Smart contract development using Solidity

Hypermine
TECHNOLOGIES

Introduction to smart contracts

- Act as normal contract but fully digitise.
- Collection of code that resides on an address (CA)

Properties

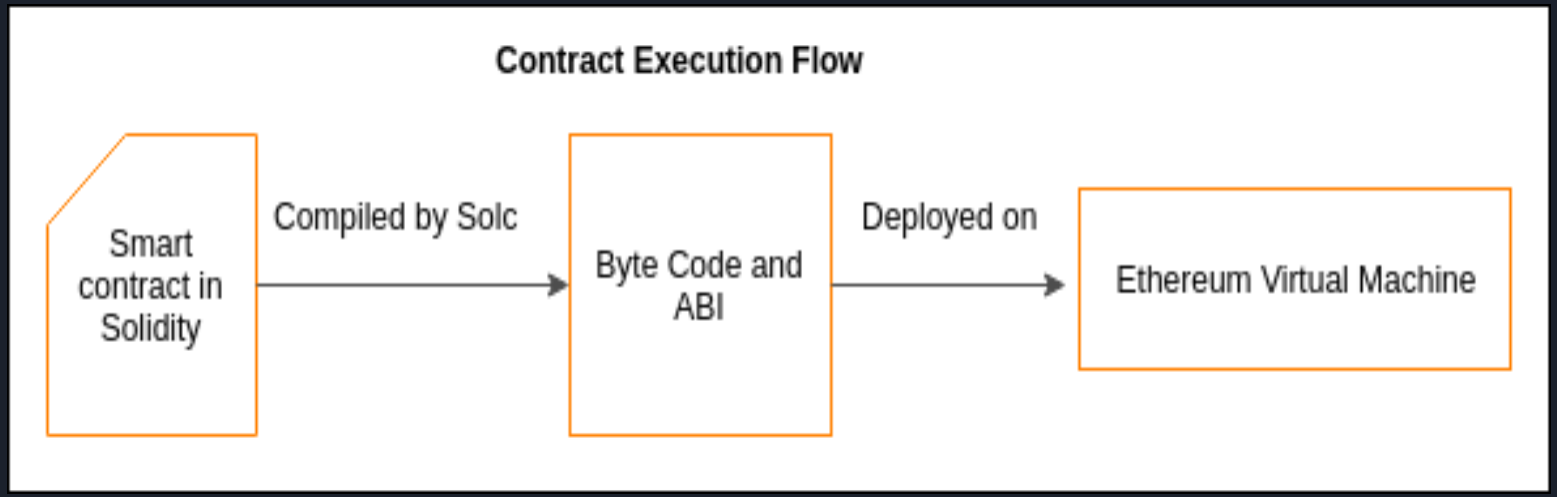
- Self verifying
- Self executing : Remove third party
- Written in HLL and compiled into bytecode.
- Resides on blockchain in bytecode format

Hypermine
TECHNOLOGIES

HLL to write smart contracts

- Solidity : similar to javascript
- Serpent : similar to Python
- Lisp Like Language (LLL) : similar to Assembly

How does it works?



Hypermine
TECHNOLOGIES

Solidity - v0.4.24

Hypermine
TECHNOLOGIES

Basic Data Types - Integer, Bool

Value types : Passed by value

- Boolean
  - Operators : **!, ||, &&, ==, !=**
  - Default : **false**
- Integer
  - int / uint : signed or unsigned of various types
  - Operators:
    - Comparison : **>,>=,<,<=,==,!=**
    - Bitwise : **&, |, ^,~**
    - Arithmetic : **+, *, %, /, - , ** , <<, >>**
  - Default **0**

Hypermine
TECHNOLOGIES

Basic Data Types - Address

address

- Holds 20 byte value
- Initialize by 0x0
- Members
  - Balance : query balance
  - **Transfer** : send ether, throws on failure
  - Send : send ether, return false on failure ; DANGER

Hypermine
TECHNOLOGIES

Complex Data Types - Arrays 1

Array

- Fixed-sized arrays
  - Created by specifying the type and size
  - Ex. bool [10] array, int8[10] array
- Dynamic-sized
  - Created by specifying the type
  - Ex. int8[] arrayInt, bool[] arrayBool
  - **new** operator for allocating memory. Ex. arrayInt = new int8[](10);

Hypermine
TECHNOLOGIES

Complex Data Types - Arrays 2

Special Array

- Byte & Bytes
  - byte[15] data : static
  - byte [] data : dynamic
  - bytes1 data = byte [1] data
  - bytes32 //32 byte array
  - bytes1, bytes2, ... bytes32
- String
  - Not a basic type
  - Dynamically sized
  - Fixed length not supported, like string[7] //error
  - Member *.length* is not present as well

Hypermine
TECHNOLOGIES

Complex Data Types - Mapping, Enum

Mapping

- Similar to hashtable or dictionary object : key-value pair
- Allowed as **storage** or **state** variable
- E.x. *mapping(address => uint) balances;*
- Not iterable : can not loop through

Enum

- User defined types with finite set of values
- Ex. *enum gender {MALE, FEMALE, OTHER}*
- Not a part of ABI

Hypermine
TECHNOLOGIES

Complex Data Types - Struct

Struct

- User defined types declared with keyword *struct*.
- Can group several types
- Can not have members of its own types
- Can be contained in array and mapping
- Not part of ABI

```
struct Candidate {
    string name;
    uint age;
    string gender;
}

Candidate candidateObj;
```

Hypermine
TECHNOLOGIES

Type Conversion

Implicit

- Compiler allows if no data loss

```
contract TypeConversion {

    //Implicit type conversion
    uint8 i8 = 255; //store 1 byte

    uint8 i8 = 256; //erros as cant store more than 1 byte

    function ImplicitConversions() public view {
        int i256; //int means int256,
                  //hence can store 256 bit or 32 byte data
        i256 = i8; //No error
        i8 = i256;// fails as maximum value of int256
                  //can not be stored in uint8
    }

}
```

Explicit

- Loss of information

```
function ExplicitConversion() public view {
    //Explicit type conversion
    uint32 i32 = 200;
    uint24 i24 = uint24(i32); // potential data loss
}
```

Hypermine
TECHNOLOGIES

Data Location

Complex types need memory for storage of data. Depending upon their context, EVM logically divides the memory into 3 regions:

Storage

- Persistence/ Permanent
- Read and Write operations are costly.
- State variables, Local variables by default

Memory

- Non persistence / Temporary
- Cheap as compared to Storage
- Function args and return params by default

Calldata

- Non persistence / Temporary
- Non Modifiable
- Used in function calls

```
contract MemoryMgmt {

    // x is state variable
    uint[] x; // the data location ?

    //the data location of memoryArray ?
    function f(uint[] memoryArray) public {
        x = memoryArray; // copies the whole array to storage

        // y and z are local variables
        uint[] y = x; // the data location ?
        uint[3] memory z; // default can be overridden

        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    // default can be overridden
    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public returns (bool){}
}
```

Hypermine
TECHNOLOGIES

Visibility

Private

- Visible in the contract its defined, not even from derived contract

Public

- Visible from within and outside
- By default functions are public
- For public state variables, compiler automatically creates Getter method

Internal

- Accessed from within the contract and contract deriving from it.
- State variables are internal by default

External

- Called from other contract (using its address) or by itself by message call

Hypermine
TECHNOLOGIES

Function Modifiers

- Re-executable piece of code
- Helps to change behaviour of a function
- Check certain conditions before even executing the function
- Hence can reject the function call.
- Can have arguments just like methods have
- ***Payable, view, pure*** are inbuilt modifier

```
contract SimpleModifier {

    address owner; //store owner's address
    string private name;

    constructor() public{
        // setting owner as the creator of this contract
        owner = msg.sender;
    }

    //modifier onlyOwner ensures that only owner
    //of the contract can call the function
    modifier onlyOwner{
        require(msg.sender == owner); // caller should be the owner
        _; //execute the function body
    }

    //only owner can all this method
    function setValue(string _name) public onlyOwner returns(bool){
        name = _name;
    }

    // any one can call this method
    function getValue() public view returns(string){
        return name;
    }

}
```

Hypermine
TECHNOLOGIES

Events

- Used to capture log
- Implemented using ***event*** keyword
- Event can be emitted by smart contract using ***emit*** keyword
- Once the transaction mined, logs in all nodes gets updated
- Dapps can watch these events

```
contract SimpleEvent {

 uint8 i;

 //event for setting value
 event valueSet(uint8 value);

 constructor() public{}

 function setValue(uint8 _value) public {
    i = _value;
    // emit the event when new value is set
    emit valueSet(i);
 }
}
```

Hypermine
TECHNOLOGIES

Keywords : View, Pure

View

- Allowed to read the storage variables.
- Can not change the state of contract.

Pure

- Can not even read storage variables.
- Can not change the state of contract

Keywords : Fallback, Payable

“Well, try to learn yourself?”

Hypermine
TECHNOLOGIES

Error Handling

- No try-catch
- **revert()**
  - Throws an exception
  - Refunds the unused gas
  - Abort execution and revert state changes
- **require(condition)**
  - If condition is not met, then throw exception
  - Unused gas returned
- **assert(condition)**
  - If condition is not met, then throw exception
  - All gas consumed
  - Runtime exceptions, like divide by 0

Hypermine
TECHNOLOGIES

Interface, Library, Inheritance

Interface

- Function can’t have implementation
- Can’t have constructor, variable, struct, enum, can’t inherit interface or contract
- Inherited class must implement all methods of interface
- Starts with ***interface*** keyword.

Library

- Similar to contract
- Starts with ***library*** keyword
- Can write all structs, enums and use it anywhere in the project.

Hypermine
TECHNOLOGIES

References

http://www.ethdocs.org/

https://github.com/ethereum/wiki/wiki/Useful-%C3%90app-Patterns

http://solidity.readthedocs.io

Hypermine
TECHNOLOGIES