A Programmer's Guide to Ethereum and Serpent

Kevin Delmolino
del@terpmail.umd.edu

Mitchell Arnett
marnett@umd.edu

April 1, 2015

Contents

1 Introduction

The goal of this document is to teach you everything you need to know about Ethereum in order to start developing your own Ethereum contracts and decentralized apps. So, what is Ethereum? Ethereum can be seen as a decentralized platform that uses the network unit Ether as the fuel to power all contracts on the network. Ethereum is more than a cryptocurrency (even though mining is involved), it is a network that enables and powers Ethereum contracts. So what is an Ethereum contract? Think of it as a program that aims to provide decentralized services including: voting systems, domain name registries, financial exchanges, crowdfunding platforms, company governance, self-enforcing contracts and agreements, intellectual property, smart property, and distributed autonomous organizations. Ethereum is the ubiquitous bitcoin. It uses a similar underlying blockchain technology as bitcoin while broadening the scope of what it is capable of accomplishing.

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 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 ethereum 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 return a value that is double the parameter a. Please note that this is the only non-python code in this section.

```
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" using ethereum. 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.k*0 who is a fake public key used for testing. We are sending no ether into 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

Let's say we want to write a smart contract between two people, we need to make sure they are able to identify as themselves. In ethereum, every user has one (or probably more) addresses that they are associated with. Due to the nature of public key encryption, the public key is what identifies them on the Ethereum network, and the private key is what they use to sign and authorize transactions. When testing a contract, we don't want to go through the hassle of making all of these addresses, so we can use tester addresses. We used one of these in the previous section (*tester.k*0). However, there is of course more than just one tester address. We can also use *tester.k*1, *tester.k*2 and so on, all the way up to *tester.k*9. Therefore, it is possible to test a contract with up to 9 parties using this method.

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. Note that all code after this point is Serpent, not Python. In order to test it, it must be put in the $serpent_code$ variable mentioned previously. Another thing to note is that many, if not all, of

the built-in fuctions you may come accross in other documentation for Serpent 1.0 will work in 2.0.

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. Here is an example of it in use:

```
def main(a):
    log(a)
    return(a)
```

This code will output the variable stored in a. Since we passed in a three, it should be a three. Below is the output of the log function:

```
('LOG', 'c305c901078781c232a2a521c2af7980f8385ee9', [3L], [])
```

The part that is important to us is the third piece of data stored in the tupple, specifically, the $[3L]$. This tells us that the value in the variable is a three. Unfortunately, the rest of this function is not well documented currently.

4.2 Variables

Assigning variables in Serpent 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, we would see 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 (and reworded portions) for your reference below. [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

Wei is the smallest unit of ether (the currency used in ethereum). Any time ether is referenced in a contract, it is in terms of wei.

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. This will also apply to functions and loops when we define those later on. [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 and most other languages. [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-th index and assigns 40 to index 129. It then returns the value at index 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.[3] In order to return an array, append : 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. 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 are 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 string 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 *arbitrary_string* to $'Y'$.

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

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

  This will retrieve the ASCII value at the fifth index in *arbitrary_string*.

- 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)
```

Defining functions is very simple and makes code a lot easier to read and write [3]. But how do we call these functions from within a contract? We must call them using $self.function\_name(params)$. Any time we reference a function within the contract, we must call it from self (a reference to the current contract). Note that any function can be called directly by a user. For example, lets say we have a function A and a function B. If B has the logic that sends ether and A just does the check, and A calls B to send the ether, an aversary could simply call funcion B and get the ether without ever going through the checks. We can fix this by not putting that type of logic in seperate functions.

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 Sending Wei

Contracts not only can have ether (currency) sent to them (via *msg.value*), but they can also send ether themselves. *msg.value* holds the amount of wei that was sent with the contract.

In order to send wei to another user, we use the send function. For example, lets say I wanted to send 50 wei to the user's address stored in x, I would use the code below.

```
send(x, 50)
```

This would then send 50 wei from this contract's pool of ether (the ether that other users/contracts have sent to it), to the address stored in x.

How do we get a user's address? The easiest way is to store it when that user sends a command to the contract. The user's address will be stored in $msg.sender$. If we save that address in persistent storage, we can access it later when needed [3] (we will go over persistent storage in the next section).

4.9 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 persistent throughout the contract's execution (In any command/-function called by any user to the same contract instance). Please note that data should not be declared inside of a function, rather should be at the top of the contract before any function definitions.

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 using a period and then the name of the value we want to access. 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 = self.arrayWithTupples[2].item1
```

And that will put the $item1$ value stored in the $self.arrayWithTupples$ array into x. [3] Note that we will need the self declaration so the contract knows we are referencing the arrayWithTupples structure in this contract.

Self.storage[] Ethereum also supplies a persistent key-value store called $self.storage[]$. This is mostly used in older contracts and also is used in our example below for simplicity. Essentially, put the key in the brackets and set it equal to the value you want. An example is below when I set the value y to the key x.

```
self.storage["x"] = "y"
```

Now whenever $self.storage["x"]$ is called, it will return y. For simple storage, $self.storage[]$ is useful, but for larger contracts, we reccomend the use of data (unless you need a key value storage, of course). [3, 4]

4.10 Hashing

Serpent allows for hashing using three different hash functions - SHA3, 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:

- $x = sha3(a, size = s)$ for SHA3
- $x = sha256(a, size = s)$ 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. When we want to decode the secret in the hash, we should then send the nonce and the text to the blockchain, rehash it, and compare them with the prestored hash value. There is more detail about this process in the section "Failing to Use Cryptography".

4.11 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 the random number is in the necessary range. 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 blocks. 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. Note that this is python code, and the s variable references the current state of the "blockchain". You can not mine from inside of a Serpent contract. This function must be used after we have create the state [1]

4.12 The Callstack

The maximum callstack in Ethereum is of size 1024. An attacker could call a contract with an already existing callstack. If a send function (or any function) is called while already at the maximum callstack size, it will create the exception, but the execution of the contract will continue. Therefore, they could cause certain portions of the contract to be skipped. To solve this, put the following code at the beginning of your functions to ensure that an attacker can not try to skip portions of the contact:

```
if self.test_callstack() != 1: return(-1)
```

Then create the function $test_callstack()$:

```
def test_callstack(): return(1)
```

This will add a function to the callstack. If an attacker tries to break the callstack by 1, it will cause the contract to not execute.

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 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 is invoked a parameter called *value* needs to be provided in order for it to run properly. This parameter 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]

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 + 10] = 1000
    contract.storage[msg.sender + 20] = 1000

def code(num, value):
    toAsset = (num * 10) + 10
    toDebt = (num * 10) + 20
    fromAsset = (msg.sender * 10) + 10
    fromDebt = (msg.sender * 10) + 20

    if contract.storage[fromAsset] >= value:
        contract.storage[fromAsset] =
            contract.storage[fromAsset] - value
    else:
        contract.storage[fromDebt] =
            value - contract.storage[fromAsset]
        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
```

Always keep in mind what a given contract is seeking to accomplish, and ensure you are on the right path during each step of development.

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 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 refund - 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 previous section "Sending Wei" for more information.

7.2 Implementing Cryptography

It goes without saying that as a student in a computer security course you would implement cryptographic practices wherever you can. Thus given a contract that requires impactful user inputs (ones that affect the outcome of said contract) cryptography 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 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)
```

7.3 Incentive Compatability

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 within a time frame to avoid cheating. 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 the same amount.

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 passed affect the security of a contract. We will look to alter our contract further so if someone doesn't open (verify) their rock/paper/scissors within a given timeframe (i.e. 5 blocks after they are added to the contract), the contract would, by default, 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 verify you are *guaranteed* to lose.

7.4 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
    self.winnings_table[0][1] = 2

    self.storage["MAX_PLAYERS"] = 2
    self.storage["WINNINGS"] = 0
```

```
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)

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)

def check():
    #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"],
                 self.storage["WINNINGS"]/2)
        send(100, self.storage["player2"],
                 self.storage["WINNINGS"]/2)
        return(0)

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

Implement the changes from each of the aboved sections to have a much stronger contract!

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. All 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] KenK. Dec. 2014. http://forum.ethereum.org/discussion/1634/tutorial-1-your-first-contract

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

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