Towards a Formally Verified EVM in Production Environment

Xiyue Zhang[1], Yi Li[1], and Meng Sun[1]

School of Mathematical Sciences, Peking University
`{xiyuezhang, liyi_math, sunm}@pku.edu.cn`

Abstract. Among dozens of decentralized computing platforms, Ethereum attracts widespread attention for its native support of smart contracts by means of a virtual machine called EVM. Programs can be developed in various front-end languages. For example, Solidity can be deployed to the blockchain in the form of compiled EVM opcodes. However, such flexibility leads to critical safety challenges. In this paper, we formally define the behavior of EVM in Why3, a platform for deductive program verification, which facilitates the verification of different properties. Furthermore, the extracted implementation in OCaml can be directly integrated into the production environment and tested against the standard test suite. The combination of proofs and testing in our framework serves as a powerful analysis basis for EVM and smart contracts.

Keywords: EVM · Why3 · Verification · Testing.

1 Introduction

Ever since the inception of the Bitcoin blockchain system [12], cryptocurrencies have become a well-known global revolutionary phenomenon. Meanwhile, the decentralized blockchain system with no server or central authority, which emerges as a side product of Bitcoin and provides a continuously growing ledger of transactions being represented as a chained list of blocks distributed and maintained over a peer-to-peer network [16], shows great potential in carrying out secure online transactions. From then on, there have been a lot of changes and growth on the blockchain technology. Ethereum [4] extends Bitcoin's design, which can process not only transactions but also complex programs and *smart contracts*. Smart contracts can automatically run inside a blockchain, making it possible to use blockchain techniques in many other application domains besides cryptocurrencies, and has attracted a lot of attention from government, finance, health, entertainment and industry. This feature makes Ethereum a popular ecosystems for building blockchain-applications, which gains much more interest to innovate the options to utilize blockchain.

Smart contracts are often written in a Turing-complete programming language called *Solidity* [3] and then compiled into low-level machine instructions (*opcodes*), which can be encoded into bytecode. Ethereum Virtual Machine

(EVM) is a quasi-Turing complete machine which implements the execution model of the Ethereum. Given a sequence of bytecode instructions, which are compiled from smart contracts by EVM compiler, and the environment data, this execution model specifies how the blockchain transits from one state to another.

However, EVM and smart contracts are faced with several security vulnerabilities. In fact, a lot of attacks on smart contracts succeeded in the past years, such as the famous DAO (Decentralized Autonomous Organization) attack in 2016 resulting in the lost of more than 60M USD, and the Parity Wallet Hacks in which vulnerabilities in the wallet was exploited by an attacker to successfully steal over 150,000 ETH (about 30M USD), and leads to more than 250M USD frozen later in 2017. A taxonomy of vulnerabilities and related attacks against Solidity, the EVM, and the blockchain is presented in [1].

Formal verification techniques such as theorem proving and testing have become essential to guarantee the correctness of systems. The adoption of formal methods can facilitate the production of trustworthy and reliable smart contracts as well. To deal with the security challenges against EVM, we present a formal framework of generating verified EVM for production environment in this paper. The contributions of this work are:

– A formal definition of EVM specified in WhyML, the programming and specification language used in Why3 [7].
– An implementation of EVM in OCaml generated through an extraction mechanism based on a series of customized drivers.
– The verification of sample properties and testing of the OCaml implementation for EVM against a standard test suite for Ethereum.

This paper is organized as follows: We outline the framework for formalizing, property verifying and testing of EVM in Section 2. Section 3 presents some related work. Finally, we summarized this paper and point out the future research direction in Section 4.

2 EVM Framework

In this section, we present the EVM framework in detail. The main idea is to combine verification and testing towards developing secure EVM implementations. It also provides the potential to verify smart contracts at the bytecode level. This framework is mainly comprised of two parts: EVM specification in Why3 and experimental testing based on OCaml Extraction and Rust connection. This approach leverages on the formal methods and engineering, allowing us to perform both rigorous verification and efficient testing for EVM implementations and further smart contracts.

2.1 EVM in Why3

The first phase of the framework is to define a formal specification of EVM in Why3 and provide a platform for rigorous verification. We develop the EVM specification, following the Ethereum project yellow paper [15]. More specifically, the EVM implementation is translated into WhyML, the programming and specification language of Why3. Verification conditions can be further generated through Why3 based on the EVM and property specification. The verification goals can be split into a set of subgoals or directly be proved through the supported Satisfiability Modulo Theories (SMT) solvers. In cases when the automatic SMT solvers cannot deal with, users can resort to interactive theorem provers based on the files generated by Why3.

EVM is essentially a simple stack-based machine. The memory model of EVM is a word-addressed byte array, which is volatile. The storage model of EVM is a non-volatile word-addressed word array. The above three components form the infrastructure of EVM. Based on the formalization of the infrastructure, the most important is to capture the execution result of the EVM instructions. The perspective from which we deal with the execution process of a sequence of opcodes (instructions) is as a state transition process. This process starts with an initial state and leads to a series of changes in the stack, memory etc. The formalization of base infrastructure and the instruction set are specified through *Type Definition* and *Instruction Definition*, respectively. *Interpreter Definition* provides the specification of the interpreter and auxiliary functions.

Type Definition To formalize the infrastructure of EVM, we first need to provide the formalization of some commonly-used types in EVM, such as the types of machine word and the address in EVM. Hence, we developed a series of type modules such as `UInt256` and `UInt160` to ease the representation of corresponding types in EVM. We also use type alias supported by Why3 to make the basic formalization more readable and consistent with the original definition.

To this end, the components of the base infrastructure can be specified. Stack is defined as a list of elements whose type is `uint256`, aliased by `machine word`. Memory is defined as a function that maps `machine word` to an option type `option memory_content`. Since the memory content could be accessed through 256-bit store and 8-bit store instructions, `memory_content` is defined as an enumeration type including `Item8` and `Item256`. Similarly, Storage is defined as a function that maps `machine word` to `machine word`. To reflect the implicit change of the machine state, we defined more miscellaneous types. For example, we use `vmstatus`, `error` and `return_type` to capture the virtual machine status, the operation error, and the view of the returned result. The record type `machine_state` is defined to represent the overall machine state which consists of stack, memory, program counter, vmstatus, the instruction list, etc.

Instruction Definition The infrastructure has been built to specify the state of the virtual machine. Inspired by the instruction formalization in Lem [9], we defined the instruction set in multiple groups, such as arithmetic operations and stack operations, then integrated them into a summarized type definition `instruction`. Different subsets of instructions are wrapped up to form the complete specification in the definition of `instruction`.

The organization of the instruction category is a bit different from the yellow paper [15]. The comparison operations are grouped in the arithmetic operation set `Arith`; and the comparison operations for signed arithmetic are included in the signed arithmetic operation set `Sarith`. The bitwise operations and operation `BYTE` are categorized into the bit-related instruction group `Bits`. The information related instructions including environmental and block information ones are defined in type `info_inst`, except `CALL` and `CODE` instructions, such as `CALLDATACOPY`, `CODECOPY`, `CALLDATALOAD` and etc. These instructions are more closely related to memory and stack status. Therefore, extra instructions but closely-related ones are added to the original memory and stack instruction groups presented in the yellow paper. In case when some illegal command occurs, the instruction `Invalid` is included in the `instruction` definition. The `STOP` operation and the system operations from the yellow paper are defined in the `System` instruction set. The specification of the remaining instruction groups are basically the same as the corresponding instruction subsets in [15].

Interpreter Definition The specification of `interpreter` formalizes the state transition result of different instructions. For a specific instruction, the interpreter determines the resultant machine status developing from the current status. To make the definition of the interpreter more concise and compact, we defined some auxiliary functions. For example, the following function is used to obtain the next instruction to be executed. It is obtained from the instruction list following the program counter.

```
let get_inst (mac_st: machine_state): option instruction =
  match mac_st.mac_pc, mac_st.mac_insts with
  | pc, insts -> (nth pc insts)
  end
```

Pop and push operations are most commonly-used manipulations for the state transition of stack. Auxiliary functions `push_stack` and `pop_stack` are defined to control the change of stack status. For capturing the status transition result of `Swap` instructions, recursive functions `fetch` and `drop` are defined to support more direct manipulation on lists. Function `swap_stack` is defined further for state transition of swap instructions. With the pre-defined auxiliary functions, the definition of the interpreter function is essentially comprised of machine status update with regard to concrete instructions.

2.2 Running EVM in Production Environment

In this section, we introduce the second phase of the framework: deploy the verified EVM, encoded in Why3, in production environments. The deployment is essentially based on a co-compilation framework between OCaml and Rust.

OCaml is a functional programming language where the programs can be compiled either to OCaml bytecode or executable binaries. In the latter case, a runtime library is attached to handle various non-functional features like memory management. Through the `Callback` module, OCaml allows us to register a set of callback functions which can be used as an interface in static or dynamic linking. So we first defined a series of customized drivers to extract WhyML programs into Ocaml programs. After the extraction, the OCaml implementation of EVM is then encapsulated and exposed as a static linked library. This library can be further called by the EVM host in Rust.

The framework provides the interaction mechanism between Rust, the productive programming language, and Why3 the formal specification language. By gluing them together, verified models can be directly executed in production environments for further testing.

3 Related Work

Research interest of blockchain technology has exploded since the inception of Bitcoin. As the popularity of the second generation of blockchain, Ethereum, grows, a series of security vulnerabilities have also appeared. Since EVM and smart contracts deal directly with the transactions of valuable c ryptocurrency units among multiple parties, the security of smart contract programs and EVM implementations is of paramount importance. To address the security challenges, researchers resorted to the techniques of formal methods and program analysis.

– **Formalization Foundation of EVM** An executable formal semantics of EVM has been created in K framework by Everett et al. [8]. Hirai [9] proposed an EVM implementation in Lem, a language that can be compiled for a few interactive theorem provers. Further, safety properties of smart contracts can be proved in proof assistants like Isabelle/HOL. The `hevm` project [13] is implemented in Haskell for unit testing and debugging smart contracts, though the EVM implementation is not completed yet. Besides the above formalizations of EVM, there are also implementations of EVM in Javascript [10], Go [6], Ruby [5] and etc.
– **Verification of Smart Contracts** Sergey et al. [14] provided a new perspective between smart contracts and concurrent objects, based on which existing tools and insights for understanding, debugging and verifying concurrent objects can be used on smart contract behaviors. In [11], several new security problems were pointed out and a way to enhance the operational semantics of Ethereum was proposed to make smart contracts less vulnerable. Due to the difficulty of correcting the semantics of Ethereum, Luu et al. [11] also implemented a symbolic execution tool `OYENTE` to find security bugs. A framework to analyze and verify both the runtime safety and the functional correctness of Solidity contracts in F* was presented in [2].

4 Conclusion and Future Work

References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive **2016**, 1007 (2016)
2. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96. ACM (2016)
3. Documentation, S.: `https://solidity.readthedocs.io/en/v0.5.6/`, last ccessed April 2, 2019
4. Ethereum: `https://github.com/ethereum`, last ccessed April 2, 2019
5. implementation of Ethereum, A.R.: `https://github.com/cryptape/ruby-ethereum`, last accessed April 2, 2019
6. implementation of the Ethereum protocol, O.G.: `https://github.com/ethereum/go-ethereum`, last accessed April 2, 2019
7. Filliâtre, J.C., Paskevich, A.: Why3-where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
8. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.M., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the ethereum virtual machine. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 204–217. IEEE Computer Society (2018)
9. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: International Conference on Financial Cryptography and Data Security. pp. 520–535. Springer (2017)
10. in Javascript, I.E.V.: `https://github.com/ethereumjs/ethereumjs-vm`, last accessed April 2, 2019
11. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
12. Nakamoto, S., et al.: Bitcoin: A peer-to-peer electronic cash system (2008)
13. Project, T.H.: `https://github.com/dapphub/dapptools/tree/master/src/hevm`, last accessed April 2, 2019
14. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: International Conference on Financial Cryptography and Data Security. pp. 478–493. Springer (2017)
15. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
16. Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H.: Blockchain challenges and opportunities: a survey. International Journal of Web and Grid Services **14**(4), 352–375 (2018)