A Programmer's Guide to Ethereum and Serpent

May 12, 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 is a decentralized cryptocurrency that uses the its built-in currency, Ether, as "fuel" to power the programmable "smart contracts" that live on its blockchain. Ethereum is more than a cryptocurrency (even though mining is involved). Think of a "contract" as a program that provides services such as: 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 can also be thought of as an expanded version of Bitcoin. It uses a similar underlying blockchain technology, while broadening the scope of what it can do. [1]

2 Ethereum Tools

2.1 Acquiring the Virtual Machine

We have made a virtual machine that contains all of the necessary software. The virtual machine is running Ubuntu 14.04 LTS, Pyethereum and Serpent 2.0. Pyethereum is the program that allows for us to interact with the blockchain and test our contracts. We will be using Pyethereum, a Python based ethereum client, but there are also Ethereum implementations in C++ (cpp-ethereum) and Go (go-ethereum). Serpent 2.0 will allow for us to compile our serpent code into the stack-based language that is actually executed on the blockchain.

The virtual machine requires the host to be a 64-bit operating system, and for optimal performance, hardware acceleration should be turned on (VT-d/AMD-V). Normally, this is turned on by default when supported by your processor. Due to the advanced graphics used in the Ubuntu desktop environment, we recommend turning on 3D acceleration. For more information, refer to your virtual machine's documentation.

The Virtual Machine has been tested using VMWare Fusion (`https://www.vmware.com/products/fusion`) and VirtualBox (`https://www.virtualbox.org/`), however, it should work with any VM software that supports VMDK files. The Virtual Machine is available from `https://drive.google.com/file/d/0BzlG8wGYwTrGWlp0LWctYVIxRVU/view?usp=sharing`. The username is "user" and the password is "dees".

2.2 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. 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:

```
python setup.py install --user
```

This actually installs Pyethereum on our computer. Note that commands may be different if you are on a non-Unix-like platform. We recommend running Ethereum on Unix-like operating systems such as Mac OS X and Linux.

Now, we are going to install serpent. 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
python setup.py install --user
```

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 The Smart-Contract Programming Model

The Underlying Cryptocurrency. We shall make some simplifying assumptions about the security model of the underlying cryptocurrency. Loosely speaking, we assume that the cryptocurrency has a secure and incentive compatible consensus protocol.

The underlying cryptocurrency is based around a blockchain, which allows users to post messages and transfer units of a built-in currency. The data in the blockchain is guaranteed to be "valid" according to the predefined rules of the system (e.g., there are no double-spends or invalid signatures). All of the data in the blockchain is public, and every user can access a copy of it. No one can be *prevented* from submitting transactions and getting them included in the blockchain (with at most some small delay). There is global agreement about the contents of the blockchain history, except for the most recent handful of blocks (if there are "forks" at all, then longer forks are exponentially more unlikely).

We also assume that the built-in currency (ether, in this case) has a consistent monetary value. Users have an incentive to gain more of (or avoid losing) this currency. Anyone with can acquire ether by purchasing it or trading for it. The currency is assumed to be fungible; one unit of ether is exactly as valuable as any other, regardless of the currency's "history".

In reality, existing decentralized cryptocurrencies achieves only heuristic security. But we will make these assumptions nevertheless. How to design a provably secure decentralized consensus protocol under rationality assumptions is a topic of future research.

Contracts and Addresses. The system keeps track of "ownership" of the currency by associating each unit of currency to an "address". There are two kinds of addresses: one for users, and one for contracts. A user address is a hash of a public key; whoever knows the corresponding private key can spend the money associated to that address. Users can create as many accounts as they want, and the accounts need not be linked to their real identity.

A contract is an instance of a computer program that runs on the blockchain. It consists of program code, a storage file, and an account balance. Any user can create a contract by posting a transaction to the blockchain. The program code of a contract is fixed when the contract is created, and cannot be changed. The contract's code is executed whenever it receives a message, either from a user or from another contract. While executing its code, the contract may read from or write to its storage file. A contract can also receive money into its account balance, and send money from its account balance to other contracts or users.

The code of a contract determines how it behaves when it receives messages, under what conditions (and to whom!) it sends money out, and how it interacts with other contracts by sending messages to them. This document is especially about how to write code for useful and dependable contracts.

Transactions, Messages and Gas. A transaction always begins with a message from a user to some recipient address (either another user or a contract). This message must be signed by the user, and can contain data, ether, or both. If the recipient is a contract,

then the code of that contract is executed. If that code contains an instruction to send a message to another contract, then that contract's code is executed next. So, a transaction must contain at least one message, but can trigger several messages before it completes.

Messages act a bit like function calls in ordinary programming languages. After a contract finishes processing a message it receives, it can pass a return value back to the sender.

In some cases, a contract can encounter an "exception" (e.g., because of an invalid instruction). After an exception, control is also returned to the sender along with a special return code. The state of *all* contract, including account balances and storage, is reverted to what it was just prior to calling the exception-causing message.

Ethereum uses the concept of "gas" to discourage overconsumption of resources. The user who creates a transaction must spend some of the ether from that account to purchase gas. During the execution of a transaction, every program instruction consumes some amount of gas. If the gas runs out before the transaction reaches an ordinary stopping point, it is treated as an exception: the state is reverted as though the transaction had no effect, but the ether used to purchase the gas is not refunded! When one contract sends a message to another, the sender can offer only a *portion* of its available gas to the recipient. If the recipient runs out of gas, control returns to the sender, who can use its remaining gas to handle the exception and tidy up.

4 Simulating Contracts with 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 real blockchain - even a private one - it would take a lot of time to mine enough blocks to get our contract published on the blockchain and to run commands on it. Therefore, we use the tester.

Below is a simple example that we will use to show how to set up a contract. [5, 9]

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

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

s = tester.state()
c = s.abi_contract(serpent_code)

o = c.multiply(5)
print(str(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. From `pyethereum`, we need `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 (such as generating public addresses).

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

This is our actual serpent code. We will discuss Serpent's syntax later in the guide, but this code contains one function, named `multiply()`. This function will return a value that is double the parameter `a`. Please note that the code between the triple quotes is the only non-python code in this section.

```
s = tester.state()
c = s.abi_contract(serpent_code)
```

Here, we set up the tester. The first line sets up the initial state of the tester - a genesis block. This is the initial block of any block chain. Since we are testing on an independent chain, we will need to start it.

The second line calls the `abi_contract()` function. This will compile the code within our `serpent_code` variable and adds the contract to the block chain. At this point, we can now call `multiply()` function that we wrote.

```
o = c.multiply(5)
print(str(o))
```

Now, we can call any function that we wrote in the contract. When we call a function in the contract, the function call returns exactly what would be returned in serpent, just in python. We store what is returned in variable `o`. In this case, we simply printed out what was returned, though we can process it anyway we choose. In our example, the contract function returns 10, as expected.

In this case, the person sending the command to the contract is not defined. This isn't a problem in this contract, since the data returned is independent of the sender, it is irrelevant. However, what if we had a contract, as we will later, where a function is dependent on the sender? Simple, add a parameter that sends the private key of the sender's address. below is an example:

```
o = multiply(5, sender=tester.k0)
```

In this example, we send the data to the same contract, but the sender is defined. If we called `msg.sender` in our contract, it would return the public key of the sender. Note that `tester.k0` represents the private key (we'll go into this more in the next section). This is a unique identifier for the testing user. Pyethereum provides 10 of these (`tester.k0`-`tester.k9`), each an individual "user".

What if we wanted to send ether? No problem!

```
o = multiply(5, value=1000, sender=tester.k0)
```

All, we need to do, as seen in the example, is add a value parameter. We send the value in units of wei.

[5, 9]

4.1 Public and Private Keys

All cryptocurrencies are based on some form of public key encryption. What does this mean? It means that messages can be encrypted with one key (the private key) and decrypted with the public key. The Pyethereum tester provides us with fake addresses we can use for testing (`tester.k0 - tester.k9`), each of them representing an individual party in the contract. However, these are private addresses that we are using to sign transactions. This tells the world that the sender authorized this transaction to exist. Others can confirm this by using our public key.

Now, lets say we want someone to be able to submit public keys to a contract as a parameter. How do we calculate the public keys from the private tester keys we have? There is a function in pyethereum's utils that allows for us to do this:

```
public_k1 = utils.privtoaddr(tester.k1)
data = c.transfer(500,public_k1,sender=tester.k0)
```

We don't want to send our private key to a contract, because then others could sign transactions as us and take all of our ether! The code above uses the `utils.privtoaddr(private_key)` function, which returns the public key associated with `private_key`. We can then send the public key with the transaction, as we do in line two.

5 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 may be the language for you. 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 functions you may come across in other documentation for Serpent 1.0 will work in 2.0.

5.1 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 tuple, specifically, the `[3L]`. This tells us that the value in the variable is a three.

5.2 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. [8]

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. There are several other denominations as seen in the table below [2]:

Denomination	Amount (in ether)
wei	1.0×10^{18}
Kwei	1.0×10^{15}
Mwei	1.0×10^{12}
Gwei	1.0×10^{9}
Szabo	1.0×10^{6}
Finney	1000
Ether	1
Kether	.001
Mether	1.0×10^{-6}
Gether	1.0×10^{-9}
Tether	1.0×10^{-12}

A very easy to use converter is available at `http://ether.fund/tool/converter`.

5.3 Control Flow

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

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

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. [8]

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. [8]

5.4 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. [6]

5.5 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 [8, 6].

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 [8]. 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 [8].

5.6 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. [8]

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`. [8]

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.

[8, 6]

To check for the equality of two strings, it gets a little more difficult, and requires the `getch()` method. An example is given below that returns -1 if `str1` and `str2` are not equal, and 1 if they are.

```
def compare_equals():
        str1 = text("String 1")
        str2 = text("String 1")
        i = 0
        while i < len(str1):
                if getch(str1,i) != getch(str2,i):
                        return(-1)
                i = i + 1
        return(1)
```

5.7 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 [8]. 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 checks if the ether should be sent, and A calls B to send the ether, an adversary could simply call function B and get the ether without ever going through the check. We can fix this by not putting that type of logic in separate 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. This function is good for if we wanted a constant. Then, the constant would be declared before every other function's execution, so the constant would always exist.

Finally, there is the `any` function. The `any` function is executed before any other function except the `init` function [8].

5.8 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 [8] (we will go over persistent storage in the next section).

One thing to note is that the send function will send all of the remaining gas in the contract to the destination address, minus 25. if we want to define how much gas to send, we specify it as the first parameter. If we wanted to send only 100 gas, we would send the following:

```
send(100,x, 50)
```

5.9 Persistent Data Structures

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

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

The next example will declare an array of tuples. The tuples contain two items each - `item1` and `item2`.

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

These variables will be persistent throughout the contract's execution (In any 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. Example:

```
data arrayWithTuples[](item1, item2)
def someFunction1(params):
        ....
def someFunction2(params):
        ....
```

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 tuples 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 `arrayWithTuples` structure from the second array address, I would do that like so:

```
x = self.arrayWithTuples[2].item1
```

And that will put the `item1` value stored in the `self.arrayWithTuples` array into `x`. [8] Note that we will need the self declaration so the contract knows we are referencing the arrayWithTuples 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 examples 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 recommend the use of data (unless you need a key-value storage, of course). [8, 6] In this guide, we will use `self.storage[]`, but our "How to Program a Safe Smart Contract" guide's example is much more complex and uses data.

5.10 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. [8]

```
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 pre-stored hash value. There is more detail about this process in the accompanying "How To Program A Safe Smart Contract" guide.

5.11 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. [4]

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 [9]

5.12 Gas

As we know, Ethereum smart contracts are essentially small programs. As any programmer knows, infinite loops and inefficient code can cause problems. The ethereum network is not extremely powerful, as it is only designed to execute small programs. To incentivize efficient

programming, the execution of contracts requires gas. An amount of gas is "burned" for every operation that occurs in the transaction. Since a contract must be funded, this eliminates the ability for an infinite loop to occur.

When using the tester, we can simply send an arbitrary amount of gas (that is above the amount the contract needs to execute) since it is free. However, when executing contracts on an actual block chain, we need to make sure that we only spend what we need to. The best way to do this in pyethereum.tester is to use the variable `s.block.gas_used` where `s` is the current state. This stores the gas used thus far in the current block. Since this is the tester, and we are the only ones putting transactions into the block, this only counts the gas used by our transactions. Let's look at an example:

```
s = tester.state()
print(s.block.gas_used)  #Call 1 = 0 gas
c = s.abi_contract(serpent_code)

print(s.block.gas_used) #Call 2 = 3016 gas

o = c.deposit(value=1000, sender=tester.k0)
print(o)

print(s.block.gas_used) #Call 3 = 3879 gas
```

In the example above, we print the amount of gas used three times. At the first call, we have not added any transactions to the block chain, so we have not used any gas yet. At the second call, we have added our contract to the block chain, so we have used 3016 gas. Let's say we wanted to know how much gas the deposit command used. We can subtract the amount of gas used at call 3 from the amount of gas used at call 2 ($3879 - 3016 = 863$) to show that calling deposit with the given parameters costs 863 gas.

Now that we know the quantity of gas needed to execute the transaction, we need to figure out how much that will cost. Currently, on the public block chain, gas cost 10 szabo per unit. However, that unit will adjust when ethereum is officially released. The total price of the contract will be equal to the gas price multiplied by the gas cost of the transaction.

Note that when a transaction runs out of gas, the execution of the transaction simply rolls back - it's like it never happened. However, gas and any value sent to the contract or miner will not be refunded. [7, 3]

5.13 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 or contract, 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 [8].

One thing to note is that the send function will send all of the remaining gas in the contract to the destination address, minus 25. if we want to define how much gas to send, we specify it as the first parameter. If we wanted to send only 100 gas, we would send the following:

```
send(100,x, 50)
```

Note that if we are not sending it to another contract, but rather just sending it to a user, we should not send any gas, since the user is not a contract, and therefore does not need gas to complete the transaction.

5.14 The Call Stack

The maximum call stack in Ethereum is of size 1024. An attacker could call a contract with an already existing call stack. If a send function (or any function) is called while already at the maximum call stack 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 call stack. If an attacker tries to break the call stack, it will cause the contract to not execute.

6 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] V. Buterin. Ethereum white paper. https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf, 2014.

[2] Ether.fund. Ether.fund ether unit converter. http://ether.fund/tool/converter, 2015.

[3] Ursium newhampshire22. What options does one have if a contract is about to run out of gas? https://www.reddit.com/r/ethereum/comments/2e13vp/what_options_does_one_have_if_a_contract_is_about/, 2014.

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

[5] Pyethereum team. pyethereum/tests/test_contracts.py. https://github.com/ethereum/pyethereum/blob/develop/tests/test_contracts.py, 2015.

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

[7] Ethereum Wiki. Subtleties. https://github.com/ethereum/wiki/wiki/Subtleties, 2015.

[8] Etheruem Wiki. Serpent. https://github.com/ethereum/wiki/wiki/Serpent, 2015.

[9] Pyethereum Wiki. Using pyethereum.tester. https://github.com/ethereum/pyethereum/wiki/Using-pyethereum.tester, 2014.