much each person has in their account, so we can provide up to their account's balance on demand. This is stored as the value we are associating with the key in `self.storage[]`.

This method first adds the value sent with the deposit to the person's account (stored in `self.storage[]`). Then, it returns 1. Since something will always be deposited, there isn't really an error condition that can occur (where we may return something else).

```
#Withdraw the given amount (in wei)
def withdraw(amount):
        #Check to ensure enough money in account
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                #If there is enough money, complete with withdraw
                self.storage[msg.sender] -= amount
                send(0, msg.sender, amount)
                return(1)
```

This method here is doing essentially the opposite of the of the deposit method. Here we are taking `amount` ether out of our account and sending it to ourself. First, we check to make sure they have enough to withdraw. If we don't, we return -1. We could technically return anything, but in this guide, we use negative numbers to symbolize that there is an error. If they do have enough wei in there account, we simply subtract that from that from their account (still using `msg.sender` as a key). However, how do we send that wei back to the account owner? Simple! We simply use the send function. The send function takes three parameters. First, it takes the amount of gas we are sending with the contract. Since we are going to assume that this is being refunded to a user and not another contract, we don't need to send any gas with it. The next parameter is the address that we are sending this money to. Since `msg.sender` own the account, we are going to send this ether back to msg.sender. Next, since we have shown that there is at least the requested amount in

the account, we will send that amount to them. Finally, we will return 1 to show that the operation completed successfully.

Finally, lets look at our transfer method:

```
#Transfer the given amount (in wei) to the destination's public key
def transfer(amount, destination):
        #Check to ensure enough money in sender's account
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                #If there is enough money, complete the transfer
                self.storage[msg.sender] -= amount
                self.storage[destination] += amount
                return(1)
```

This method allows for someone to move ether from one account to another. It works very similarly to the deposit and withdraw methods we have already looked at. The only difference with this one is that one of the parameters is called "destination". This parameter takes in the public address of the person's account we are sending the money to. Remember that we use the public address as the key in our `self.storage[]` key-value store.

In this method, we first check to make sure there is enough money in the account. If there is, we transfer the funds between the accounts.

I will leave it as an exercise to you to see how the balance function works.

2.4 Student Exercise - Mutual Credit System

Now that you have looked at a few examples of ethereum contracts, it's time for you to try it for yourself. We are going to continue with the idea of a banking contract, but we are going to change it up. We want you set up what we are a calling a "Mutual Credit System". In this system, everyone will start off with a balance of zero. When you make a transaction, you pay using debt, so your balance becomes negative. The person you pay gains credit, so his balance becomes positive. After all of the transactions, people will have varying amounts of money, some positive, some negative. We are limiting the amount of debt one is allowed to spend to 1000 credits. Note that we will be using our own currency, not ether.

To complete this task you will need to use `self.storage[]` for persistent storage. You will need to create two methods. The first "transfer" which accepts a public key and a value. This will transfer the credits from the `msg.sender`'s account to the public key's account (return 0). If the account that is sending the credits will exceed 1000 credits of debt, the transaction should be declined (return -1).

You will also need to implement a balance method that takes in the public key the sender wants the balance of, and returns the balance of that public key.

For more information on Mutual Credit Systems, visit `http://p2pfoundation.net/Mutual_Credit`.

3 Designing a Secure Contract.

In this section, we'll explore the security and incentive alignment pitfalls in designing a smart contract. We'll use an easy-to-understand application as a running example, based on a Rock-Paper-Scissors game. We then analyze a plausible (but subtly buggy) initial implementation, pointing out its flaws. Mistakes resembling these were actually observed in our Smart Contract Programming Lab in "CMSC 414 - Undergraduate Security". This section is centered around the exercises. We provide hints to guide the reader towards discovering how to improve on them. Our "reference" solution can be found in the accompanying materials.

3.1 Corner Cases in Coding State Machines

The first contract design error we will talk about is contracts causing money to disappear. Some contracts require the participants to send an amount of money to enter the contract (lotteries, games, investment apps). All contracts that require some amount of money to participate have the potential to have that money lost in the contract if things don't go accordingly. Below is the *add_player* function from our RPS contract. The function adds a player and stores their unique identifier (`msg.sender`). The contract also takes a value (*msg.value*) that is sent to the contract. The value is the currency used by Ethereum, ether. Ether can be thought of as similar to bitcoins. Bitcoins are generated by mining, and can be used for trading and to pay transaction fees; ether is also mined, and is used as the currency to fuel all contracts as well as the currency that individuals will trade within contracts. Let's dive in and see if we can find a contract theft error in the `add_player` contract below:

```
def add_player():
        if not self.storage["player1"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] =
                                self.storage["WINNINGS"] + msg.value
                        self.storage["player1"] = msg.sender
                        return(1)
                return (0)
        elif not self.storage["player2"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] =
                                self.storage["WINNINGS"] + msg.value
                        self.storage["player2"] = msg.sender
                        return(2)
                return (0)
        else:
                return(0)
```

In this section, a user adds themselves to the game by sending a small amount of ether with their transaction. The contract takes this ether, stored in `msg.value`, and adds it to the winnings pool, the prize that the winner of each round will receive. Let's consider two scenarios our contract currently allows 1) a potential entrant sends too much or too little ether, 2) there are already two participants, so additional players send transactions to join, but are not allowed. In both of the following scenarios the contract will keep their money. If someone sent too much or too little to enter they will not be added as a player, but their funds will be kept. Even worse, if the match is full any person who tries to join (they have no way of knowing it is full) will pay to play but never be added to a game! Both of these errors will cause distrust in our contract, eventually resulting in the community not trusting this particular contract and, more importantly, this contract's author - you.

So how do we fix these issues? It seems like our contract needs the ability to give refunds to users who try to sign up too late. Think about how you would do this. Go ahead and try it and see if your idea works! Are there any other edge cases where issuing a refund should be considered? Look at the section "Sending Wei" in the Serpent Tutorial for inspiration.

3.2 Implementing Cryptography

Cryptography is often the first line of defense against security hazards in smart contract programming. In the example above, players reveal too much plaintext information, which can be used by an attacker to spoil the game. In the section, we'll describe how to apply cryptographic commitments to fix this problem.

In our RPS contract the user is using a numeric scale as their input with 0: rock, 1: paper, 2: scissors. Let's take a look at the function that registers their inputs and think about possible vulnerabilities:

```
def input(choice):
        if self.storage["player1"] == msg.sender:
                self.storage["p1value"] = choice
                return(1)
        elif self.storage["player2"] ==  msg.sender:
                self.storage["p2value"] = choice
                return(2)
        else:
                return(0)
```

We can see that our `input()` function identifies the sender with `msg.sender` and then stores their input `choice` in plaintext (where `choice` = 0, 1, or 2). The lack of encryption means that the other player could see what their opponent played by looking at a block that published it; with that information they could input the winning choice to ensure they always win the prize pool. This can be fixed by using a commitment scheme. We will alter `input()` to accept a hash of [sender, choice, and a nonce]. After both players have committed

their inputs they will send their `choice` and `nonce` (as plaintext) to an `open()` function. `open()` will verify what they sent to *input*(). What they send to *open*() will be hashed, and that hash will be checked against the hash the user committed through `input()`. If the two hashes don't match then the player will automatically lose based on the assumption they were being dishonest. Understanding where crypto elements should be used is crucial to justifying why others should use your contract.

In order to enhance the security and fairness of our contract we will implement a commitment scheme using the hashing functions discussed earlier in this guide. The first change that is necessary in our contract is to have the *input*() function accept the hash given from the user. Our RPS application would prompt the participants in our game to send a hash of their input and a nonce of their choosing. Thus `choice` = SHA3(msg.sender's public address, numerical input (0 or 1 or 2) + `nonce`). This hashed value is stored in the contract, but there is no way for either opponent to discover the other's input based on their committed choice alone.

Now that we have the hash stored in the contract we need to implement an *open*() function that we discussed earlier. Our `open()` function will take the plaintext inputs and nonces from the players as parameters. We will hash these together with the unique sender ID and compare to the stored hash to verify that they claim to have committed as their input is true. Remember, up until this point the contract has *no way of knowing* who the winner is because it has *no way of knowing* what the inputs are. The contract doesn't know the nonce, so it cannot understand what the `choice` sent to `input()` was. Below is the updated, cleaned up contract (version2.py) implementing an *open*() and modifying *check*() to work with our new scheme. Notice we have added a method `open()` and reorganized our `check()`:

```
def input(player_commitment):
        if self.storage["player1"] == msg.sender:
                self.storage["p1commit"] = player_commitment
                return (1)
        elif self.storage["player2"] ==  msg.sender:
                self.storage["p2commit"] = player_commitment
                return(2)
        else:
                return(0)

def open(choice, nonce):
        if self.storage["player1"] == msg.sender:
                if sha3([msg.sender, choice, nonce], items=3) == self.storage["p1commit"]:
                        self.storage["p1value"] = choice
                        self.storage["p1reveal"] = 1
                        return(1)
```

```
                else:
                        return(0)
        elif self.storage["player2"] == msg.sender:
                if sha3([msg.sender, choice, nonce], items=3) == self.storage["p2commit"]:
                        self.storage["p2value"] = choice
                        self.storage["p2reveal"] = 1
                        return(2)
                else:
                        return(0)
        else:
                return(-1)

def check():
        #check to see if both players have revealed answer
        if self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                #If player 1 wins
                if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
                        send(100,self.storage["player1"], self.storage["WINNINGS"])
                        return(1)
                #If player 2 wins
                elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
                        send(100,self.storage["player2"], self.storage["WINNINGS"])
                        return(2)
                #If no one wins
                else:
                        send(100,self.storage["player1"], 1000)
                        send(100,self.storage["player2"], 1000)
                        return(0)
        #if p1 revealed but p2 did not, send money to p1
        elif self.storage["p1reveal"] == 1 and not self.storage["p2reveal"] == 1:
                send(100,self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        #if p2 revealed but p1 did not, send money to p2
        elif not self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                send(100,self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        #if neither p1 nor p2 revealed, keep both of their bets
        else:
                return(-1)
```

3.3 Incentive Compatability

Designing an effective smart contract often means considering the incentives of the players involved, and aligning these incentives with the desired behavior. Can a user profit by using the contract in an unexpected way? Is "honest" behavior more expensive than the alternative? We strive to make "incentive compatible" contracts, which roughly means that using the contract as intended is the most cost-effective behavior. In a typical escrow contract, a collateral deposit is collected from both individuals so they each have an incentive to complete their exchange. In a game contract where inputs are encrypted, a collateral deposit should be implemented to encourage both players to decrypt their responses within a time frame to avoid cheating or stalling the contract. Let's look and see how our RPS contract holds up with regard to incentives:

```
def check():
        #check to see if both players have revealed answer
        if self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                #If player 1 wins
                if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
                        send(100,self.storage["player1"], self.storage["WINNINGS"])
                        return(1)
                #If player 2 wins
                elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
                        send(100,self.storage["player2"], self.storage["WINNINGS"])
                        return(2)
                #If no one wins
                else:
                        send(100,self.storage["player1"], 1000)
                        send(100,self.storage["player2"], 1000)
                        return(0)
        #if p1 revealed but p2 did not, send money to p1
        elif self.storage["p1reveal"] == 1 and not self.storage["p2reveal"] == 1:
                send(100,self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        #if p2 revealed but p1 did not, send money to p2
        elif not self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                send(100,self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        #if neither p1 nor p2 revealed, keep both of their bets
        else:
                return(-1)
```

Given the version at the end of this section, our contract is *almost* incentive compatible. Only one party needs to call the `check()` function in order for the winnings to be fairly

distributed to the actual winner, regardless of who calls. This requires one player to spend gas to check to see who won, while the other player doesn't need to spend any gas. There is currently no way to require two people to spend equal amount of gas to call one function. How could this affect the incentives of the contract?

In the next section we will look at how the current block number and the amount of blocks that have arrived previously affect the security of a contract. We will look to alter our contract further so that if someone doesn't open (verify) their rock/paper/scissors commitments within a given timeframe (i.e. 5 blocks after they are added to the contract), then the contract would send the money to the person who *did* verify their input by the deadline. This incentivizes both users to verify their inputs before the `check()` function is called after a random amount of blocks have been published. If you don't reveal your commitment, then you are *guaranteed* to lose.

3.4 Original Buggy Rock, Paper, Scissor Contract

```
data winnings_table[3][3]

def init():
        #If 0, tie
        #If 1, player 1 wins
        #If 2, player 2 wins

        #0 = rock
        #1 = paper
        #2 = scissors

        self.winnings_table[0][0] = 0
        self.winnings_table[1][1] = 0
        self.winnings_table[2][2] = 0

        #Rock beats scissors
        self.winnings_table[0][2] = 1
        self.winnings_table[2][0] = 2

        #Scissors beats paper
        self.winnings_table[2][1] = 1
        self.winnings_table[1][2] = 2

        #Paper beats rock
        self.winnings_table[1][0] = 1
```

```
26          self.winnings_table[0][1] = 2
27
28          self.storage["MAX_PLAYERS"] = 2
29          self.storage["WINNINGS"] = 0
30
31  def add_player():
32          if not self.storage["player1"]:
33                  if msg.value == 1000:
34                          self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
35                          self.storage["player1"] = msg.sender
36                          return(1)
37                  return (0)
38          elif not self.storage["player2"]:
39                  if msg.value == 1000:
40                          self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
41                          self.storage["player2"] = msg.sender
42                          return(2)
43                  return (0)
44          else:
45                  return(0)
46
47  def input(choice):
48          if self.storage["player1"] == msg.sender:
49                  self.storage["p1value"] = choice
50                  return(1)
51          elif self.storage["player2"] ==  msg.sender:
52                  self.storage["p2value"] = choice
53                  return(2)
54          else:
55                  return(0)
56
57  def check():
58          #If player 1 wins
59          if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
60                  send(100,self.storage["player1"], self.storage["WINNINGS"])
61                  return(1)
62          #If player 2 wins
63          elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
64                  send(100,self.storage["player2"], self.storage["WINNINGS"])
65                  return(2)
66          #If no one wins
67          else:
```

```
68                 send(100,self.storage["player1"], self.storage["WINNINGS"]/2)
69                 send(100,self.storage["player2"], self.storage["WINNINGS"]/2)
70                 return(0)
71
72 def balance_check():
73         log(self.storage["player1"].balance)
74         log(self.storage["player2"].balance)
```

4 State Machine Transitions

Maintaining State in Smart Contracts In many scenarios, there is a need to adapt the behavior of a contract depending on the the messages it receives, or depending on how much time has passed since a certain event. In other words, several applications need a stateful contract that acts differently to similar messages, depending on its state.

Maintaining the notion of a state in a contract requires a mechanism to handle state transitions, which we classify into event-based and time-based. We present simple approaches for how to express these in Serpent.

Event-based state transitions. In this case, the state changes based on messages that the contract receives. One example is a puzzle contract that gives a reward to the first person who solves a problem, or a game contract that waits for two players to join before starting the game. The behavior of the contract should adapt when such events occur, as otherwise money may be needlessly lost. Such contracts can be straightforwardly implemented by maintaining state variables in the contract storage. The following example is a proof-of-work contract that gives an award to the first message sender who solves a Bitcoin-like proof-of-work puzzle.

```
1  def init(puzzle, target):
2       self.storage["isSolved"] = 0                ## State variable
3       self.storage["puzzle"] = puzzle
4       self.storage["target"] = target
5
6
7  def receiveSolution(solution):
8      if(self.storage["isSolved"] == 0 AND
9              SHA3([self.storage["puzzle"], solution],2) < self.storage["target"]):
10 send(msg.sender, 10000)              # Sending reward
11 self.storage["isSolved"] = 1    # Changing the state variable
```

Time-based state transitions. Employing event-based transitions may not be enough to capture all the possible scenarios in typical applications. Think of an auction that accepts

any number of bidders, but sets a specific deadline after which no new bids are accepted. The contract in this case should have a way to decide whether to accept bids or not based on the time of the transaction.

There are two simple ways to use refer to the current time in a contract: `block.timestamp` or `block.number`. For example, the following is a fragment of an auction contract that only accepts bids submitted before a deadline. The deadline is 100 blocks ahead from the contract creation time.

```
def init():
     self.storage["deadline"] = 100 + block.number
     # Think of other auction details

def receiveBid(bid):
     if(block.number <= self.storage["deadline"]):
                # accept bid
     else:
                # abort
```

(Exercise: Think how to complete the auction contract above. You can add other methods.).

Hybrid transitions. Sometimes, complex contracts will need to incorporate both kinds of state transitions. For example, consider a fundraising contract that either concludes immediately after a certain target amount of money is collected, or else after a month passes without reaching the target. Therefore, the contract must change its state if a month passes, or when the contract balance exceeds a threshold, whatever happens first.

(Exercise: Think how to write a fundraising contract as described above).

References

[1] KenK. Kenk's first contract tutorial. `http://forum.ethereum.org/discussion/1634/tutorial-1-your-first-contract`", 2014.