A Programmer's Guide to Ethereum and Serpent

Kevin Delmolino
del@terpmail.umd.edu

Mitchell Arnett
mitchell.arnett@gmail.com

March 8, 2015

Contents

1 Introduction

2 Installing Pyethereum and Serpent

> NOTE: This section is not required if the provided virtual machine is used. We have preinstalled all of the necessary applications to program Ethereum contracts using Pyethereum and Serpent. This section goes over installing a native copy of Pyethereum and Serpent on your machine and give a brief overview of what each component does.

This section assumes you are comfortable with the command line and have git installed. If you need assistance getting git installed on your local machine, please consult `http://git-scm.com/book/en/v2/Getting-Started-Installing-Git`.

First, lets install Pyethereum. This is the tool that allows for us to interact with the blockchain and test our contracts. We will be using Pyethereum, but there are also Ethereum implementations in C++ (cpp-ethereum) and Go (go-ethereum).

In order to install Pyethereum, we first need to download it. Go to a directory you don't mind files being downloaded into, and run the following command:

```
git clone https://github.com/ethereum/pyethereum
```

This command clones the code currently in the ethereum repository and copies it to your computer. Next, change into the newly downloaded pyethereum directory and execute the following command

```
git branch develop
```

This will change us into the develop branch. This code is usually relatively stable, and we found that it has better compatibility with the more modern versions of Serpent. Please note that later on, this step may not be necessary as the Ethereum codebase becomes more stable, but with the current rapid development of Ethereum, things are breaking constantly, so it pays to be on the cutting edge.

Finally, we need to install Pyethereum. Run the following command:

```
sudo python setup.py install
```

This actually installs Pyethereum on our computer. Note that if you are on a non-unix-like operating system, such as Windows, the sudo command, which executes the command with root privileges, may be different. We recommend running Ethereum on unix-like operating systems such as Mac OS X and Linux.

Now, we are going to install serpent. This allows for us to compile our serpent code into the stack-based language that is actually executed on the blockchain. The steps are extremely similar. Go to the directory that you downloaded the ethereum directory into and run the following commands:

```
git clone https://github.com/ethereum/serpent
cd serpent
git branch develop
sudo python setup.py install
```

Now that Pyethereum and Serpent are installed, we should test that they are working. Go to the pyethereum/tests directory and run the following command:

```
python pytest -m test_contracts.py
```

If the test states that it was successful, then everything is installed correctly and you are ready to continue with this guide!

3 Using Pyethereum Tester

In order to test our smart contacts, we will be using the Pyethereum Tester. This tool allows for us to test our smart contracts without interacting with the blockchain itself. If we were to test on a blockchain - even a private one - it would take a lot of time to mine enough blocks to get our contract in the chain and acquire enough ether to run it. It would waste a lot of time. Therefore, we use the tester.

Below is a simple contract that will be used as an example to show how to set up a contract. [2, 1]

```
import serpent
from pyethereum import tester, utils, abi

serpent_code='''
def main(a):
        return (a*2)
'''

evm_code = serpent.compile(serpent_code)
translator = abi.ContractTranslator(
        serpent.mk_full_signature(serpent_code))
data = translator.encode('main', [2])
s = tester.state()
```

```
c = s.evm(evm_code)
o = translator.decode('main', s.send(tester.k0, c, 0, data))

print(o)
```

Now what is this code actually doing? Let's break it down.

```
import serpent
from pyethereum import tester, utils, abi
```

This code imports all of the assets we need to run the tester. We need serpent to compile our contract, we need pyethereum tester to run the tests, we need ABI to encode and decode the transactions that are put on the blockchain, and we need utils for a few minor operations.

```
serpent_code='''
def main(a):
        return (a*2)
'''
```

This is our actual serpent code. We will discuss Serpent's syntax later in the guide, but this code will double the parameter.

```
evm_code = serpent.compile(serpent_code)
translator = abi.ContractTranslator(
        serpent.mk_full_signature(serpent_code))
```

Here, we finally get ready to run our actual code. The evm_code variable holds our compiled code. This is the byte code that we will actually run on the virtual machine. The translator variable holds the code that will allow for us to encode and decode the code that will be run on the blockchain.

```
data = translator.encode('main', [2])
s = tester.state()
```

The data variable holds our encoded variables. We are going to call the "main" function, and we are going to send one parameter to it, the number 2. We encode using the translator. Next, we are going to create a state (essentially a fake blockchain). This state is what we will run our contract on.

```
c = s.evm(evm_code)
o = translator.decode('main', s.send(tester.k0, c, 0, data))
```

The c variable holds our contract. The evm() function puts our contract onto our fake blockchain. Finally, we run a transaction. We use the send() function to execute the contract (whose address is stored in c). The entity sending the transaction is "tester.k0" who is a fake public key used for testing. We are sending no money to run the contract, so the third parameter is a zero. Finally, we send our encoded data.

```
o = translator.decode('main', s.send(tester.k0, c, 0, data))
print(o)
```

Finally here, we will use our translator to decode out what the function returned. We will print that using the standard python print() function.

The code can be executed using the command "python file_name.py". When executed, this code will output double the input parameter. So this code will output the number 4. [2, 1]

3.1 Testing Contracts with Multiple Parties

4 Language Reference

There are several different languages used to program smart contracts for Ethereum. If you are familiar with C or Java, Solidity is the most similar language. If you really like Lisp or functional languages, LLL is probably the most functional language. The Mutant language is most similar to C. We will be using Serpent 2.0 (we will just refer to this as Serpent, since Serpent 1.0 is deprecated) in this reference, which is designed to be very similar to Python. Even if you are not very familiar with Python, Serpent is very easy to pickup.

4.1 The log() Function

The log function allows for easy debugging. If X is defined as the variable you want output, log(X) will output the contents of the variable. We will use this function several times throughout this document.

4.2 Variables

Assigning variables in LaTex is very easy. Simply set the variable equal to whatever you would like the variable to equal. Here's a few examples:

a = 5
b = 10
c = 7
a = b

If we printed out the variables a, b and c, they would be 10, 10 and 7, respectively.

Special Variables Serpent creates several special variables that reference certain pieces of data or pieces of the blockchain that may be important for your code. We have reproduced the table from the official Serpent 2.0 wiki tutorial for your reference. [3]

Variable	Usage
tx.origin	Stores the address of the address the transaction was sent from.
tx.gasprice	Stores the cost in gas of the current transaction.
tx.gas	Stores the gas remaining in this transaction.
msg.sender	Stores the address of the person sending the information being processed to the contract
msg.value	Stores the amount of ether (measured in wei) that was sent with the message
self	The address of the current contract
self.balance	The current amount of ether that the contract controls
x.balance	Where x is any address. The amount of ether that address holds
block.coinbase	Stores the address of the miner
block.timestamp	Stores the timestamp of the current block
block.prevhash	Stores the hash of the previous block on the blockchain
block.difficulty	Stores the difficulty of the current block
block.number	Stores the numeric identifier of the current block
block.gaslimit	Stores the gas limit of the current block

4.3 Control Flow

In Serpent, we mostly will use if..elif..else statements to control our programs. For example:

```
if a == b:
        a = a + 5
        b = b - 5
        c = 0
        return(c)
elif a == c:
        c = 5
        return(c)
else:
        return(c)
```

Tabs are extremely important in Serpent. Anything that is inline with the tabbed section after the if statement will be run if that statement evaluates to true. Same with the elif and else statements. [3]

Important to also note is the not modifier. For example, in the following code:

```
if not (a == b):
        return(c)
```

The code in the if statement will not be run if a is equal to b. It will only run if they are different. The not modifier is very similar to the ! modifier in Java. [3]

4.4 Loops

Serpent supports while loops, which are used like so:

```
somenum = 10
while somenum > 1:
        log(somenum)
        somenum = somenum - 1
```

This code will log each number starting at 10, decrementing and outputting until it gets to 1. [4]

4.5 Arrays

Arrays are very simple in serpent. A simple example is below:

```
def main():
        arr1 = array(1024)
        arr1[0] = 10
        arr1[129] = 40
        return(arr1[129])
```

This code above simply creates an array of size 1024, assigns 10 to the zero address and assigns 40 to address 129. It then returns the value at address 129 in the array. [3, 4]

Functions that can be used with Arrays include:

- slice(*arr*, items=*s*, items=*e*) where *arr* is an array, *s* is the start address and *e* is the end address. This function splits out the portion of the array between s and e, where $s <= e$. That portion of the array is returned.
- len(*arr*) returns the length of the *arr* array.

Returning arrays is also possible. In order to return an array, append a ":arr" to the end of the array in the return statement. For example:

```
def main():
        arr1 = array(10)
        arr1[0] = 10
        arr1[5] = 40
        return(arr1:arr)
```

This will return an array where the values were initialized to zero and address 0 and 5 will be initialized to 10 and 40, respectively. [3]

4.6 Strings

Serpent uses two different types of strings, with each treated differently. The first is called short strings. These are treated like a number by Serpent and can be manipulated as such. Long strings are treated like an array by serpent, and treated as such. Long strings are very similar to strings in C, for example. As a contract programmer, we must make sure we know which variables are short strings and which variables are long strings, since we will need to treat these differently. [3]

Short Strings Short strings are very easy to work with since they are just treated as numbers. Let's declare a couple new short strings:

```
str1 = "string"
str2 = "string"
str3 = "string3"
```

Very simple to do. Comparing two short strings is also really easy:

```
return (str1 == str2)
return (str1 == str3)
```

The first return statement will output 1 which symbolizes true while the second statement will output 0 which symbolizes false. [3]

Long Strings Long strings are implemented similarly to how they are in C, where the strings is just an array of characters. There are several commands that are used to work with long strings:

- In order to define a new long string, do the following:

```
arbitrary_string = text("This is my string")
```

- If you would like to change a specific character of the string, do the following:

```
arbitrary_string = text("This is my string")
setch(arbitrary_string, 5, "Y")
```

  In the setch() function, we are changing the fifth index of the string to 'Y'.

- If you would like to have returned the ASCII value of a certain index of the string, do the following:

```
arbitrary_string = text("This is my string")
getch(arbitrary_string, 5)
```

  This will retrieve the ASCII value at the fifth index.

- All functions that work on Arrays will also work on long strings

[3, 4]

4.7 Functions

Functions work in Ethereum very similarly to how they work in other languages. You can probably infer how they are used from some of the previous examples. Here is an example with no parameters:

```
def main():
        #Some operations
        return(0)
```

And here is an example with three parameters.

```
def main(a,b,c):
        #Some operations
        return(0)
```

Special Function Blocks There are three different special function blocks. These are used to declare functions that will always execute before certain other functions.

First, there is init. The init function will be run once when the contract is created. It is good for declaring variables before they are used in other functions.

Next, there is shared. The shared function is executed before init and any other functions. Finally, there is the any function. The any function is executed before any other function except the init function. [3]

4.8 Persistant Data Structures

Persistant data structures can be declared using the "data" declaration. This allows for the declaration of arrays and tupples. For example, the following code will declare a two dimensional array:

```
data twoDimArray[][]
```

Very simple, the next example will declare an array of tupples. The tupples contain two items each - item1 and item2.

```
data arrayWithTupples[](item1, item2)
```

These variables will be persistant throughout the contract's execution.

Now, lets say I wanted to access the data in these structures. How would I do that? Its simple, the arrays use standard array syntax and tupples can be accessed like functions. Lets say, for example I wanted to access the "item1" value from the arrayWithTupples strucutre from the second array address, I would do that like so:

```
x = arrayWithTupples[2].item1
```

And that will put the value of the second index of the array in the item1 field into x. [3]

4.9 Hashing

Serpent allows for hashing using two different hash functions - SHA-256 and RIPEMD-160. The function takes the parameters a and s where a is the array of elements to be hashed and s is the size of the array to be hashed. For example, we are going to hash the array [4,5,5,11,1] using SHA-256 and return the value below. [3]

```
def main(a):
        bleh = array(5)
        bleh[0] = 4
        bleh[1] = 5
        bleh[2] = 5
        bleh[3] = 11
        bleh[4] = 1
        return(sha256(bleh, items=5))
```

The output is [92958224028375895182299457531563411438064489993925166733548623543505998847 01*L*]

The function definitions are:

- x = sha256(a, size=s) for SHA-256
- x = ripemd160(a, size=s) for RIPEMD-160

Please note that any inputs to the hash function can be seen by anyone looking at the block chain. Therefore, when keeping secrets between two parties, the hash values should be computed off of the blockchain then only the hash value put on the block chain. Then, when the values are verified, then compute the hash on the blockchain, and compare to the precomputed hash.

4.10 Random Number Generation

In order to do random number generation, you must use one of the previous blocks as a seed. Then, use modulus to ensure that it is a number within the range necessary. In the following examples, we will do just this.

In this example, we will the function will take a parameter a. It will generate a number between 0 and a (including zero).

```
def main(a):
        raw = block.prevhash
        if raw < 0:
                raw = 0 - raw
        return(raw%a)
```

Note that we must make sure that the raw number is positive. [5]

If we wanted the lowest number to be a number other than zero, we must add that number to the random number generated.

Now, when we are referencing previous blocks, we need to make sure there are blocks before our current block that we can reference. On the actual ethereum blockchain, this would not be a big deal since once we build one block on the genesis block, we will always have a previous block. When testing, however, we will need to create more blcoks. This will also give us more ether if our tester runs out of ether. The code to mine a block is below:

```
s.mine(n=1,coinbase=tester.a0)
```

where n refers to the number of blocks to be mined and coinbase refers to the tester address that will "do" the mining. [1]

5 Basic Serpent Contract Example

Before moving into more difficult examples, let's take a quick look at an Easy Bank example from KenK's first tutorial. 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 contracual currency and not Ether (which we will dicuss and implement in a later contract example). So creating the currency is done within our contract. Now that we know what our bank does (create and send 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():
        self.storage[msg.sender] = 10000
        self.storage["Taylor"] = 0
def send_currency_to(value):
        to = "Taylor"
        from = msg.sender
        amount = value
        if self.storage[from] >= amount:
                self.storage[from] = self.storage[from]  - amount
                self.storage[to] = self.storage[to] + amount
```

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():
        self.storage[msg.sender] = 10000
        self.storage["Taylor"] = 0
```

The init function in serpent is very similar to init in python, which is very similar to common constructors in Java. The init function runs once and only once at contract creation. In our contract the init block runs and instantiates two objects in contract storage.

Our init method, from a general perspective, intializes one account with a balance of 10,000U (this will be how we donote our contract-exclusive currency) and another account with a balance of 0U. 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 10,000U at key msg.sender (we'll identify what this is in a moment) and at the key "Taylor" there is a value of 0U.

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. We now have a heightened understanding of init, lets look at our send method.

```
def send_currency_to(value):
        to = "Taylor"
        from = msg.sender
        amount = value
        if self.storage[from] >= amount:
                self.storage[from] = self.storage[from] - amount
                self.storage[to] = self.storage[to] + amount
```

Let's take a look at this one piece at a time. The first three lines aim are setting up variables that we will use in the last three lines. The first line is establishing who we are sending our funds to - and just as we setup in init, we are sending our funds to our friend Taylor. from is being set to the address that the funds are from, which is us - msg.sender. Finally, the value variable is set to the parameter passed to our send_currency_to function. When the contract was invoked a value needed to be sent in order for it to run properly, this value is the value that is going to be sent to Taylor.

Okay, now that we understand what variables we are working with let's dive into the last portion of our contract. We want to check that the balance for the bank account in the contract's storage at from (remember that you are who this is from!) is greater than or equal to the amount we are attempting to send - obviously we do not want our contract sending money that the sender does not have.

If the account balance passes our check we subtract the amount being sent from the sender balance:self.storage[from] = self.storage[from] - value. We then add to the balance of the account receiving the currency:self.storage[to] = self.storage[to] + value.

Great! We have officially worked our way through a very basic contract example! Now our friend Taylor has 1000U! 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?

6 Moderate Serpent Contract Example

So we've made it through the first serpent example, which we now have realized wasn't as daunting as it first seemed. We understand that every contract has its own contractual storage that is accessed through self.storage[key] = value. We understand that we can use parameters passed to function. Lastly we understand that msg.sender gives us the unique identifier of whoever sent the message, and that all participants involved with a contract have their own unique identifier that can be used in whatever creative way you like.

Let's look at a more moderate contract that keeps with our bank theme. So, just like with our first contract, we need to classify what we are making and what characteristics the contract will need to leverage our desired features. We are going to be implementing what is known as a mutual credit system. A generalized idea of a mutual credit system is the intersection of a barter system and a non-regulated currency model. So, let's define a community that implements a mutual credit system and every participant gets a 1000 Unit credit (in this case 1UC = 1USD). In the beginning there is no money at all. It only comes into circulation when one of the participants uses his credit to pay another participant. If he uses his 1000 Units his balance is -1000 U. His supplier's balance is now +1000U. The total amount in circulation is now also 1000 U. This means there is always exactly as much in circulation as there is outstanding credit: a zero sum game.

One can clearly notice that this system creates money at the time of the transaction. At time 0, before any of the community's participants completed a transaction, the currency in circulation was zero. It is also clear that, unlike fiat currencies, this model does not require any centralized money supply management which, when discussing decentralized apps built on blockchain technology, is an attractive idea to implement. Regardless of your opinion on such a system, let's automate a contract to initiate these transactions and act as a public ledger to keep track of the community's participants and their account balances.

```
def init():
        contract.storage[((msg.sender * 0x10) + 0x1)] = 0x1
        contract.storage[((msg.sender * 0x10) + 0x2)] = 0x1

def code():
        toAsset = (msg.data[0] * 0x10) + 0x1
        toDebt = (msg.data[0] * 0x10) + 0x2
        fromAsset = (msg.sender * 0x10) + 0x1
        fromDebt = (msg.sender * 0x10) + 0x2
```

```
        value = msg.data[1]

        if contract.storage[fromAsset] >= value:
                contract.storage[fromAsset] = contract.storage[fromAsset] - v
        else:
                contract.storage[fromDebt] = value - contract.storage[fromAss
                contract.storage[fromAsset] = 0

        if contract.storage[toDebt] >= value:
                contract.storage[toDebt] = contract.storage[toDebt] - value
        else:
                value = value - contract.storage[toDebt]
                contract.storage[toAsset] = contract.storage[toAsset] + value
                contract.storage[toDebt] = 0
```

7 An Advanced Contract Example

Now that we have gone through and annotated several contract examples it is time to consider a couple key design concepts required to create a high-level smart contract. By the end of this section we will talk about several key mistakes that show up in high-level contracts, and you will aim to identify and resolve them in a rock, paper, scissor contract example (RPS).

7.1 Contract Theft

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 in a similar light to Bitcoin; Ether is mined and 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 below:

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

```
            self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.val
            return(2)
    else:
            return(0)
```

In this section, a user adds themselves to the game by sending a small amount of Ether with the 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 very probable scenarios 1) a potential entrant sends too much Ether or too little Ether or, 2) the entrant is not allowed in the game for any reason. In both of the following scenarios the contract will keep their money. The first scenario allows an unfair entrance fee, where one user could bet below the expected amount and the other user pays the expected amount - both participants should be betting the same amount! The second scenario results in our contract taking Ether from those who aren't necessarily the two locked-in participants. Both of these errors will cause distrust in our contract, eventually resulting in the community not trusting this contract and its author - you.

It seems like our contract could use some case checks on when to issue refunds - 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?

7.2 Failing to use Cryptography

It goes without saying that as a student in a computer security course you would implement cryptographic practices wherever you can. Thus with any contract that requires any user inputs that affect the outcome of said contract, crypto should be implemented. 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.

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

We can see that our do() function identifies the sender with msg.sender and then stores their input 'a' in plaintext (where a = 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 their choice to ensure they always win. This can be fixed by hashing (with a nonce) the value to send to the contract. Then, when both parties have decided what to play, they send their answer and a nonce to show what they played. Understanding where crypto elements should be used is crucial to justifying your contract.

In order to enhance the security and fairness of our contract we will use the hashing functions discusses here[INSERT LINK TO HASING SECTION HERE]. The first change that is necessary in our contract is to have the do() 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 a = SHA256(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.

Now that we have the hash stored in the contract we need to look at the check() function:

```
def check():
        if

        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        else:
                return(-1)
```

In its current state this function will no longer work because 'p1value' and 'p2value' no longer have the cleartext input, they have a randomized hash. Thus, we need will modify the check() so the user will send their plaintext input and their nonce as parameters so check() can verify that what they gave matches the hash that is stored in the contract. 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 input a send to do() was. Now that we know that check() was verify the answer, below is the updated, cleaned up contract:

```
def do(player_hash):
        if self.storage["player1"] == msg.sender:
                self.storage["p1hash"] = player_hash
                return(1)
        elif self.storage["player2"] == msg.sender:
                self.storage["p2hash"] = player_hash
```

```
                return(2)
        else:
                return(0)

def verify(a, nonce):
        if self.storage["player1"] == msg.sender:
                player_one_hash = array(1)
                player_one_hash[0] = a + nonce
                self.storage["p1value"] = a
        elif self.storage["player2"] == msg.sender:
                player_two_hash = array(1)
                player_two_hash[0] = a + nonce
                self.storage["p2value"] = a
        else:
                return(-1)

def check():
        if self.storage["p1value"] and self.storage["p2value"]:
                if self.storage["p1value"] == self.storage["p2value"]:
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                return(0)
                elif self.storage["p1value"] > self.storage["p2value"] and se
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else
                                pay_out_to("player1")
                                return(1)
                elif self.storage["p1value"] > self.storage["p2value"] and se
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player2")
                                return(2)
                elif self.storage["p2value"] > self.storage["p1value"] and se
```

```
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player2")
                                return(2)
                elif self.storage["p2value"] > self.storage["p1value"] and se
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player1")
                                return(1)
        else:
                return(-1)

def pay_out_to(player):
        send(self.storage[player], self.storage["WINNINGS"])
```

7.3 Incentive incompatability

The final key bug to watch out for is incentive incompatibility. There are contract ideas that must consider user incentives in order for them to run as planned. If I had an escrow contract incentives must be implemented so both individuals don't always so they did not receive their promised service. If I have a game contract where inputs are encrypted, incentives must be implemented to ensure both players decrypt their responses simultaneously to avoid cheating. Let's look and see how our RPS contract holds up with regard to incentives:

```
def check():
        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
```

```
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        else:
                return(-1)
```

Given the version at the end of this section our contract is 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. If we implemented the crypto elements from part 2 of this section we would need to wait for a certain number of blocks to be mined and published before checking the results. If someone didnt bother to decrypt their rock/paper/scissors within that timeframe, the contract would, by default, send the money to the person who *did* decrypt their input. This incentivizes both users to decrypt their inputs before the check() function is called after a random amount of blocks have been published; if you don't decrypt you are *always* guaranteed not to win.

7.4 Rock, Paper, Scissor Contract

```
def init():
        self.storage["MAX_PLAYERS"] = 2
        self.storage["WINNINGS"] = 0

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

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

```
def check():
        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.stora
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        else:
                return(-1)

def balance_check():
        log(self.storage["player1"].balance)
        log(self.storage["player2"].balance)
```

8 Resource Overview

This guide is provided as a "one stop shop" for a quick way to learn how to program smart contracts with ethereum. However, the platform is always changing and it would be impossible for this guide to cover everything. We have provided some links below that provide some additional insight into programming ethereum contracts. Many of these sources were actually used in creating this guide.

- Ethereum Wiki - `https://github.com/ethereum/wiki/wiki` - This source has some fantastic tutorials and reference documentation about the underlying systems that power Ethereum. This should be your first stop when you have problems with Ethereum.
- Serpent Tutorial - `https://github.com/ethereum/wiki/wiki/Serpent` - This is the official serpent tutorial that is on the Ethereum Wiki. It gives a good, brief overview of many of the most used components of serpent and goes over basic testing.
- KenK's Tutorials - Most of these tutorials use old versions of Serpent, but should be updated soon. These give a great overview of some of Ethereum's more advanced features. Note that these tutorials use cpp-ethereum and not pyethereum.

- Part 1: http://forum.ethereum.org/discussion/1634/tutorial-1-your-first-contract
- Part 2: http://forum.ethereum.org/discussion/1635/tutorial-2-rainbow-coin
- Part 3: http://forum.ethereum.org/discussion/1636/tutorial-3-introduction-to-the-javascript-api

References

[1] Using pyethereum.tester. Pyethereum Github. 2014. https://github.com/ethereum/pyethereum/wiki/Using-pyethereum.tester

[2] pyethereum/tests/test_contracts.py. Pyethereum Github. 2015. https://github.com/ethereum/pyethereum/blob/develop/tests/test_contracts.py

[3] Serpent. Ethereum Wiki. 2015. https://github.com/ethereum/wiki/wiki/Serpent

[4] Serpent 1.0 (old). Ethereum Wiki. 2015. https://github.com/ethereum/wiki/wiki/Serpent-1.0-(old)

[5] PeterBorah. ethereum-powerball. 2014. https://github.com/PeterBorah/ethereum-powerball/tree/master/contracts

[6] Shi, E. Undergraduate Ethereum Lab at Maryland and Insights Gained. 2015. https://docs.google.com/presentation/d/1esw_lizWG06zrWaOQKcbwrySM4K9KzmRD3rtBUx0zEw/edit?usp=sharing

[7] Buterin, V. 2014. https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf