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 [13] blockchain system, cryptocurrencies have become a global revolutionary phenomenon known to majority of people. Meanwhile, the decentralized system with no server or central authority, which emerges as a side product of Bitcoin, shows great potential in carrying out secure online transactions. From then until now, there have been so much change and growth on blockchain technology. The researchers and developers see plenty of potential and massive possibilities of blockchain applications, especially in financial, governmental services. Ethereum[1] extends Bitcoin's design, which can not only process transactions but also process complex smart contracts and programs. 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 high level programming language called *Solidity* [6] and then compiled into low-level machine instructions (*opcodes*), which is 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. Atzei et al. presented a taxonomy of vulnerabilities and related attacks against Solidity, the EVM, and the blockchain in [7]. The adoption of formal methods can facilitate the production of trustworthy and reliable software systems. Verification such as theorem proving techniques and testing have become essential to guarantee the correctness of safety-critical systems.

To address the security challenges against EVM, we present a framework of generating formally verified EVM for production environment in this paper. The contributions of this work are: Firstly, a formal definition of EVM is specified in WhyML, the language of programming and specification in Why3. Secondly, an implementation of EVM in OCaml is generated through an extraction mechanism based on a series of customized drivers. Lastly, 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: Section 2 presents some background about Why3, Ethereum Virtual Machine (EVM) and smart contracts. We outline the framework for formalizing, property verifying and testing of EVM in Section 3. Based on the framework, we perform some experiments in Section 4 and highlight th e verification and testing results for the properties of EVM. Section 5 presents some related work. Finally, we summarized this paper and point out the future research direction in Section 6.

2 Preliminary

2.1 Why3

Why3 [9] is a tool for deductive program verification. It provides a standard library of logical theories, such as integer and real arithmetic, and basic programming data structures, such as arrays and queues. WhyML is the programming and specification language of Why3. The specification language is used to write program annotations and background logical theories, which serves as a common format for proof goals. Ghost code is also supported in WhyML, which serves to facilitate verification without affecting the final result of a program.

With the specification language formalizing the properties, a verification condition (VC) generator can produce the proof obligations that need to be discharged. Furthermore, logical goals can be proved using a series of automated or interactive theorem provers, including Alt-Ergo, CVC3, CVC4, Z3, Coq and PVS. To get the executable code, users can write programs in WhyML and obtain correct-by-construction OCaml programs through an automated extraction mechanism. In the extraction process, uninterpreted WhyML types are either mapped to existing OCaml type or left as abstract data type. And such mapping can be customized through user-defined drivers.

2.2 EVM and Smart Contract

Different from the general virtual machine, Ethereum Virtual Machine (EVM) is designed to serve as a run-time environment for smart contracts, whose specification is tersely defined in Ethereum Yellow Paper[15]. EVM is a 256-bit register stack-based architecture, which can store 1024 items at most. The memory model of EVM is a volatile word-addressed byte array. Some opcodes use contract memory to retrieve or pass data. When some contract execution finishes, the memory contents are cleared. Unlike the memory model, the storage model of EVM is non-volatile which acts like a database. The data stored in *storage* is accessible for future contract executions.

Smart contracts are essentially programs deployed on the Ethereum blockchain used to conduct transactions or perform specific actions. Any user can create a contract by posting a transaction to the blockchain. The program code of the contract is fixed after deployment and it will be invoked whenever it receives a transaction.

3 EVM Framework

In this section, we demonstrate the EVM framework in detail. The main idea of the framework 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.

3.1 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. Towards the verification, verification conditions can be further generated through Why3, based on the EVM and property specification. The verification goals will then 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 slovers cannot deal with, users can resort to interactive theorem proves 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 a initial state and leads to a series of changes in the stack, memory and 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 `UInt256`, `UInt160` and etc. 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 and etc.

Instruction Definition The infrastructure has been built to specify the state of the virtual machine. Here we present the formal specification of the action set, i.e., the instruction set. Inspired by the instruction formalization in Lem [11], we defined the instruction set in multiple groups, such as arithmetic operations, stack operations and etc., then integrated them into a summarized type definition `instruction`. The definition of `instruction` is presented as follows, where different subsets of instructions are wrapped up to form the complete specification.

```
type instruction =
  | Invalid byte
  | Arith arith_inst
  | Sarith sign_arith_inst
  | Bits bits_inst
  | Info info_inst
  | Memory memory_inst
  | Storage storage_inst
  | Pc pc_inst
  | Stack stack_inst
```

```
  | Dup dup_inst
  | Swap swap_inst
  | Log log_inst
  | System system_inst
```

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.

3.2 Experimental Testing

We customizedly defined a series of drivers to extract WhyML programs into Ocaml programs.

4 Experiment and Analysis

We have presented the formal implementation of EVM in Why3, based on which important properties can be verified through automatic and interactive theorem provers. We proceed to test our executable OCaml implementation extracted from WhyML against the standard test suite.

5 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 cryptocurrency 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. [10]. Hirai [11] 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 [2] 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 [3], Go [4], 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 [12], 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. [12] 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 [8].

6 Conclusion and Future Work

References

1. Ethereum project. `https://github.com/ethereum`, accessed April 2, 2019
2. The hevm project. `https://github.com/dapphub/dapptools/tree/master/src/hevm`, accessed April 2, 2019

3. Implements ethereum's vm in javascript. https://github.com/ethereumjs/ethereumjs-vm, accessed April 2, 2019
4. Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum, accessed April 2, 2019
5. A ruby implementation of ethereum. https://github.com/cryptape/ruby-ethereum, accessed April 2, 2019
6. Solidity documentation. https://solidity.readthedocs.io/en/v0.5.6/, accessed April 2, 2019
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive **2016**, 1007 (2016)
8. 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)
9. Filliâtre, J.C., Paskevich, A.: Why3-where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
10. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: Kevm: A complete semantics of the ethereum virtual machine. Tech. rep. (2017)
11. 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)
12. 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)
13. Nakamoto, S., et al.: Bitcoin: A peer-to-peer electronic cash system (2008)
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)