

Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode

Matt Suiche
Comae Technologies
m@comae.io

July 7, 2017

Abstract

Ethereum is gaining a significant popularity in the blockchain community, mainly due to fact that it is design in a way that enables developers to write decentralized applications (Dapps) and smart-contract using blockchain technology. This new paradigm of applications opens the door to many possibilities and opportunities. Blockchain is often referred as secure by design, but now that blockchains can embed applications this raise multiple questions regarding architecture, design, attack vectors and patch deployments. In this paper I will discuss the architecture of the core component of Ethereum (Ethereum Virtual Machine), its vulnerabilities as well as my open-source tool “Porosity”. A decompiler for EVM bytecode that generates readable Solidity syntax contracts. Enabling static and dynamic analysis of such compiled contracts.

Contents

1	Ethereum Virtual Machines (EVM)	4
2	Memory Management	4
2.1	Stack	4
2.2	Storage (Persistent)	4
2.3	Memory (Volatile)	5
3	Addresses	6
4	Call Types	6
4.1	EVM	6
4.1.1	Basic Blocks	6
4.1.2	EVM functions	7
4.1.3	EVM Call	8
4.2	User-defined functions (Solidity)	9
5	Type Discovery	10
5.1	Address	10
5.1.1	Non-optimized Address Mask	10
5.1.2	Optimized Address Mask	10
5.1.3	Parameter Address Mask	12
6	Smart-Contract	12
6.1	Pre-Loader	12
6.2	Runtime Dispatcher	13
6.2.1	Function Hashes	14
6.2.2	Dispatcher	15
7	Code Analysis	20
7.1	Vulnerable Contract	20
7.1.1	Solidity source code	21
7.1.2	Runtime Bytecode	22
7.1.3	ABI Definition	23
7.1.4	Decompiled version	24

8	Bugs	24
8.1	Reentrant Vulnerability / Race Condition	24
8.2	Call Stack Vulnerability	25
8.3	Time Dependence Vulnerability	25
9	Future	25
10	Acknowledgments	26

List of Figures

1	Static CFG	19
2	Emulated CFG	19

List of Listings

1	Storage (Persistent) Exmample	5
2	EVM Parameter/Return Stack Location Example	7
3	call Proto-type Declaration	8
4	Pre-compiled Contracts	9
5	CALLDATALOAD Example	9
6	CALLDATALOAD EVM Pseudo-code	10
7	Non-optimized Assembly Code Example	10
8	Optimized Assembly Code Example	11
9	msg.sender EVM Bytecode Example	11
10	Parameter Address Mask Example	12
11	Porosity Pre-loader Disassembly Output	13
12	ABI Definition	14
13	double Function Declaration	14
14	double/triple Function Hashes	15
15	EVM Runtime Bytecode Example	15
16	Runtime Bytecode Porosity Disassembly	17
17	dispdiasm	18
18	EVM Emulator	18
19	Static/Dynamic Graph Pseudo-C Code	19
20	Decompiled Pseudo-C code	20
21	Vulnerable Smart Contract	21
22	Vulnerable Smart Contract Runtime Bytecode	22
23	Vulnerable Smart Contract ABI Definition	23
24	Vulnerable Smart Contract Decompilation	24

1 Ethereum Virtual Machines (EVM)

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts in Ethereum. The EVM runs smart-contracts that are built up from bytecodes. Bytecodes are identified by a 160-bit address, and stored in the blockchain, which is also known as “accounts”. The EVM operates on 256-bit pseudo registers. Which means that the EVM does not operate via registers. But, through an expandable stack which is used to pass parameters not only to functions/instructions, but also for memory and other algorithmic operations.

The following excerpt is taken from the Solidity documentation, and it is also worth mentioning:

There are two kinds of accounts in Ethereum which share the same address space: External accounts that are controlled by public-private key pairs (i.e. humans) and contract accounts which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called “nonce”).

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

2 Memory Management

2.1 Stack

It does not have the concept of registers. A virtual stack is being used instead for operations such as parameters for the opcodes. The EVM uses 256-bit values from that virtual stack. It has a maximum size of 1024 elements.

2.2 Storage (Persistent)

The Storage is a persistent key-value storage mapping (256-to-256-bit integers). And is documented as below:

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called storage. Furthermore, every account has a balance which can be modified by sending transactions.

Each account has a persistent memory area which is called storage. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract and it is comparatively costly to read and even more so, to modify storage. A contract can neither read nor write to any storage apart from its own.

The storage memory is the memory declared outside of the user-defined functions and within the Contract context. For instance, in listing 1, the `userBalances` and `withdrawn` will be in the memory storage. This can also be identified by the `SSTORE` / `SLOAD` instructions.

```
1 contract SendBalance {
2     mapping ( address => uint ) userBalances;
3     bool withdrawn = false;
4     (...)
5 }
```

Listing 1: Storage (Persistent) Exmample

2.3 Memory (Volatile)

This memory is mainly used when calling functions or for regular memory operations. The official documentation explicitly indicates that the EVM does not have traditional registers. Which means that the virtual stack previously discussed will be used primarily to push arguments to the instructions. The following is the excerpt explaining such behavior:

The second memory area is called memory, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing

(either reading or writing) a previously untouched memory word (ie. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

Traditionally the `MSTORE` instruction is what we would generally consider to be the instruction responsible for adding data to the stack in any typical x86/x64 system. Therefore, the instructions `MSTORE` / `MLOAD` could be identified as such with respect to the x86/x64 system. Consequently, both `mstore(where, what)` and `mload(where)` are frequently used.

3 Addresses

EVM uses 160-bit addresses. It is extremely crucial to understand that fact when one has to deal with type discovery. As we often see the mask `0xffffffffffffffffffffffffffffffff` being applied for optimization purposes either on code or on the EVM registers.

4 Call Types

There are two types of functions to differentiate when working with the EVM. The first type is the EVM functions (or EVM instructions), while the second type is the user-defined function when creating the smart-contract.

4.1 EVM

4.1.1 Basic Blocks

Basic Blocks usually starts with the instruction `JUMPDEST`, with the exception of very few exception cases. Most of the conditional and unconditional jumps have a `PUSH` instruction preceding them in order to push the destination offset into the stack. Although, in some cases we would also notice that the `PUSH` instruction containing the offset can be executed way before the actual `JUMP` instruction, and retrieved using stack manipulation instructions such as `DUP`, `SWAP` or `POP`. Those cases require dynamic execution of the code to record the stack for each `JUMP` instruction, as we are going to discuss this later on in sub-section 6.2.2.

4.1.2 EVM functions

EVM functions and/or instructions includes, but are not limited to, some of the the following:

- Arithmetic Operations.
- Comparison & Bitwise Logic Operations.
- SHA3.
- Environmental Information.
- Block Information.
- Stack, Memory, Storage and Flow Operations.
- Push/Duplication/Pop/Exchange Operations.
- Logging Operations.
- System Operations.

Since the EVM does not have registers, therefore all instructions invocation are done through the EVM stack. For example, an instruction taking two parameters such as an addition or a subtraction, would use the stack entries index 0 and 1. And the return value would be stored in the stack entry index 0. In listing 2, we can see more clearly how it looks like under the hood.

```
1 PUSH1 0x1 ==> {stack[0x0] = 0x1}
2 PUSH2 0x2 ==> {stack[0x0] = 0x2, stack[0x1] = 0x1}
3 ADD      ==> {stack[0x0] = 0x3}
```

Listing 2: EVM Parameter/Return Stack Location Example

The above EVM assembly snippet would translate to the EVM pseudo-code `add(0x2, 0x1)` and returns 0x3 in the stack entry 0. The EVM stack model follows the standard last-in, first-out (LIFO) algorithm.

4.1.3 EVM Call

There are two possible types of external EVM function calls. They can be identified with the `CALL` instruction. However, this is not necessarily always a concrete identifier to the call being external.

Some mathematical and cryptographic functions have to be called through external contracts such as `sha256` or `ripemd160` using the `call` function. Despite the fact of having an explicitly defined instruction for the `sha3` function. Which is due to the frequent usage, especially with mapping arrays such as `mapping(address => uint256) balances`. Where the `sha3` function is used to compute the index.

The function `call` is where the dispatching magic happens. Listing 3 shows the proper proto-type declaration for such function.

```
1 call(  
2     gasLimit,  
3     to,  
4     value,  
5     inputOffset,  
6     inputSize,  
7     outputOffset,  
8     outputSize  
9 )
```

Listing 3: call Proto-type Declaration

There are four ‘pre-compiled’ contracts that are present as extensions of the current design. The four contracts in addresses 1, 2, 3 and 4 executes the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme and the identity function respectively. Listing 4 shows such contracts, obtained from the EVM source code.

```

1 precompiled.insert(
2     make_pair(Address(1), PrecompiledContract(3000, 0,
3         PrecompiledRegistrar::executor("ecrecover"))));
4
5 precompiled.insert(
6     make_pair(
7         Address(2),
8         PrecompiledContract(
9             60,
10            12,
11            PrecompiledRegistrar::executor("sha256"))));
12
13 precompiled.insert(
14     make_pair(Address(3), PrecompiledContract(600, 120,
15         PrecompiledRegistrar::executor("ripemd160"))));
16
17 precompiled.insert(
18     make_pair(Address(4), PrecompiledContract(15, 3,
19         PrecompiledRegistrar::executor("identity"))));

```

Listing 4: Pre-compiled Contracts

4.2 User-defined functions (Solidity)

In order to call user-defined functions, another level of abstraction is managed by the instruction `CALLDATALOAD`. The first parameter for that instruction is the offset in the current environment block.

The first 4-bytes indicates the 32-bit hash of the called function. Then the input parameters follows next. Listing 5, shows an example of such case.

```

1 function foo(int a, int b) {
2     return a + b;
3 }

```

Listing 5: CALLDATALOAD Example

In the previous example, the outcome of such code snippet would be `a = calldataload(0x4)` and `b = calldataload(0x24)`. Its imperative to remember that by default “registers” are 256-bits. Since the first 4 bytes are

pre-allocated for the function's hash value, therefore the first parameter will be at the offset 0x4, followed by the second parameter at offset 0x24. This is derived mathematically by simply calculating the number of bytes added to the previous number of bytes taken by the first parameter. So in short words, $4 + (256/8) = 0x24$. We can then conclude the EVM pseudo-code shown in listing 6.

```
1 return(add(calldataload(0x4), calldataload(0x24)))
```

Listing 6: CALLDATALOAD EVM Pseudo-code

5 Type Discovery

5.1 Address

Addresses can be identified by their sources such as specific instruction such as caller but in most of cases we can proceed to better results by identifying mask applied to those values.

5.1.1 Non-optimized Address Mask

In listing 7, the 0x16 bytes EVM assembly code would translate to `reg256` and `0xffffffffffffffffffffffffffffffff`.

```
1 00000188 73ffffffff + PUSH20 ffffffffffffffffffffffffffffffffff
2 0000019d 16          AND
```

Listing 7: Non-optimized Assembly Code Example

5.1.2 Optimized Address Mask

Listing 8 shows the optimized 0x9 bytes EVM assembly code, which also yields the same operation as shown previously in listing 7.

1	00000043	6001	PUSH1	0x01
2	00000045	60A0	PUSH1	0xA0
3	00000047	6002	PUSH1	0x02
4	00000049	0A	EXP	
5	0000004A	03	SUB	
6	0000004B	16	AND	

Listing 8: Optimized Assembly Code Example

We can then translate the EVM assembly code shown in listing 8 to the following 3 items:

- `and(reg256, sub(exp(2, 0xA0), 1))` (EVM)
- `reg256 & (2 ** 0xA0) - 1` (Intermediate)
- `address` (Solidity)

With that being said, in listing 9 For instance, the following EVM bytecode would simply yield as the equivalence of `msg.sender` variable in Solidity format.

1	CALLER
2	PUSH1 0x01
3	PUSH 0xA0
4	PUSH1 0x02
5	EXP
6	SUB
7	AND

Listing 9: msg.sender EVM Bytecode Example

5.1.3 Parameter Address Mask

```
1 0000003a 6004          PUSH1 04
2 0000003e 35          CALLDATALOAD
3 ...
4 00000058 73fffffff + PUSH20 ffffffffffffffffffffffffffffffffff
5 0000006d 16          AND
6 0000006e 6c00000000 + PUSH13 000000000000000000000001
7 0000007c 02          MUL
```

Listing 10: Parameter Address Mask Example

In listing 10, we can see that the EVM assembly code for what would translate to `mul(and(arg_4, 0xffffffffffffffffffffffffffffffff), 0x10000000000000000000000000000000)`, which is in fact an optimization to mask the addresses as parameters before storing them in memory.

6 Smart-Contract

When compiling a new smart-contract with Solidity, you will be asked to choose between two options to retrieve the bytecode as shown below.

- `-bin`
- `-bin-runtime`

The first one will output the binary of the entire contract, which includes its pre-loader. While the second one will output the binary of the runtime part of the contract which is the part we are interested in for analysis.

6.1 Pre-Loader

Listing 11 is a copy of the output from the porosity disassembler representing the pre-loader.

The instruction `CODECOPY` is used to copy the runtime part of the contract in EVM's memory. The offset `0x002b` is the runtime part, while `0x00` is the destination address.

Note that in Ethereum assembly, `PUSH / RETURN` means the value pushed will be the returned value from the function and won't affect the execution address.

```
1 00000000 6060          PUSH1    60
2 00000002 6040          PUSH1    40
3 00000004 52             MSTORE
4 00000005 6000          PUSH1    00
5 00000007 6001          PUSH1    01
6 00000009 6000          PUSH1    00
7 0000000b 610001        PUSH2    0001
8 0000000e 0a            EXP
9 0000000f 81            DUP2
10 00000010 54            SLOAD
11 00000011 81            DUP2
12 00000012 60ff         PUSH1    ff
13 00000014 02            MUL
14 00000015 19            NOT
15 00000016 16            AND
16 00000017 90            SWAP1
17 00000018 83            DUP4
18 00000019 02            MUL
19 0000001a 17            OR
20 0000001b 90            SWAP1
21 0000001c 55            SSTORE
22 0000001d 50            POP
23 0000001e 61bb01       PUSH2    bb01
24 00000021 80            DUP1
25 00000022 612b00       PUSH2    2b00
26 00000025 6000          PUSH1    00
27 00000027 39            CODECOPY
28 00000028 6000          PUSH1    00
29 0000002a f3           RETURN
```

Listing 11: Porosity Pre-loader Disassembly Output

6.2 Runtime Dispatcher

At the beginning of each runtime part of contracts, we find a dispatcher that branches to the right function to be called when invoking the contract.

6.2.1 Function Hashes

As we discussed earlier in the user-defined function section, the first 4 bytes of the environment block are used to pass the function hash to the runtime dispatcher that we will describe shortly. The function hash itself is generated from the ABI definition of the function using the logic presented in listing 12.

```
1  [
2    {
3      "constant":false,
4      "inputs":[{"name":"a", "type":"uint256" }],
5      "name":"double",
6      "outputs":[{"name":"", "type":"uint256" }],
7      "type":"function"
8    },
9    {
10     "constant":false,
11     "inputs":[{"name":"a", "type":"uint256" }],
12     "name":"triple",
13     "outputs":[{"name":"", "type":"uint256" }],
14     "type":"function"
15   }
16 ]
```

Listing 12: ABI Definition

We take the first 4 bytes of the `sha3` (keccak256) value for the string `functionName(param1Type, param2Type, etc)`. For instance, if we consider the above function to be declared as `double` then we also need to consider the string `double(uint256)` as illustrated below in listing 13:

```
1 keccak256("double(uint256)") =>
2   eee972066698d890c32fec0edb38a360c32b71d0a29ffc75b6ab6d2774ec9901
```

Listing 13: double Function Declaration

This means that the function signature/hash is `0xeee97206` as extracted from the return value shown above in listing 13. If we repeat the same

operation for the `triple(uint256)` function then we will get the values shown in listing 14.

```
1 Contract::setABI: Name: double(uint256)
2 Contract::setABI: signature: 0xeee97206
3
4 Contract::setABI: Name: triple(uint256)
5 Contract::setABI: signature: 0xf40a049d
```

Listing 14: double/triple Function Hashes

6.2.2 Dispatcher

Using the `--disasm` parameter of Porosity and by providing the `--abi` definition as well, Porosity will then generate a readable disassembly output resolving the symbols based on the ABI definition. Not only that, but also isolate each basic block which will help a lot in the explanation of this section. We can go ahead and examine the runtime bytecode shown in listing 15.

```
1 606060405260e06 \
2 0020a6000350463 \
3 eee972068114602 \
4 4578063f40a049d \
5 146035575b005b6 \
6 045600435600060 \
7 4f8260025b02905 \
8 65b604560043560 \
9 00604f826003603 \
10 1565b6060908152 \
11 602090f35b92915 \
12 05056
```

Listing 15: EVM Runtime Bytecode Example

Porosity will generate the following disassembly for the previously mentioned runtime bytecode which was obtained from the EVM itself as being shown in listing 16.

```

1  loc_00000000:
2  0x00000000 6060          PUSH1    60
3  0x00000002 6040          PUSH1    40
4  0x00000004 52           MSTORE
5  0x00000005 60e0          PUSH1    e0
6  0x00000007 60 02        PUSH1    02
7  0x00000009 0a           EXP
8  0x0000000a 6000          PUSH1    00
9  0x0000000c 35           CALLDATALOAD
10 0x0000000d 04           DIV
11 0x0000000e 630672e9ee   PUSH4    0672e9ee
12 0x00000013 81           DUP2
13 0x00000014 14           EQ
14 0x00000015 6024          PUSH1    24
15 0x00000017 57           JUMPI
16
17 loc_00000018:
18 0x00000018 80           DUP1
19 0x00000019 639d040af4   PUSH4    9d040af4
20 0x0000001e 14           EQ
21 0x0000001f 6035          PUSH1    35
22 0x00000021 57           JUMPI
23
24 loc_00000022:
25 0x00000022 5b           JUMPDEST
26 0x00000023 00           STOP
27
28 double(uint256):
29 0x00000024 5b           JUMPDEST
30 0x00000025 6045          PUSH1    45
31 0x00000027 6004          PUSH1    04
32 0x00000029 35           CALLDATALOAD
33 0x0000002a 6000          PUSH1    00
34 0x0000002c 604f          PUSH1    4f
35 0x0000002e 82           DUP3
36 0x0000002f 6002          PUSH1    02
37
38 loc_00000031:
39 0x00000031 5b           JUMPDEST
40 0x00000032 02           MUL
41 0x00000033 90           SWAP1
42 0x00000034 56           JUMP

```

```

43 triple(uint256):
44 0x00000035 5b          JUMPDEST
45 0x00000036 6045      PUSH1    45
46 0x00000038 6004      PUSH1    04
47 0x0000003a 35        CALLDATALOAD
48 0x0000003b 6000      PUSH1    00
49 0x0000003d 604f      PUSH1    4f
50 0x0000003f 82        DUP3
51 0x00000040 6003      PUSH1    03
52 0x00000042 6031      PUSH1    31
53 0x00000044 56        JUMP
54
55 loc_00000045:
56 0x00000045 5b          JUMPDEST
57 0x00000046 6060      PUSH1    60
58 0x00000048 90        SWAP1
59 0x00000049 81        DUP2
60 0x0000004a 52        MSTORE
61 0x0000004b 6020      PUSH1    20
62 0x0000004d 90        SWAP1
63 0x0000004e f3        RETURN
64
65 loc_0000004f:
66 0x0000004f 5b          JUMPDEST
67 0x00000050 92        SWAP3
68 0x00000051 91        SWAP2
69 0x00000052 50        POP
70 0x00000053 50        POP
71 0x00000054 56        JUMP

```

Listing 16: Runtime Bytecode Porosity Disassembly

First, the dispatcher reads the 4 bytes function hash from the environment block by calling `calldataload(0x0) / exp(0x2, 0xe0)`. Since the `CALLDATALOAD` instruction reads a 256-bit integer by default, therefore it is followed by a division to filter the first 32-bits out.

```

1 (0x12345678aaaaaaaaabbbbbbbccccccddddd000000000000000000000000 /
2 0x00000001000000000000000000000000000000000000000000000000000000)
3 = 0x12345678

```

Listing 17: dispdisasm

We can try and emulate the code using the EVM emulator or using porosity as long as Ethereum is used in the following manner as illustrated in listing 18.

```

1 PS C:\Program Files\Geth> .\evm.exe \
2 --code 60e060020a6000350463deadbabe \
3 --debug \
4 --input 12345678aaaaaaaaabbbbbbbccccccddddd
5 PC 00000014: STOP GAS: 999999920 COST: 0
6 STACK = 2
7 0000: 0000000000000000000000000000000000000000000000000000000000000000deadbabe
8 0001: 000000000000000000000000000000000000000000000000000000000000000012345678
9 MEM = 0
10 STORAGE = 0

```

Listing 18: EVM Emulator

We can notice there are two `PUSH4` instructions that corresponds to the function hashes we previously computed.

In the above scenario the equivalent EVM code would translate to the pseudo-code `jumpi(eq(calldataload(0x0) / exp(0x2, 0xe0), 0xee97206))`. Using Control Flow Graph (CFG) feature of Porosity, we can generate a static CFG or a dynamic CFG. Both graphs will be generated in GraphViz format.

Static CFG often contains orphan basic blocks, due to the fact that some destination addresses are computed at runtime. While the dynamic CFG resolves those orphan basic blocks by emulating the code as we can see in the output of both fig. 1 and fig. 2.

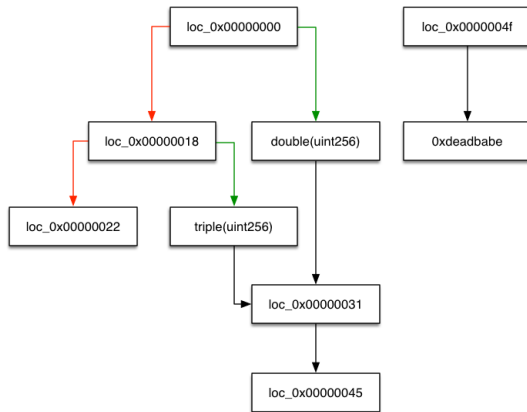


Figure 1: Static CFG

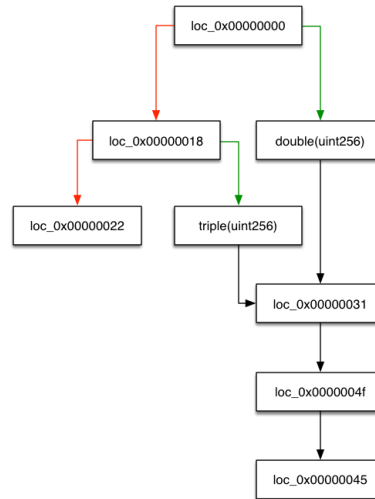


Figure 2: Emulated CFG

This helps us to translate such graph to the following pseudo like C code, as shown in listing 19.

```

1 hash = calldataload(0x0) / exp(0x2, 0xe0);
2 switch (hash) {
3     case 0xee97206: // double(uint256)
4         memory[0x60] = calldataload(0x4) * 2;
5         return memory[0x60];
6         break;
7     case 0xf40a049d: // triple(uint256)
8         memory[0x60] = calldataload(0x4) * 3;
9         return memory[0x60];
10        break;
11    default:
12        // STOP
13        break;
14 }

```

Listing 19: Static/Dynamic Graph Pseudo-C Code

As we can notice from the above pseudo code. Each runtime code has a dispatcher for each user-defined function. Once it is decompiled we get the following output shown in listing 20.

```
1 contract C {
2     function double(int arg_4) {
3         return arg_4 * 2;
4     }
5
6     function triple(int arg_4) {
7         return arg_4 * 3;
8     }
9 }
```

Listing 20: Decompiled Pseudo-C code

7 Code Analysis

7.1 Vulnerable Contract

Let's take a simple vulnerable smart contract such as the one shown in listing 21. The detailed analysis of the vulnerability has already been published by Abhiroop Sarkar in his blog and can be thoroughly read there.

7.1.1 Solidity source code

```
1 contract SendBalance {
2     mapping ( address => uint ) userBalances ;
3     bool withdrawn = false ;
4
5     function getBalance (address u) constant returns ( uint ){
6         return userBalances [u];
7     }
8
9     function addToBalance () {
10        userBalances[msg.sender] += msg.value ;
11    }
12
13    function withdrawBalance (){
14        if (!(msg.sender.call.value (
15            userBalances [msg . sender ]))()) { throw ; }
16        userBalances [msg.sender ] = 0;
17    }
18 }
```

Listing 21: Vulnerable Smart Contract

7.1.2 Runtime Bytecode

```
1 60606040526000357c01000000000000000000000000000000000000000000000000 \
2 00000000000000000000000000000000000000000000000000000000000000000000 \
3 578063c0e317fb1461005e578063f8b2cb4f1461006d576100 \
4 4d565b005b61005c6004805050610099565b005b61006b6004 \
5 80505061013e565b005b610083600480803590602001909190 \
6 505061017d565b604051808281526020019150506040518091 \
7 0390f35b3373fffffffffffffffffffffffffffffffffffffffff \
8 ff16600060005060003373fffffffffffffffffffffffffffffff \
9 ffffffffffff16815260200190815260200160002060005054 \
10 60405180905060006040518083038185876185025a03f19250 \
11 5050151561010657610002565b6000600060005060003373ff \
12 ffffffffffffffffffffffffffffffffffffffffffff168152602001 \
13 908152602001600020600050819055505b565b346000600050 \
14 60003373fffffffffffffffffffffffffffffffffffffffffff16 \
15 81526020019081526020016000206000828282505401925050 \
16 819055505b565b6000600060005060008373fffffffffffffff \
17 ffffffffffffffffffffffffffff168152602001908152602001 \
18 6000206000505490506101b6565b91905056
```

Listing 22: Vulnerable Smart Contract Runtime Bytecode

7.1.3 ABI Definition

```
1  [  
2  {  
3    "constant": false,  
4    "inputs": [],  
5    "name": "withdrawBalance",  
6    "outputs": [],  
7    "type": "function"  
8  },  
9  {  
10   "constant": false,  
11   "inputs": [],  
12   "name": "addToBalance",  
13   "outputs": [],  
14   "type": "function"  
15  },  
16  {  
17   "constant": true,  
18   "inputs": [  
19     {  
20       "name": "u",  
21       "type": "address"  
22     }  
23   ],  
24   "name": "getBalance",  
25   "outputs": [  
26     {  
27       "name": "",  
28       "type": "uint256"  
29     }  
30   ],  
31   "type": "function"  
32  }  
33  ]
```

Listing 23: Vulnerable Smart Contract ABI Definition

7.1.4 Decompiled version

```
1 function getBalance(address) {
2     return store[arg_4];
3 }
4
5 function addToBalance() {
6     store[msg.sender] = store[msg.sender];
7     return;
8 }
9
10 function withdrawBalance() {
11     if (msg.sender.call.value(store[msg.sender]))() {
12         store[msg.sender] = 0x0;
13     }
14 }
15
16 **L12 (D8193): Potential reentrant vulnerability found.**
```

Listing 24: Vulnerable Smart Contract Decompilation

8 Bugs

Keeping an eye on Solidity Compiler Bugs is one of the important notes one would consider.

8.1 Reentrant Vulnerability / Race Condition

Also known as the DAO vulnerability. similar to the `SendBalance` contract from above. In the meantime significant changes have been made to the EVM which includes the introduction of a `REVERT` instruction to restore a given state. An excerpt of the explanation is as follows:

call the function to execute a split before that withdrawal finishes. The function will start running without updating your balance, and the line we marked above as "the attacker wants to run more than once" will run more than once.

8.2 Call Stack Vulnerability

Call stack attack, explained by Least Authority[14] takes advantage of the fact that a `CALL` operation will fail if it causes the stack depth to exceed 1024 frames. Which happens to also be the current limit of the stack as previously described earlier. It will ultimately fail and not cause an exception. Unlike stack underflow which happens when frames are not present on the stack during the invocation of a specific instruction. This is a known problem that indicates an error instead of reverting back to the state to the caller. There are often a lack of assert checks in Solidity contracts, due to the poor support for actual unit testing. Given the special condition requiring to trigger this problem, which is an environment specific problem then we cannot easily spot it through static analysis. One potential mitigation would be for the EVM to implement integrity checks before executing a contract that would ensure the state of the stack, and the depth required by the contract (computed either dynamically or statically by the compiler) are met.

8.3 Time Dependence Vulnerability

`TIMESTAMP` returns the current blockchain timestamp and should not be used. As the timestamp of the block can be predicted or manipulated by the miner, which is something that the developers must keep in mind when implementing routines that depend on such variable. Because of this, developers must be extremely careful with time dependency. This was well explained by the case study from @mhswende with the Ethereum Roulette[12] that shows how an implementation of Ethereum Roulette was abused.

9 Future

As contracts are embedded in blockchain, there is no easy way to deploy updates to patch existing contracts like we would do with any regular software. This is an implementation limitation to understand. Regular softwares development has seen the integration and the raise of Security Development Lifecycle (SDL) as part of its development lifecycle, this is a process which has become increasingly popular that also includes models such as threat modeling which has yet to be seen within the smart-contract World regardless of the platform itself.

There is also a growing community that aims at raising awareness for writing secure solidity code, such as the "Underhanded Solidity Coding Contest" [15] announced early July for the first time that aims at judging code containing hidden vulnerabilities that can be interpreted as backdoors. Such vulnerabilities/backdoors that aren't obvious during the code auditing process, and can easily be misinterpreted and dismissed as coder error(s). USCC first contest is around the theme of Initial Coins Offering (ICOs), and includes Solidity Lead Developer, Christian Reitwiessner, in its jury. In addition of that, some forks such as Quorum [16] are rising interest by adding an privacy layer on top of the smart-contract blockchain, often required and currently missing with the actual Ethereum implementation.

10 Acknowledgments

- Mohamed Saher
- Halvar Flake
- DEFCON Review Board Team
- Max Vorobjov & Andrey Bazhan
- Gavin Wood
- Andreas Olofsson

References

- [1] Suiche, Matt. "Porosity: Ethereum Smart-Contract Decompiler" N.p., n.d. Web. <https://github.com/comaeio/porosity>
- [2] Woods, Gavin. "Ethereum: A Secure Decentralised Generalised Transaction Ledger." N.p., n.d. Web. <https://github.com/ethereum/yellowpaper>.
- [3] Olofsson, Andreas. "Solidity Workshop." N.p., n.d. Web. <https://github.com/androlo/solidity-workshop>.
- [4] Olofsson, Andreas. "Solidity Contracts." N.p., n.d. Web. <https://github.com/androlo/standard-contracts>.
- [5] Velner, Yarn, Jason Teutsch, and Loi Luu. "Smart Contracts Make Bitcoin Mining Pools Vulnerable." N.p., n.d. Web. <https://eprint.iacr.org/2017/230.pdf>.
- [6] Luu, Loi, Duc-Hiep Chu, Hrishi Olickel, Aquinas Hobor. "Making Smart Contracts Smarter." N.p., n.d. Web. <https://www.comp.nus.edu.sg/%7Ehobor/Publications/2016/Making%20Smart%20Contracts%20Smarter.pdf>.
- [7] Atzei, Nicola, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts." N.p., n.d. Web. <https://eprint.iacr.org/2016/1007.pdf>.
- [8] Sarkar, Abhiroop. "Understanding the Transactional Nature of Smart Contracts." N.p., n.d. Web. <https://abhiroop.github.io/Exceptions-and-Transactions>.
- [9] Siegel, David. "Understanding The DAO Attack." N.p., n.d. Web. <http://www.coindesk.com/understanding-dao-hack-journalists>.
- [10] Blockchain software for asset management. "OYENTE: An Analysis Tool for Smart Contracts." N.p., n.d. Web. <https://github.com/melonproject/oyente>.
- [11] Holst Swende, Martin. "Devcon1 and Ethereum Contract Security." N.p., n.d. Web. <http://martin.swende.se/blog/Devcon1-and-contract-security.html>.

- [12] Holst Swende, Martin. "Breaking the House", N.p.,n.d. Web. http://martin.swende.se/blog/Breaking_the_house.html.
- [13] Buterin, Vitalik. "Thinking About Smart Contract Security." N.p., n.d. Web. <https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security>.
- [14] Least Authority. "Gas Economics: Call Stack Depth Limit Errors." N.p., n.d. Web. <https://github.com/LeastAuthority/ethereum-analyses/blob/master/GasEcon.md#callstack-depth-limit-errors>.
- [15] Underhanded Solidity Coding Contest, Web. <http://u.solidity.cc/>.
- [16] Quorum. "A permissioned implementation of Ethereum supporting data privacy." <https://github.com/jpmorganchase/quorum>