View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open Repository and Bibliography - Luxembourg

Visual Emulation for Ethereum's Virtual Machine

Robert Norvill
Sedan Group, SnT
University of Luxembourg
29 Av. John F. Kennedy
L-1855 Luxembourg
robert.norvill at uni.lu

Beltran Borja Fiz Pontiveros
Sedan Group, SnT
University of Luxembourg
29 Av. John F. Kennedy
L-1855 Luxembourg
beltran.fiz at uni.lu

Radu State
Sedan Group, SnT
University of Luxembourg
29 Av. John F. Kennedy
L-1855 Luxembourg
radu.state at uni.lu

Andrea Cullen
Engineering and Informatics
University of Bradford
Richmond Rd
Bradford, BD7 1DP, UK
a.j.cullen at bradford.ac.uk

***Abstract*—In this work we present E-EVM, a tool that emulates and visualises the execution of smart contracts on the Ethereum Virtual Machine.**

By working with the readily available bytecode of smart contracts we are able to display the program's control flow graph, opcodes and stack for each step of contract execution.

This tool is designed to aid the user's understanding of the Etheruem Virtual Machine as well as aid the analysis of any given smart contract. As such, it functions as both an analysis and a learning tool. It allows the user to view the code in each block of a smart contract and follow possible control flow branches. It is able to detect loops and suggest optimisation candidates. It is possible to step through a contract one opcode at a time.

E-EVM achieved an average of 85.6% code coverage when tested.

***Index Terms*—Ethereum, Smart Contracts, Visualisation, Bytecode**

I. Introduction

Ethereum [1] is one of the world's most popular cryptocurrency systems. It is set apart by its heavy integration and usage of smart contracts. Smart contracts are programs that are stored on the blockchain and run when a user sends a transaction to them. They benefit from the immutability and consensus assurances provided by blockchain. Ethereum smart contracts are most commonly complied from source code, the compiled bytecode is executed by the Ethereum Virtual Machine (EVM).

Ethereum has been subject to a number of high profile attacks and incidents that exploited flaws in smart contracts like the DOA hack [2] and the Parity wallets frozen accounts problem [3], together having cost hundreds of millions of dollars. These attacks were caused by errors made by the developers who wrote the contracts. This highlights the need for developers and users to have tools with which to analyse the code and behaviour of contracts in order to avoid potentially dangerous bugs before putting their money into them.

Focusing on bytecode is of particular value. Only bytecode is stored on the blockchain and availability of source code is entirely at the discretion of the developer. Some third party tools, like `etherscan.io` [4], offer verification services for contracts. However, this service relies on the developer uploading their source code to the site, as well as trust in the third party site. In the vast majority of cases only the bytecode is available to the user. At the time of writing Etherscan lists > 999999 contract accounts and only 10280 verified contract accounts. E-EVM allows for insight into what a compiled contract does by following all possible execution paths.

978-1-5386-3416-/18/$31.00 © 2018 IEEE

Ethereum smart contracts can be compiled from several languages. While Solidity has come to be something of a standard, the Serpent [5] and Viper [6] projects are maintained and have their own unique purposes. Focusing on bytecode allows us to be language-agnostic.

A. *Ethereum Virtual Machine*

The EVM is responsible for the execution of smart contracts. It is a simple stack machine that reads bytecode compiled from a higher level language. The stack has a maximum depth of 1024 and is held in memory. Ethereum's bytecode is Turing complete by design.

A compiled smart contract consists of a sequence of opcodes and data, which form the instructions for the EVM and the static data values to be used during execution. The EVM reads bytecode by incrementing a program counter, all Ethereum's opcodes are 1 byte long.

Certain opcodes, including the `PUSH`, `JUMP` and `JUMPI` instructions, require that the pointer is incremented by more than 1 and have certain constraints. The jump instructions must land on an instance of the `JUMPDEST` opcode. Landing on any other opcode is considered to be an exception and causes execution of the contract to fail.

In this paper we present E-EVM, a prototype tool capable of operating on Ethereum bytecode that emulates the operations of the EVM on smart contract bytecode and visualises the control flow between blocks of code within a contract as a directed graph. The user can step through and see what values are on the stack during execution and which opcodes are responsible for stack manipulation. Two versions of E-EVM are provided, one focusing on suggesting optimisation candidates and the other on handling operations.

E-EVM uses visualisation of a contract's low level behaviour to allow the user to gain a deeper understanding of how a given compiled contract operates. E-EVM can be used to aid code analysis and optimisation as well as to understand the operation of the EVM and any given smart contract.

II. Related Work

Ethereum's online IDE, Remix [7] has added a debugger to its interface; it appears this tool is intended to allow the

user to step through the execution of a given contract. At the time of writing the debugger is in the alpha stage and we have been unable to view it working correctly. We cannot, therefore, comment on its performance or features.

The solgraph project [8] claims to be able to produce Control Flow Graphs for Ethereum smart contract bytecode.

E-EVM's ability to suggest candidates for constant synthesis was inspired by the Souper super optimiser [9] which can detect this optimisation for other forms of bytecode. It is the authors' intention that our tool can be used to better understand how optimisation might be applied to smart contracts.

In [10] a symbolic execution tool is presented which is capable of using formal methods to detect bugs in the Ethereum smart contracts. The interest in the tool suggests a strong need for tools capable of smart contract analysis.

Chen et al. present a tool for finding patterns which identify candidates for optimisation to reduce the monetary cost of smart contract execution [11]. This is another effort to analyse smart contracts, this time in terms of cost rather than security.

In our previous work we made heavy use of contract bytecode in order to ascertain contract purpose [12].

III. High level overview of E-EVM

This project is split into two main parts; the emulation engine which forms the back-end, and the JavaScript interface which forms the front-end. The engine is written in Python 3 and emulates code execution in the EVM. We provide two version of the E-EVM engine.

The first is called Sym, it does not carry out any mathematical or logical operations of opcodes. Instead it pushes symbolic variables as the results of operations. A symbolic variable is taken to be one of an unknown value. When encountered by `JUMPI` as the jump condition it is taken to resolve to both true and false, both execution paths are followed. Sym's only real values come from `PUSH` operations. It is able to suggest candidates for constant synthesis within a block: An optimisation whereby an operation that always returns the same value can be replaced with a constant of that value.

The second is called Concrete, it carries out basic arithmetic operations and pushes the results to the stack. This can only be done if both inputs are concrete values, otherwise it pushes a symbolic variable to the stack. Concrete is able to provide a more accurate representation of contract execution as more of its stack values can be real.

The output from the E-EVM's back-end is a list of the opcodes that make up the contract passed to it, a list of stack states pertaining to each opcode, the code block that each opcode is in, the call history and the CFG generated from the emulated execution of the contract. There is a separation between front and back-end, with the front-end only having to deal with static, pre-generated information. Our front-end dashboard is written in JavaScript and presented in a HTML page. It takes the data outputted by the engine and represents it for the user to step through.

This design allows for different front-ends to be used to represent the data without having to alter the operation of the back-end. As this project is open source [13] any interested reader can process the results of E-EVM's emulation in a custom visualisation.

To the best of our knowledge there does not currently exist a tool for Ethereum that provides the ability to step through execution paths by nodes in a CFG or by opcode as well as view the state of the stack and opcode responsible for each stack operation in turn.

IV. E-EVM Operational Behaviour

A. Mapping Jump Destinations & blocks

Once bytecode has been obtained and formatted a first pass over the contract is carried out by the `resolve_jump_dests` function, it is called by the `sym_ex` function which is responsible for the bulk of emulation related operations.

Ethereum's `PUSH` instructions are numbered from 1 to 32. This number indicates the number of bytes that the EVM should take as the value to be added to the stack. The program counter (`PC`) needs to move over the bytes that are to be taken as a value and pushed to the stack, therefore: `PC'=PC+1+n for PUSHn` where `n` is the integer appended to the `PUSH` operation in range: [1;32].

The `resolve_jump_dests` function records the code block that each opcode in the contract code is in. We define a block as starting from a `JUMPDEST` and continuing until a second one is encountered. As an Ethereum smart contract can only jump to an instance of `JUMPDEST`, this opcode logically defines blocks with distinct function between which control is passed. The blocks are used to model control flow with each block represented as a node in the graph. Jump/i instructions and jump destinations generate directed edges between the nodes.

When the `JUMPDEST` instruction is encountered E-EVM checks if the previous opcode was a jump instruction and if not it adds an edge between the block containing `i` (itself), and the previous block containing the opcode `i-1`. Every encounter with the `JUMPDEST` instruction is a passing of control considered to be a transition in control.

B. Emulated Execution

Once jump destinations and blocks have been mapped the program calls the function `sym_ex`, which uses a list to represent the stack. When the Sym version encounters an opcode that returns a variable a symbolic variable with the format `%vart` where `t` is a value incremented by 1 for each symbolic variable created, is added to the stack.

Upon encountering a jump instruction E-EVM checks that the destination is valid and adds an edge to the CFG from the node representing the previous block to the node representing the block containing the jump destination. In the case of an invalid destination the `JUMP` instruction halts execution and `JUMPI` does not follow the one of its two paths which points to the jump destination. No edges are added for jump instructions that end a path. Invalid jump destinations include symbolic variables, which cannot be resolved. Execution halts upon

encountering any of the opcodes that would cause contract execution to halt in the EVM. Fig. 1 shows the result of encountering a `JUMPI` instruction in bytecode. It is possible that `i+1` results in landing in some block other than: `x` if `i+1` is a `JUMPDEST` as seen in fig. 2.

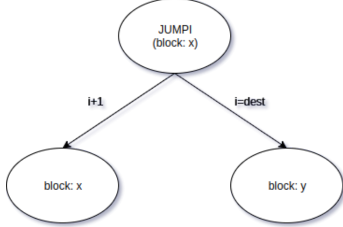


Fig. 1: Execution of `JUMPI`

Both versions of E-EVM detect re-entry into a block, this stops E-EVM getting stuck in an infinite loop when executing both branches of a conditional jump. It aids code analysis and user insight into the operations of Ethereum smart contracts. Re-entry is defined as arriving at the same block from the same previous destination more than once.

Fig. 2: Greeter contract: lines 24-27

```
PUSH2 0x66
JUMPI
JUMPDEST
PUSH1 0x0
```

C. Call History Tracking

As the program tracks the state of the stack for each opcode in a contract it is also necessary to track the call history for each opcode. The call history is defined as the previous blocks through which control has been passed before reaching the current opcode. A block can be encountered more than once, when control has passed through different blocks before arrived back at a given block it is possible that the contents of the stack might be different. This can be seen in the example in fig. 3 where the rounded boxes represent code blocks and the straight boxes represent stack states. The block starting at the jump destination located at 0x23 in the bytecode can receive two different stacks depending on which block passes control to it. In our example the difference in stack affects the outcome of the less than check.

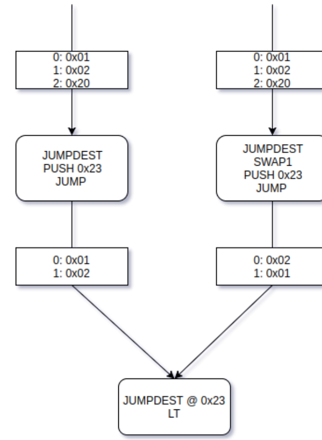


Fig. 3: Call History Example

As our aim is to visualise an accurate emulation of the EVM, the stack state displayed to the user is dependent on the history of calls made before entering the current block.

D. Visualisation

Once the graph has been built during symbolic execution it is given as output for use by our front-end, which is used to generate the view seen by the user. The contract's CFG as well each opcode and its corresponding stack state are displayed using JavaScript in a HTML page. The user can then step backwards and forwards through the program's execution as they wish. When a `JUMPI` is reached in the visualisation the user can choose which branch they wish to follow. An example can be seen in fig. 4, the blue nodes indicate the blocks visited prior to the current block.

E. Testing

We tested our program by running it on various contracts and measuring the code coverage we achieved. Code coverage was measured as a percentage for each contract by calculating:

```
nodes / blocks * 100
```

Where nodes is the number of nodes in the CFG blocks is the total number of blocks the contract has.

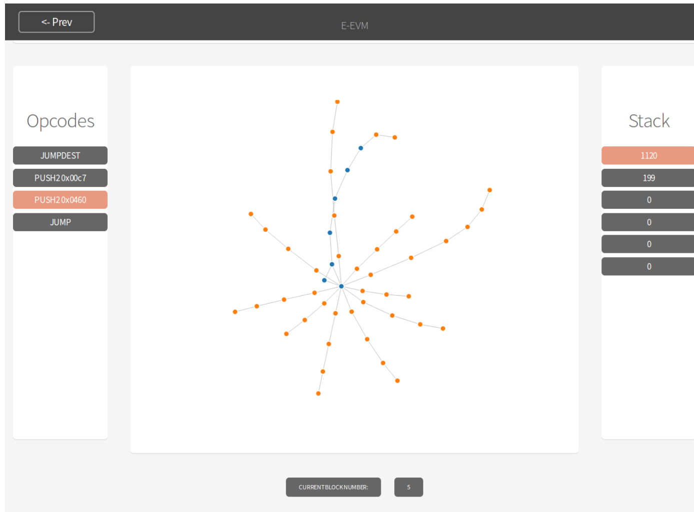
V. Results

Various metrics for each contract used for testing can be seen in table I. The results pairs are given in the format (Sym/-Concrete) for the two versions E-EVM, constant synthesis only applies to Sym.

TABLE I: Metrics for tested contracts in the format (Sym/-Concrete)

	Greeter	Wallet	Token	Golem	Raiden
Coverage%	100/100	68.6/68.6	78.8/78.8	90.8/90.8	89.6/89.6
Unkwn ops	0/0	0/0	15/15	0/0	0/0
Dyn. jmps.	0/0	0/0	5/5	0/0	0/0
Bad jmps.	0/0	15/15	0/0	0/0	62/62
Loops	8/8	78/78	19/19	89/89	82/82
Re-entries	4/4	31/31	8/8	37/37	44/44
Cnst. synth.	3	51	40	112	93

Fig. 4: Emulator GUI



The first contract we tested our program with was the greeter contract provided as an example by Ethereum, it can be found at [14].

The second contract we tested is a wallet contract found at the address 0xf1ce0A98eFbFA3f8EbEC2399847b7D88294A634e. We ran the contract creation code available from Etherscan.

The third contract is the token example found at [15]. We used the contract creation code available from Remix.

The fourth contract we tested against was the Golem MultiSig Wallet, found at the address 0x7da82C7AB4771ff031b66538D2fB9b0B047f6CF9. We used the contract creation code available from Etherscan.

The fifth and final contract is the Raiden MultiSig wallet found at 0x00C7122633A4EF0BC72f7D02456EE2B11E97561e. Interestingly, it has a similar functionality to the Golem wallet but the bad jump figures are very different.

The two versions are identical across the board, as such it is apparent that none of the contracts we use rely on the results of operations where the data comes exclusively from `PUSH` instructions. The results from Concrete would begin to diverge if historical data was used, as it would be able to resolve more variables to real (as oppose to symbolic) values.

Contracts sometimes contain unreachable code. For example the last 13 opcodes of the runtime code in the greeter example contract as compiled can be seen in 5. As there is no `JUMPDEST` after the `STOP` these opcode cannot be reached.

Fig. 5: Greeter contract: lines 342..354

```
342  STOP
343  LOG1
..
354  PUSH22  0x1A992D0029000000000000000 ...
```

VI. Conclusions

In this paper we present a tool capable of accurately emulating Ethereum Smart contracts with an average of 88.2% code coverage in our testing contracts. Our tool is capable of generating a visualisation to show contract behaviour as well as some of the inner workings of the EVM; stack, opcodes and `PC`.

Our program is capable of generating its visualisation for arbitrary contracts including popular and lengthy MultiSig contracts.

VII. Future Work

Future work includes improving Concrete to handle more operations with a view to aiding detection of dead code, and modelling Ethereum's storage to further increase the number of real values in use. Adding further information to the front-end visualisation would allow for the tool to be used more readily for debugging purposes. The addition of user-defined inputs which would allow for the emulated execution of historical contracts.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[2] D. Siegel. (2016) Understanding the dao hack. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists/

[3] R. R. O'Leary. (2017) Ico funds among millions frozen in parity wallets. [Online]. Available: https://www.coindesk.com/ico-funds-among-millions-frozen-parity-wallets/

[4] Etherscan. (2018) Ethereum blockchain explorer and search. [Online]. Available: https://etherscan.io/

[5] Ethereum-Wiki. (2017) Serpent.ethereum/wiki wiki. [Online]. Available: https://github.com/ethereum/wiki/wiki/Serpent

[6] ethereum. (2018) ethereum/vyper: New experimental programming language. [Online]. Available: https://github.com/ethereum/vyper

[7] Ethereum. (2017). [Online]. Available: https://remix.ethereum.org

[8] R. Revere. (2016) solgraph. [Online]. Available: https://github.com/raineorshine/solgraph

[9] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

[10] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[11] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 442–446.

[12] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in ethereum," in *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE, 2017, pp. 1–6.

[13] pisocrob. (2018) E-evm. [Online]. Available: https://github.com/pisocrob/E-EVM

[14] Ethereum. (2017) Create a hello world contract in ethereum. [Online]. Available: https://ethereum.org/greeter

[15] ——. (2017) Create a cryptocurrency contract in ethereum. [Online]. Available: https://www.ethereum.org/token