Serpent

Edit New Page

JeremyVinfolio edited this page on 4 Aug · 82 revisions

Serpent is one of the high-level programming languages used to write Ethereum contracts. The language, as suggested by its name, is designed to be very similar to Python; it is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with ease-of-use in programming style, and at the same time adding special domain-specific features for contract programming. The latest version of the Serpent compiler, available on github, is written in C++, allowing it to be easily included in any client.

This tutorial assumes basic knowledge of how Ethereum works, including the concept of blocks, transactions, contracts and messages and the fact that contracts take a byte array as input and provide a byte array as output. If you do not, then go here for a basic tutorial.

This documentation is not complete and these examples may further help:

https://github.com/AugurProject/augur-core/tree/master/tests/serpent_tests

https://github.com/ethereum/serpent/tree/develop/examples

https://github.com/AugurProject/augur-core/tree/master/src

https://github.com/ethereum/dapp-bin

Differences Between Serpent and Python

The important differences between Serpent and Python are:

- Python numbers have potentially unlimited size, Serpent numbers wrap around 2^{256}. For example, in Serpent the expression `3^(2^254)` suprisingly evaluates to 1, even though in reality the actual integer is too large to be recorded in its entirety within the universe.
- Serpent has no decimals.
- Serpent has no list comprehensions (expressions like `[x**2 for x in my_list]`), dictionaries or most other advanced features
- Serpent has no concept of first-class functions. Contracts do have functions, and can call their own functions, but variables (except storage) do not persist across calls.
- Serpent has a concept of persistent storage variables (see below)
- Serpent has an `extern` statement used to call functions from other contracts (see below)

Installation

In order to install the Serpent python library and executable do:

```
$ git clone https://github.com/ethereum/serpent.git
$ cd serpent
$ git checkout develop
$ make && sudo make install
$ python setup.py install
```

You can install pyethereum itself as well:

```
$ git clone https://github.com/ethereum/pyethereum.git
$ cd pyethereum
$ git checkout develop
$ pip install -r requirements.txt
$ python setup.py install
```

Tutorial

Now, let's write our first contract. Paste the following into a file called `mul2.se`:

```
def double(x):
    return(x * 2)
```

This contract is a simple two lines of code, and defines a function. Functions can be called either by transactions or by other contracts, and are the way that Serpent contracts provide an "interface" to other contracts and to transactions; for example, a contract defining a currency might have functions `send(to, value)` and `check_balance(address)`.

Additionally, the Pyethereum testing environment that we will be using simply assumes that data input and output are in this format.

Now, let's try actually compiling the code. Type:

```
$ serpent compile mul2.se
604380600b600039604e567c0100000000000000000000000000000000000000000000000000000000600035046
```

And there we go, that's the hexadecimal form of the code that you can put into transactions. Or, if you want to see opcodes:

```
$ serpent pretty_compile mul2.se
[PUSH1, 67, DUP1, PUSH1, 11, PUSH1, 0, CODECOPY, PUSH1, 78, JUMP, PUSH29, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, PUSH1, 0,
CALLDATALOAD, DIV, PUSH4, 238, 233, 114, 6, DUP2, EQ, ISZERO, PUSH1, 65, JUMPI, PUSH1, 4,
CALLDATALOAD, PUSH1, 64, MSTORE, PUSH1, 2, PUSH1, 64, MLOAD, MUL, PUSH1, 96, MSTORE,
PUSH1, 32, PUSH1, 96, RETURN, JUMPDEST, POP, JUMPDEST, PUSH1, 0, RETURN]
```

Alternatively, you can compile to LLL to get an intermediate representation:

```
$ serpent compile_to_lll mul2.se
(return 0
  (lll
    (with '__funid
      (div (calldataload 0)
        26959946667150639794667015087019630673637144422540572481103610249216
      )
      (unless (iszero (eq (get '__funid) 4008276486))
        (seq
          (set 'x (calldataload 4))
          (seq
            (set '_temp_521 (mul (get 'x) 2))
            (return (ref '_temp_521) 32)
          )
        )
      )
    )
    0
  )
)
```

This shows you the machinery that is going on inside. As with most contracts, the outermost layer of code exists only to copy the data of the inner code during initialization and return it, since the code returned during initialization is the code that will be executed every time the contract is called; in the EVM you can see this with the `CODECOPY` opcode, and in LLL this corresponds to the `lll` meta-operation. Inside of the LLL, we have a wrapper whose purpose it is to grab the first four bytes of the contract data, (that's the `(div (calldataload 0) 26959946667150639794667015087019630673637144422540572481103610249216)`; we grab four bytes by grabbing the first 32 bytes and dividing it by 2^224) to get the function ID that is being called, and then inside of that we have a series of functions (in this case only one function) that checks the function ID provided against all function IDs supported by the contract. If the function ID is 4008276486, then it sets a variable `'x` to the message data bytes 4-35 (that's `(calldataload 4)`), sets a temporary variable to equal to `2 * x`, and returns the 32 byte memory slice that contains that variable.

- RLP Encoding
- Node Discovery Protocol
- ĐΞVp2p Wire Protocol
- ĐΞVp2p Whitepaper (WiP)
- Web3 Secret Storage

Ethereum Technologies

- Patricia Tree
- Wire protocol
- Light client protocol
- Subtleties
- Solidity Documentation
- NatSpec Format
- Contract ABI
- Bad Block Reporting
- Bad Chain Canary

Ethash/Dashimoto

- Ethash
- Ethash C API
- Ethash DAG

Whisper

- Whisper Proposal
- Whisper Overview
- PoC-1 Wire protocol
- PoC-2 Wire protocol
- PoC-2 Whitepaper

Misc

- Hard Problems of Cryptocurrency
- Chain Fibers
- Glossary

Clone this wiki locally

https://github.com/ethereum

Clone in Desktop

The function ID is calculated by computing a hash based on the function name and arguments and taking the first four bytes. In this case we have a function named `double` with a single integer as an argument; on the command line we can do:

```
$ serpent get_prefix double i
4008276486
```

Note that you can have multiple functions with the same name, if they take different combinations of inputs. For instance, a hypothetical function double that takes three integers as input (and, say, returns an array consisting of 2x each one) would have a different prefix:

```
$ serpent get_prefix double iii
1142360101
```

The letter `i` is meant for integers, and for fixed-length (up to 32 byte) strings (which are treated the same as integers in Serpent and EVM). Use the letter `s` for variable-length string arguments, and `a` for arrays; more on these later.

Now, what if you want to actually run the contract? That is where pyethereum comes in. Open up a Python console in the same directory, and run:

```
>>> from ethereum.tools import tester as t
>>> c = t.Chain()
>>> x = c.contract('mul2.se', language='serpent')
>>> x.double(42)
84
```

The second line initializes a new state (ie. a genesis block). The third line creates a new contract, and creates an object in Python which represents it. You can use `x.address` to access this contract's address. The fourth line calls the contract with argument 42, and we see 84 predictably come out.

Example: Name Registry

Having a multiply-by-two function on the blockchain is kind of boring. So let's do something marginally more interesting: a name registry. The main function will be a `register(key, value)` operation which checks if a given key was already taken, and if is unoccupied then register it with the desired value and return 1; if the key is already occupied return 0. We will also add a second function to check the value associated with a particular key. Let's try it out:

```
def register(key, value):
    # Key not yet claimed
    if not self.storage[key]:
        self.storage[key] = value
        return(1)
    else:
        return(0)  # Key already claimed

def ask(key):
    return(self.storage[key])
```

Here, we see a few parts in action. First, we have the `key` and `value` variables that the function takes as arguments. The second line is a comment; it does not get compiled and only serves to remind you what the code does. Then, we have a standard if/else clause, which checks if `self.storage[key]` is zero (ie. unclaimed), and if it is then it sets `self.storage[key] = value` and returns 1. Otherwise, it returns zero. `self.storage` is also a pseudo-array, acting like an array but without any particular memory location.

Now, paste the code into `namecoin.se`, if you wish try compiling it to LLL, opcodes or EVM, and let's try it out in the pyethereum tester environment:

```
>>> from ethereum.tools import tester as t
>>> c = t.Chain()
>>> x = c.contract('namecoin.se', language='serpent')
```

```
>>> x.register(0x67656f726765, 45)
1
>>> x.register(0x67656f726765, 20)
0
>>> x.register(0x6861727279, 65)
1
>>> x.ask(0x6861727279)
65
```

Including files, and calling other contracts

Once your projects become larger, you will not want to put everything into the same file; things become particularly inconvenient when one piece of code needs to create a contract. Fortunately, the process for splitting code into multiple files is quite simple. Make the following two files:

mul2.se:

```
def double(x):
    return(x * 2)
```

returnten.se:

```
extern mul2.se: [double:[int256]:int256]

MUL2 = create('mul2.se')
def returnten():
    return(MUL2.double(5))
```

And open Python:

```
>>> from ethereum.tools import tester as t
>>> c = t.Chain()
>>> x = c.contract('returnten.se', language='serpent')
>>> x.returnten()
10
```

Note that here we introduced several new features. Particularly:

- The `create` command to create a contract using code from another file
- The `extern` keyword to declare a class of contract for which we know the names of the functions
- The interface for calling other contracts

`create` is self-explanatory; it creates a contract and returns the address to the contract.

The way `extern` works is that you declare a class of contract, in this case `mul2`, and then list in an array the names of the functions, in this case just `double`. To generate `extern mul2.se: [double:[int256]:int256]` use

```
$ serpent mk_signature mul2.se
```

From there, given any variable containing an address, you can do `x.double(arg1)` to call the address stored by that variable. The arguments are the values provided to the function. If you provide too few arguments, the rest are filled to zero, and if you provide too many the extra ones are ignored. Function calling also has some other optional arguments:

- `gas=12414` - call the function with 12414 gas instead of the default (all gas)
- `value=10^19` - send 10^19 wei (10 ether) along with the message
- `data=x, datasz=5` - call the function with 5 values from the array `x`; note that this replaces other function arguments. `data` without `datasz` is illegal
- `outsz=7` - by default, Serpent processes the output of a function by taking the first 32 bytes and returning it as a value. However, if `outsz` is used as here, the function will instead return an array containing 7 values; if you type `y = x.fun(arg1, outsz=7)` then you will be able to access the output via `y[0]`, `y[1]`, etc.

Another similar operation to `create` is `inset('filename')`, which simply puts code into a particular place without adding a separate contract.

returnten.se

```
inset('mul2.se')

def returnten():
    return(self.double(5))
```

Storage data structures

In more complicated contracts, you will often want to store data structures in storage to represent certain objects. For example, you might have a decentralized exchange contract that stores the balances of users in multiple currencies, as well as open bid and ask orders where each order has a price and a quantity. For this, Serpent has a built-in mechanism for defining your own structures. For example, in such a decentralized exchange contract you might see:

```
data user_balances[][]
data orders[](buys[](user, price, quantity), sells[](user, price, quantity))
```

Then, you might do something like:

```
def fill_buy_order(currency, order_id):
    # Available amount buyer is willing to buy
    q = self.orders[currency].buys[order_id].quantity
    # My balance in the currency
    bal = self.user_balances[msg.sender][currency]
    # The buyer
    buyer = self.orders[currency].buys[order_id].user
    if q > 0:
        # The amount we can actually trade
        amount = min(q, bal)
        # Trade the currency against the base currency
        self.user_balances[msg.sender][currency] -= amount
        self.user_balances[buyer][currency] += amount
        self.user_balances[msg.sender][0] += amount *
self.orders[currency].buys[order_id].price
        self.user_balances[buyer][0] -= amount *
self.orders[currency].buys[order_id].price
        # Reduce the remaining quantity on the order
        self.orders[currency].buys[order_id].quantity -= amount
```

Notice how we define the data structures at the top, and then use them throughout the contract. These data structure gets and sets are converted into storage accesses in the background, so the data structures are persistent.

The language for doing data structures is simple. First, we can do simple variables:

```
data blah

x = self.blah
self.blah = x + 1
```

Then, we can do arrays, both finite and infinite:

```
data blah[1243]
data blaz[]

x = self.blah[505]
y = self.blaz[3**160]
self.blah[125] = x + y
```

Note that finite arrays are always preferred, because it will cost less gas to calculate the storage index associated with a finite array index lookup. And we can do tuples, where each element of the tuple is itself a valid data structure:

```
data body(head(eyes[2], nose, mouth), arms[2], legs[2])

x = self.body.head.nose
y = self.body.arms[1]
```

And we can do arrays of tuples:

```
data bodies[100](head(eyes[2], nose, mouth), arms[2](fingers[5], elbow), legs[2])

x = self.bodies[45].head.eyes[1]
y = self.bodies[x].arms[1].fingers[3]
```

Note that the following is unfortunately not legal:

```
data body(head(eyes[2], nose, mouth), arms[2], legs[2])

x = self.body.head
y = x.eyes[0]
```

Accesses have to descend fully in a single statement. To see how this could be used in a simpler example, let's go back to our name registry, and upgrade it so that when a user registers a key they become the owner of that key, and the owner of a key has the ability to (1) transfer ownership, and (2) change the value. We'll remove the return values here for simplicity.

```
data registry[](owner, value)

def register(key):
    # Key not yet claimed
    if not self.registry[key].owner:
        self.registry[key].owner = msg.sender

def transfer_ownership(key, new_owner):
    if self.registry[key].owner == msg.sender:
        self.registry[key].owner = new_owner

def set_value(key, new_value):
    if self.registry[key].owner == msg.sender:
        self.registry[key].value = new_value

def ask(key):
    return([self.registry[key].owner, self.registry[key].value], items=2)
```

Note that in the last ask command, the function returns an array of 2 values. If you wanted to call the registry, you would have needed to do something like `o = registry.ask(key, outsz=2)` and you could have then used `o[0]` and `o[1]` to recover the owner and value.

Simple arrays in memory

The syntax for arrays in memory are different: they can only be finite and cannot have tuples or more complicated structures.

Example:

```
def bitwise_or(x, y):
    blah = array(1243)
    blah[567] = x
    blah[568] = y
    blah[569] = blah[567] | blah[568]
    return(blah[569])
```

There are also two functions for dealing with arrays:

```
len(x)
```

Returns the length of array x.

```
slice(x, items=start, items=end)
```

Takes a slice of x starting with position start and ending with position end (note that we require `end >= start`; otherwise the result will almost certainly result in an error)

Arrays and Functions

Functions can also take arrays as arguments, and return arrays.

```
def compose(inputs:arr):
    return(inputs[0] + inputs[1] * 10 + inputs[2] * 100)

def decompose(x):
    return([x % 10, (x % 100) / 10, x / 100]:arr)
```

Putting the `:arr` after a function argument means it is an array, and putting it inside a return statement returns the value as an array (just doing `return([x,y,z])` would return the integer which is the memory location of the array).

If a contract calls one of its functions, then it will autodetect which arguments should be arrays and parse them accordingly, so this works fine:

```
def compose(inputs:arr, radix):
    return(inputs[0] + inputs[1] * radix + inputs[1] * radix ** 2)

def main():
    return self.compose([1,2,3,4,5], 100)
```

However, if a contract wants to call another contract that takes arrays as arguments, then you will need to put a "signature" into the extern declaration:

```
extern composer: [compose:[int256[],int256]:int256, main:[]:int256]
```

If you want to determine the signature to use from a given file, you can do:

```
> serpent mk_signature compose_test.se
extern compose_test: [compose:[int256[],int256]:int256, main:[]:int256]
```

Strings

There are two types of strings in Serpent: short strings, eg. `"george"`, and long strings, eg. `text("afjqwhruqwhurhqkwrhguqwhrkuqwrkqwhwhrugquwrguwegtwetwet")`. Short strings, given simply in quotes as above, are treated as numbers; long strings, surrounded by the `text` keyword as above, are treated as array-like objects; you can do `getch(str, index)` and `setch(str, index)` to manipulate characters in strings (doing `str[0]` will treat the string as an array and try to fetch the first 32 characters as a number).

To use strings as function arguments or outputs, use the `str` tag, much like you would use `arr` for arrays. `len(s)` gives you the length of a string, and `slice` works for strings the same way as for arrays too.

Here is an example of returning/retrieving a string:

```
data str

def t2():
    self.str = text("01")
    log(data=self.str)
    return(self.str, chars=2)

def runThis():
    s = self.t2(outsz=2)
    log(data=s)
```

Macros

WARNING: Relatively new/untested feature, here be ~~dragons~~ serpents

Macros allow you to create rewrite rules which provide additional expressivity to the language. For example, suppose that you wanted to create a command that would compute the median of three values. You could simply do:

```
macro median($a, $b, $c):
    min(min(max($a, $b), max($a, $c)), max($b, $c))
```

Then, if you wanted to use it somewhere in your code, you just do:

```
x = median(5, 9, 7)
```

Or to take the max of an array:

```
macro maxarray($a:$asz):
    $m = 0
    $i = 0
    while i < $asz:
        $m = max($m, $a[i])
        $i += 1
    $m

x = maxarray([1, 9, 5, 6, 2, 4]:6)
```

For a highly contrived example of just how powerful macros can be, see https://github.com/ethereum/serpent/blob/poc7/examples/peano.se

Note that macros are not functions; they are copied into code every time they are used. Hence, if you have a long macro, you may instead want to make the macro call an actual function. Additionally, note that the dollar signs on variables are important; if you omit a dollar sign in the pattern $a then the macro will only match a variable actually called a. You can also create dollar sign variables that are in the substitution pattern, but not the search pattern; this will generate a variable with a random prefix each instance of the macro. You can also create new variables without a dollar sign inside a substitution pattern, but then the same variable will be shared across all instances of the pattern and with uses of that variable outside the pattern.

Types

WARNING: Relatively new/untested feature, here be ~~dragons~~ serpents

An excellent compliment to macros is Serpent's ghetto type system, which can be combined with macros to produce quite interesting results. Let us simply show this with an example:

```
type float: [a, b, c]

macro float($x) + float($y):
    float($x + $y)

macro float($x) - float($y):
    float($x - $y)

macro float($x) * float($y):
    float($x * $y / 2^32)

macro float($x) / float($y):
    float($x * 2^32 / $y)

macro unfloat($x):
    $x / 2^32

macro floatfy($x):
    float($x * 2^32)

macro float($x) = float($y):
    $x = $y
```

```
macro with(float($x), float($y), $z):
    with($x, $y, $z)

a = floatfy(25)
b = a / floatfy(2)
c = b * b
return(unfloat(c))
```

This returns 156, the integer portion of 12.5^2. A purely integer-based version of this code would have simply returned 144. An interesting use case would be rewriting the elliptic curve signature pubkey recovery code using types in order to make the code neater by making all additions and multiplications implicitly modulo P, or using long integer types to do RSA and other large-value-based cryptography in EVM code.

Miscellaneous

The three other useful features in the tester environment are:

- Block access - you can dig around `s.block` to see block data (eg. `s.block.number`, `s.block.get_balance(addr)`, `s.block.get_storage_data(addr, index)`)
- Snapshots - you can do `x = s.snapshot()` and `s.revert(x)`
- Advancing blocks - you can do `s.mine(100)` and 100 blocks magically pass by with a 60-second interval between blocks. `s.mine(100, addr)` mines into a particular address.
- Full block data dump - type `s.block.to_dict()`

Serpent also gives you access to many "special variables"; the full list is:

- `tx.origin` - the sender of the transaction
- `tx.gasprice` - gas price of the transaction
- `msg.gas` - estimated gas by sender
- `msg.sender` - the sender of the message
- `msg.value` - the number of wei (smallest units of ether) sent with the message
- `self` - the contract's own address
- `self.balance` - the contract's balance
- `x.balance` (for any x) - that account's balance
- `block.coinbase` - current block miner's address
- `block.timestamp` - current block timestamp
- `block.prevhash` - previous block hash
- `block.difficulty` - current block difficulty
- `block.number` - current block number
- `block.gaslimit` - current block gaslimit

Serpent recognises the following "special functions":

- `def init():` - executed upon contract creation, accepts no parameters
- `def shared():` - executed before running `init` and user functions
- `def any():` - executed before any user functions

There are also special commands for a few crypto operations; particularly:

- `addr = ecrecover(h, v, r, s)` - determines the address that produced the elliptic curve signature `v, r, s` of the hash `h`
- `x = sha256(a, items=4)` - returns the sha256 hash of the 128 bytes consisting of the 4-item array starting from `a`
- `x = ripemd160(a, items=4)` - same as above but for ripemd160
- To hash an arbitrary number of bytes, use chars syntax. Example: `x = sha256([0xf1fc122bc7f5d74df2b9441a42a14695000000000000000000000000000000000], chars=16)` - returns the sha256 of the first 16 bytes. Note: padding with trailing zeroes, otherwise the first 16 bytes will be zeroes, and the sha256 of it will be computed instead of the desired.
- you can also use sha3 instead of sha256, e.g. `sha3(a, items=4)`

Tips

- If a function is not returning the result you expect, double-check that all variables are correct: there is no error/warning when using an undeclared variable.
- `Invalid argument count or LLL function` usually means you just called foo() instead of `self.foo()`.
- Sometimes you may be intending to use unsigned operators. eg div() and lt() instead of '/' and '<'.
- To upgrade Serpent, you may need to do `pip uninstall ethereum-serpent` and `python setup.py install`. (Avoid `pip install ethereum-serpent` since it will get from PyPI which is probably old.) (Also avoid using the master branch, which is probably even older than the PyPI version; use the develop branch instead.)
- When calling abi_contract(), if you get this type of error `Exception: Error (file "main", line 1, char 5): Invalid object member (ie. a foo.bar not mapped to anything)` make sure you are specifying correct path to the file you are compiling.
- If you get a core dump when calling `abi_contract()`, check that you do not have functions with the same name.
- Use macro for constants, example:

```
macro CONSTANT: 99
```

- Be careful that if your flow requires going through a number of contracts, that someone can't just directly short-circuit and call one of your latter contracts with data they've manipulated elsewhere. Example: If you have contract C which gives someone ether, but relies on computation from Contract A->B->C, that someone can't just call B or C to give themselves ether.

Other

http://mc2-umd.github.io/ethereumlab/docs/serpent_tutorial.pdf - some outdated but can generally be helpful

[English | Deutsch | Español | Français | 日本語 | Română | فارسی | Italiano | 한국어 | 中文]