EVM Bytecode and smart contract internals

Fedor Sakharov
fedor.sakharov@gmail.com

Agenda

- About me
- A bit about Ethereum
- Smart contracts and their bytecode
- Implementing EVM support in r2
- Future work

About me

- @m0nt3kk1 on twitter, @montekki on github

About me

- @m0nt3kk1 on twitter, @montekki on github
- Code mostly in C or Go, sys and net stuff

About me

- @m0nt3kk1 on twitter, @montekki on github
- Code mostly in C or Go, sys and net stuff
- @xvilka brought me to r2 ~5 years ago

About me

- @m0nt3kk1 on twitter, @montekki on github
- Code mostly in C or Go, sys and net stuff
- @xvilka brought me to r2 ~5 years ago
- Made a living working in infosec and crypto

About me

- @m0nt3kk1 on twitter, @montekki on github
- Code mostly in C or Go, sys and net stuff
- @xvilka brought me to r2 ~5 years ago
- Made a living working in infosec and crypto
- Now unemployed

A bit about Ethereum

Ethereum: known facts

- Distributed blockchain-based computing platform and operating system featuring smart contract functionality (wiki)

Ethereum: known facts

- Distributed blockchain-based computing platform and operating system featuring smart contract functionality (wiki)
- Often referred to as blockchain 2.0 (hype)

Ethereum: known facts

- Distributed blockchain-based computing platform and operating system featuring smart contract functionality (wiki)
- Often referred to as blockchain 2.0 (hype)
- p2p blockchain stuff + Turing-complete VM (reality)

Ethereum: known facts

- Distributed blockchain-based computing platform and operating system featuring smart contract functionality (wiki)
- Often referred to as blockchain 2.0 (hype)
- p2p blockchain stuff + Turing-complete VM (reality)
- Some money inside ~$30 000 000 000

Ethereum: known facts

- Distributed blockchain-based computing platform and operating system featuring smart contract functionality (wiki)
- Often referred to as blockchain 2.0 (hype)
- p2p blockchain stuff + Turing-complete VM (reality)
- Some money inside ~$30 000 000 000

Ethereum: known hacks

- Parity Freeze ~400 000 000 $ freezed
- Parity Multisig hack ~32 000 000 $ stolen
- DAO hack ~70 000 000 $ stolen

Smart contracts and their bytecode

Smart contracts

- Smart contracts are a digitized version of real-world contracts

Smart contracts

- Smart contracts are a digitized version of real-world contracts
- In Eth different SC languages exist

Smart contracts

- Smart contracts are a digitized version of real-world contracts
- In Eth different SC languages exist
- Every one of them is translated into EVM bytecode

Smart contracts

- Runtime environment is the Ethereum VM

Smart contracts

- Runtime environment is the Ethereum VM
- Specified in the Yellow Paper

Smart contracts

- Runtime environment is the Ethereum VM
- Specified in the Yellow Paper
- A stack-based machine

Smart contracts

- Runtime environment is the Ethereum VM
- Specified in the Yellow Paper
- A stack-based machine
- Code always runs as intended (no side effects)

Smart contracts

- Runtime environment is the Ethereum VM
- Specified in the Yellow Paper
- A stack-based machine
- Code always runs as intended (no side effects)
- Code always terminates

Smart contracts: example

```
pragma solidity ^0.4.0;

contract Example {

    uint a;

    function setA(uint b) public {

        a = b;

    }

}
```

- Solidity code has no single entry point
- Transactions to contract call it’s methods with some arguments
- Code is compiled to bytecode which is stored on-chain with it’s abi

Smart contracts: example

```
pragma solidity ^0.4.0;

contract Example {

    uint a;

    function setA(uint b) public {

        a = b;

    }

}
```

```
$ solc ./example.sol --abi -o ./out/

$ cat ./out/Example.abi | pjson
[{
    "constant": false,
    "inputs": [
      {
        "name": "b",
        "type": "uint256"
      }
    ],
    "name": "setA",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable"
    "type": "function"
  }]
```

Smart contracts: example

```
pragma solidity ^0.4.0;

contract Example {

    uint a;

    function setA(uint b) public {

        a = b;

    }

}
```

```
$ solc ./example.sol --bin-runtime -o
./out/

$ cat ./out/Example.bin-runtime

608060405260043610603f576000357c0100000
0000000000000000000000000000000000000
0000000000900463ffffffff168063ee919d5
0146044575b600080fd5b348015604f57600080
fd5b50606c6004803603810190808035906020
0190929190505050606e565b005b806000819055
50505600a165627a7a72305820c4b0ba922b4d
8b5902a3090f2e7e045ffcfe0c553ebe553c
decabfca42c50029
```

Smart contracts: example

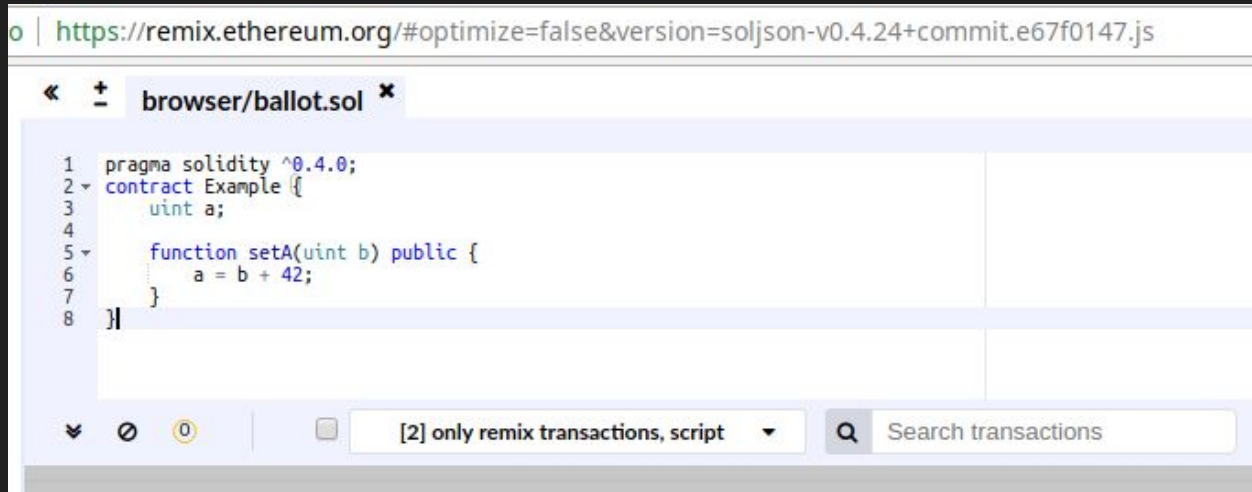
- Hex-encoded bytecode of the Smart Contract
- Execution starts from the first byte
- Input is stored in the transaction parameters

```
$ solc ./example.sol --bin-runtime -o
./out/

$ cat ./out/Example.bin-runtime

608060405260043610603f576000357c0100000
0000000000000000000000000000000000000
0000000000900463ffffffff168063ee919d5
0146044575b600080fd5b348015604f57600080
fd5b50606c60048036038101908080359060200
190929190505050606e565b005b806000819055
50505600a165627a7a72305820c4b0ba922b4d
8b5902a3090f2e7e045ffcfe0c553ebe553c
decabfca42c50029
```

Smart contracts: run & develop

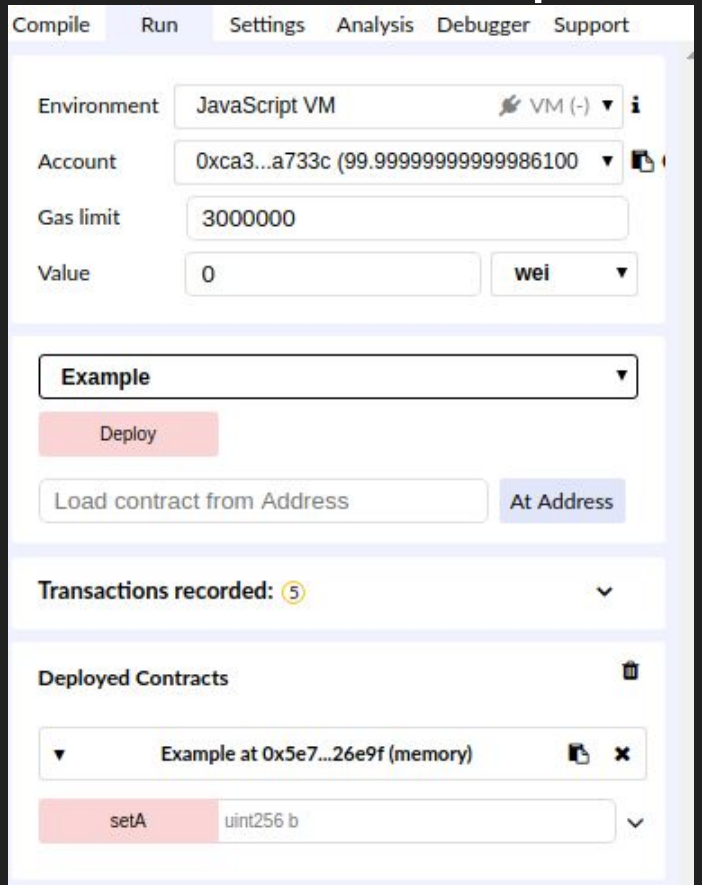
The widely-used IDE for Solidity is Remix (https://remix.ethereum.org)

- Editor
- Compiler
- Debugger
- Embedded VM



Smart contracts: run and develop

You have to deploy the contract and then you can call it’s methods



Smart contracts: run & develop

Calling setA method displays TX summarized:

[vm] **from:**0xca3...a733c **to:**Example.setA(uint256) 0x5e7...26e9f **value:**0 wei **data:**0xee9...0002a **logs:**0 **hash:**0x794...ac2fd Debug

status	0x1 Transaction mined and execution succeed
transaction hash	0x7945500403636dffa934de05968b50221b79230587c01683d8d44f16bf9ac2fd
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Example.setA(uint256) 0x5e72914535f202659083db3a02c984188fa26e9f
gas	3000000 gas
transaction cost	41675 gas
execution cost	20211 gas
hash	0x7945500403636dffa934de05968b50221b79230587c01683d8d44f16bf9ac2fd
input	0xee9...0002a
decoded input	{ "uint256 b": "42" }
decoded output	{}
logs	[]
value	0 wei

Smart contracts: run & devel

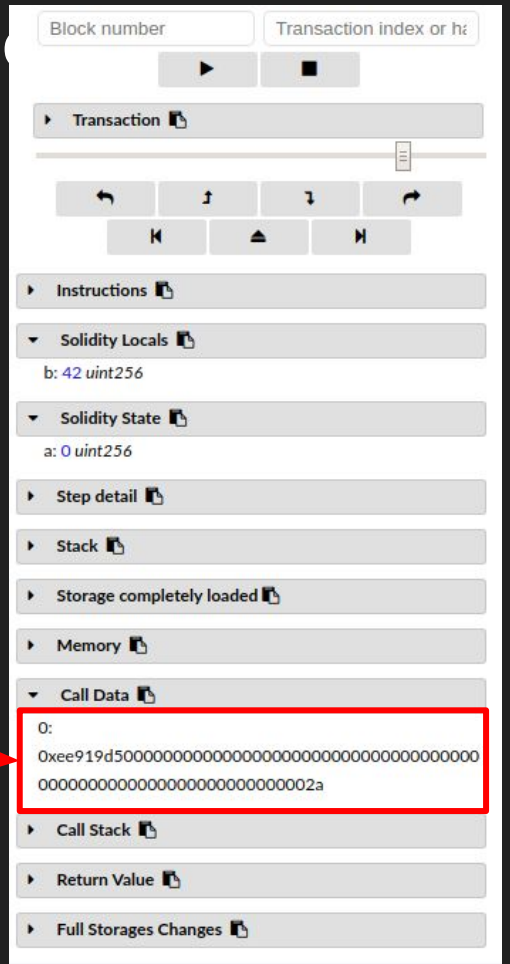
And if you go to

Debugging you may

see details about

The calldata for

instance:



EVM Bytecode

EVM bytecode

- Evm is a stack-based machine
- During runtime data is stored in memory or stack
- Data is stored permanently in contract’s storage on-chain

```
PUSH1 32      | Stack: [ 32 ]
PUSH1 42      | Stack: [ 42, 32 ]
ADD           | Stack: [ 74 ]
```

EVM bytecode

- Flow control is done pretty conventionally
- Beginnings of the basic blocks are JUMPDEST instructions

```
PUSH1 01          | Stack: [ 1 ], PC: 0
JUMPDEST          | Stack: [ 1 ], PC: 2
PUSH1 02          | Stack: [ 1, 2 ], PC: 3
ADD               | Stack: [ 3 ], PC: 5
PUSH1 0x2         | Stack: [ 3, 2 ], PC: 6
JUMP              | Stack: [ 3 ], PC: 7
PUSH1 02          | Stack: [ 3, 2 ], PC: 3
ADD               | Stack: [ 5 ], PC: 5
```

EVM bytecode

Two types of instructions:

1. Ones you expect to find in a stack-based VM
2. Ethereum-specific instructions:

   CREATE, CALL, CALLCODE, DELEGATECALL, REVERT, SUICIDE, etc.

EVM bytecode

The code in the contract is always executed from the 0x0000 addr

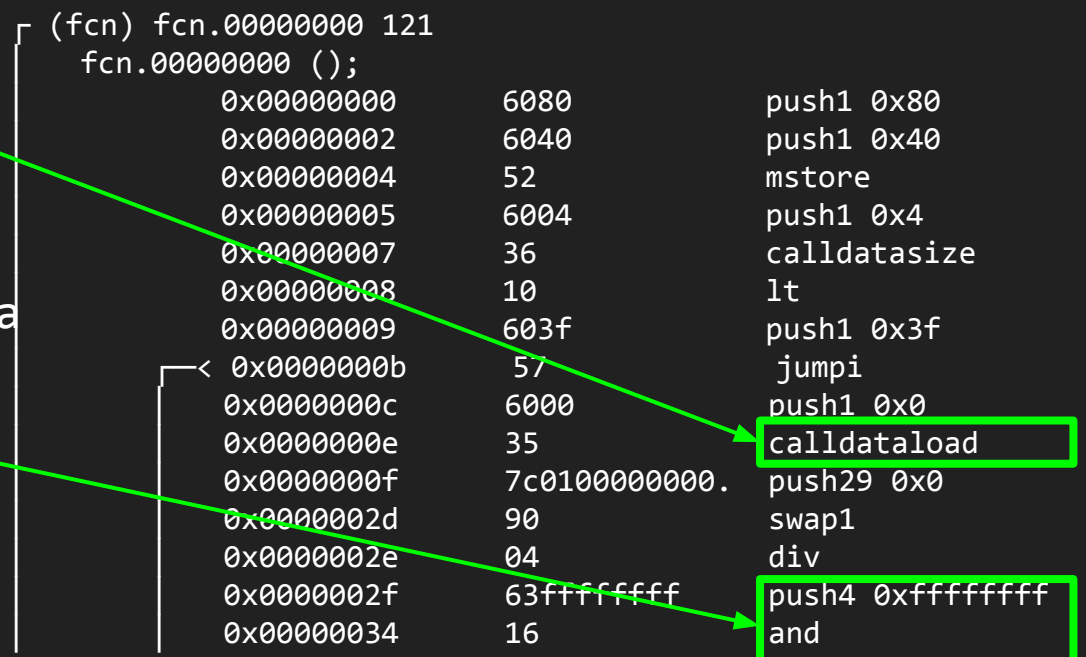
First instructions load TX data into VM memory and do other initialization

```
┌ (fcn) fcn.00000000 121
│   fcn.00000000 ();
│           0x00000000     6080           push1 0x80
│           0x00000002     6040           push1 0x40
│           0x00000004     52             mstore
│           0x00000005     6004           push1 0x4
│           0x00000007     36             calldatasize
│           0x00000008     10             lt
│           0x00000009     603f           push1 0x3f
│       ┌─< 0x0000000b      57             jumpi
│       │   0x0000000c     6000           push1 0x0
│       │   0x0000000e     35             calldataload
│       │   0x0000000f     7c0100000000.  push29 0x0
│       │   0x0000002d     90             swap1
│       │   0x0000002e     04             div
│       │   0x0000002f     63ffffffff     push4 0xffffffff
│       │   0x00000034     16             and
```

EVM bytecode

This code will load the data into memory

And it will AND this data with 0xffffffff thus taking the first four bytes of it



EVM bytecode

Those value is actually a hash signature of the function

A hash signature is the first four bytes of keccak hash of the function signature

```
    0x00000035      80              dup1
    0x00000036      63ee919d50      push4 0xee919d50
    0x0000003b      14              eq
    0x0000003c      6044            push1 0x44
┌─< 0x0000003e      57              jumpi
│ └> 0x0000003f     5b              jumpdest
│   0x00000040      6000            push1 0x0
│   0x00000042      80              dup1
│   0x00000043      fd              revert
└─> 0x00000044      5b              jumpdest
    0x00000045      34              callvalue
```

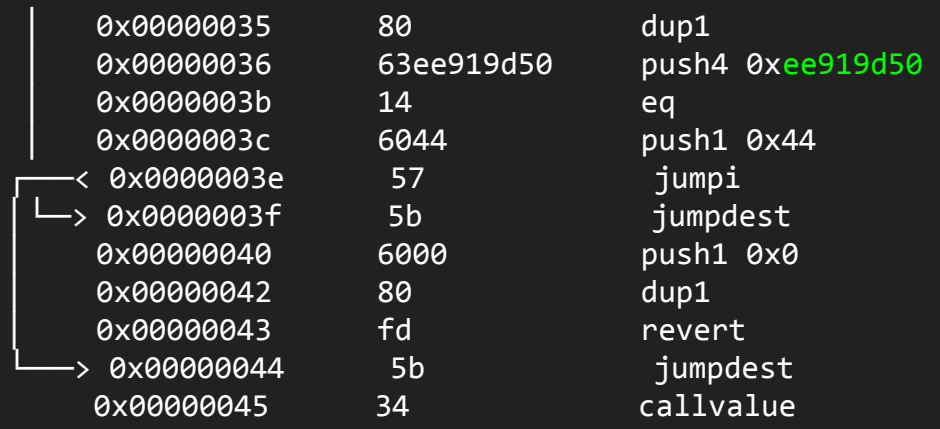
keccak256(“setA(uint256)”)

ee919d50445cd9f463621849366a537968fe1ce096894b0d0c001528383d4769

EVM bytecode

Those value is actually a hash signature of the function

A hash signature is the first four bytes of keccak hash of the function signature

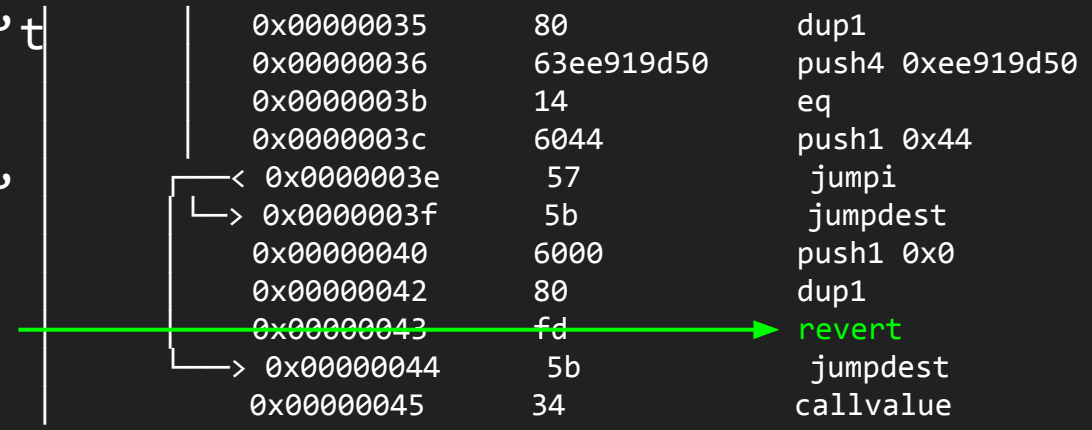


keccak256(“setA(uint256)”)

ee919d50445cd9f463621849366a537968fe1ce096894b0d0c001528383d4769

EVM bytecode

If the first four bytes of the input data doesn’t contain a hash known to the function dispatcher, the execution is terminated and the transaction is reverted



EVM bytecode

And after some other checks and init code we end up in our function

it adds 0x42 to an input parameter

and stores it if you recall

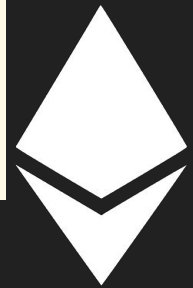
```
function setA(uint b) public {
     a = b + 0x42;
}
```

```
└─> 0x0000006e      5b          jumpdest
    0x0000006f      6042        push1 0x42
    0x00000071      81          dup2
    0x00000072      01          add
    0x00000073      6000        push1 0x0
    0x00000075      81          dup2
    0x00000076      90          swap1
    0x00000077      55          sstore
    0x00000078      50          pop
    0x00000079      50          pop
    0x0000007a      56          jump
    0x0000007b      00          stop
```

EVM bytecode: specification experience

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{pc}+1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if } x < \|I_{\mathbf{b}}\| \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0),\dots,c(\boldsymbol{x}_{\|x\|-1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮



Implementing EVM support in R2

ASM plugin

- The opcodes and the commands format is straightforward
- Opcodes are 1-byte
- Most of them have no operands
- Just follow the yellowpaper

ANAL plugin

- Things become more tricky
- All jumps take values from stack as operands
- In general case not possible to retrieve the dst addrs of the jump during the analysis
- If previous command is PUSH_*, take dst addr from it

ASM + ANAL: let’s take a look

```
theodor@fran:r2presentation$ rax2 -s < ./out/Example.bin-runtime > ./out/Example.bin-runtime.bin
theodor@fran:r2presentation$ r2 -a evm ./out/Example.bin-runtime.bin
 -- Trace register changes while debugging with 'e trace.cmtregs=true'
[0x00000000]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00000000]> pd 20
┌ (fcn) fcn.00000000 118
│   fcn.00000000 ();
│           0x00000000      6080            push1 0x80
│           0x00000002      6040            push1 0x40
│           0x00000004      52              mstore
│           0x00000005      6004            push1 0x4
│           0x00000007      36              calldatasize
│           0x00000008      10              lt
│           0x00000009      603f            push1 0x3f
│       ┌─< 0x0000000b      57              jumpi
│       │   0x0000000c      6000            push1 0x0
│       │   0x0000000e      35              calldataload
│       │   0x0000000f      7c01000000000.  push29 0x0
│       │   0x0000002d      90              swap1
│       │   0x0000002e      04              div
│       │   0x0000002f      63ffffffff      push4 0xffffffff
│       │   0x00000034      16              and
│       │   0x00000035      80              dup1
│       │   0x00000036      63ee919d50      push4 0xee919d50
│       │   0x0000003b      14              eq
│       │   0x0000003c      6044            push1 0x44
│       └─< 0x0000003e      57              jumpi
[0x00000000]>
```

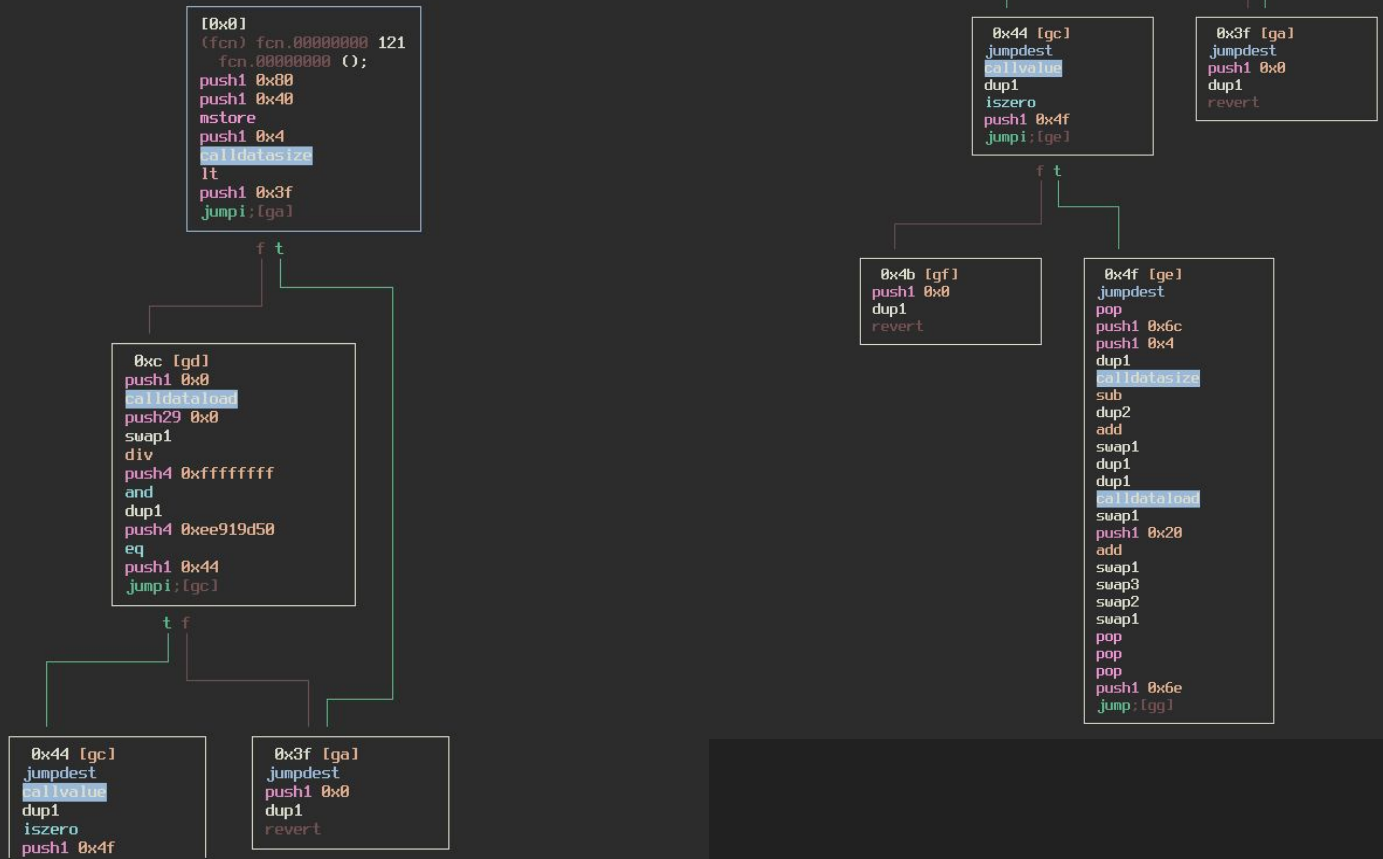
ASM + ANAL: let’s take a look

```
theodor@fran:r2presentation$ rax2 -s < ./out/Example.bin-runtime > ./out/Example.bin-runtime.bin
theodor@fran:r2presentation$ r2 -a evm ./out/Example.bin-runtime.bin
 -- Trace register changes while debugging with 'e trace.cmtregs=true'
[0x00000000]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00000000]> pd 20
┌ (fcn) fcn.00000000 118
│   fcn.00000000 ();
│           0x00000000      6080            push1 0x80
│           0x00000002      6040            push1 0x40
│           0x00000004      52              mstore
│           0x00000005      6004            push1 0x4
│           0x00000007      36              calldatasize
│           0x00000008      10              lt
│           0x00000009      603f            push1 0x3f
│       ┌─< 0x0000000b      57              jumpi
│       │   0x0000000c      6000            push1 0x0
│       │   0x0000000e      35              calldataload
│       │   0x0000000f      7c01000000000.  push29 0x0
│       │   0x0000002d      90              swap1
│       │   0x0000002e      04              div
│       │   0x0000002f      63ffffffff      push4 0xffffffff
│       │   0x00000034      16              and
│       │   0x00000035      80              dup1
│       │   0x00000036      63ee919d50      push4 0xee919d50
│       │   0x0000003b      14              eq
│       │   0x0000003c      6044            push1 0x44
│       └─< 0x0000003e      57              jumpi
[0x00000000]>
```

Well, at least it’s usable

However, the long-operand commands like push29 are not displayed correctly

ASM + ANAL: let’s take a look



IO plugin

- Contract lives on-chain, why not read it from there?

IO plugin

- Contract lives on-chain, why not read it from there?
- Go-ethereum exposes a set of RPC methods

IO plugin

- Contract lives on-chain, why not read it from there?
- Go-ethereum exposes a set of RPC methods, eth_GetCode, for instance

```
curl -X POST --data
'{"jsonrpc":"2.0","method":"eth_getCode","params":["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
"0x2"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result":
"0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f25b600060078202905091905056"
}
```

IO plugin

- Contract lives on-chain, why not read it from there?
- Go-ethereum exposes a set of RPC methods, eth_GetCode, for instance
- Bring in libcurl and a json parser to the plugin

IO plugin

- Contract lives on-chain, why not read it from there?
- Go-ethereum exposes a set of RPC methods, eth_GetCode, for instance
- Bring in libcurl and a json parser to the plugin
- Now we can talk to the node

IO plugin

```
$ r2 -a evm "evm://localhost:8545@0x075121e8f930cb7bc21cc726600e532b3e60e7d1"
= attach 1 1
 -- In Soviet Russia, radare2 has documentation.
[0x00000000]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00000000]> pd 10
|   fcn.eax ();
|           0x00000000      6080          push1 0x80
|           0x00000002      6040          push1 0x40
|           0x00000004      52            mstore
|           0x00000005      6004          push1 0x4
|           0x00000007      36            calldatasize
|           0x00000008      10            lt
|           0x00000009      603f          push1 0x3f
|       ┌─< 0x0000000b      57             jumpi
|       │   0x0000000c      6000          push1 0x0
|       │   0x0000000e      35            calldataload
```

DBG plugin

- EVM has no debugging std debugging interfaces

DBG plugin

- EVM has no debugging std debugging interfaces
- You can not trace the live execution of a call to smart contract

DBG plugin

- EVM has no debugging std debugging interfaces
- One can not trace the live execution of a call to smart contract
- However, RPC API allows to get a trace of any transaction

```
{"method": "debug_traceTransaction", "params": [txHash, {}]}
```

DBG Plugin

{"method": "debug_traceTransaction", "params": [txHash, {}]}

Will return a step-by-step trace of a transaction in JSON form

```
 ...
{
  gas: 85301,
  returnValue: "",
  structLogs: [{
      depth: 1,
      error: "",
      gas: 162106,
      gasCost: 3,
      memory: null,
      op: "PUSH1",
      pc: 0,
      stack: [],
      storage: {}
  },
...
```

DBG Plugin

- Read and parse the whole transaction trace
- Implement DBG plugin API
- Emulate “live” execution, actually seeking forward in the trace
- Breakpoints are just checking if the current element in the trace equals bpt addr
- And so on

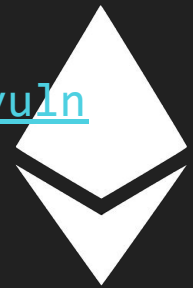
Future work

Future work

- ESIL looks promising, since we can emulate the finite executions well
- R2dec support also, the code looks simple to decompile
- Support for displaying 32-byte operands
- Better JMP/CALL/RET dst computation

Useful links

- https://medium.com/@theo_montekki my blog with written info
- A series of blog posts on blog.zeppelin.solutions
- CTF https://ethernaut.zeppelin.solutions/
- https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI
- http://yellowpaper.io/
- https://hackernoon.com/smart-contract-security-part-1-reentrancy-attacks-ddb3b2429302
- https://applicature.com/blog/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes



Thank you!

Questions?