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

3.1 Methodology of EVM Framework

This framework can be considered as an experiment in a methodology that combines verification and testing for developing secure EVM implementations, and also as a tool for verifying smart contracts after compiling into bytecodes. Figure TOREF outlines the whole framework for developing a formally verified EVM in production environment.

3.2 EVM in Why3

According to the Ethereum project yellow paper[15], we translated EVM into Why3, which provides a programming and specification language WhyML. Based on the EVM definitions, Why3 further generates verification conditions according to the property specification, and allows the users to split the goals into different subgoals and prove each subgoal through different 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.

As have been mentioned before, EVM is 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 perspective we see the execution process of a sequence of opcodes (instructions), which could be complied from a smart contract, is as a state transition process and it finally leads to a series of changes in the stack, memory and etc.

Type Definition Based on the needs of type requirements of Ethereum, such as that the machine word of EVM is 256 bits and the address is 160 bits, we developed a series of type modules `UInt256`, `UInt160` and etc. to ease the representation of the corresponding types in Ethereum. Why3 also supports type alias which makes the source code in WhyML more readable. Stack is defined as a list of elements whose type are `uint256`, aliased by `machword`. Memory is defined as a function that maps `machword` to an option type `option memory_content`. The `memory_content` is an enumeration type including `Item8` and `Item256` since the memory content could be accessed through 256-bit store and 8-bit store instructions. Storage is defined as a function that maps `machword` to `machword`. There are also other enumeration types defined to reflect the virtual machine status, the operation error, and a big view of the returned result which are `vmstatus`, `error` and `return_type`. The record type machine_state represents the machine state whose components contain stack, memory, program counter, vmstatus, the instruction list and etc..

Instruction Definition Inspired by the instruction formalization of EVM in Lem [11], we defined the instruction set in different groups, such as arithmetic operations, stack operations and etc., then integrated them into a final type definition. The final instruction type definition is presented as follows, which shows the different instruction groups:

```
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 group is a bit different from the yellow paper[15]. The comparison operations are contained in the arithmetic operations `Arith` and the comparison operations for signed arithmetic are contained in the signed arithmetic operations `Sarith`, respectively. The bitwise operations and `BYTE` operation are included in the bit-related instructions `Bits`. Almost all of the information related instructions including environmental and block information are defined in type `info_inst`, except some `CALL` and `CODE` instructions like the `CALLDATACOPY`, `CODECOPY` and `CALLDATALOAD` which are closely related to memory and stack. Therefore, for the memory and stack instruction groups, some extra instructions are added to the original instruction subsets presented in the yellow paper. In case when there exists some instruction that is illegal, the instruction `Invalid` is included in the instruction type definition. The `STOP` operation and the system operations from the yellow paper except `Invalid` are contained in the *System* instruction set. As for the remaining instruction groups, they are basically the same as the corresponding instruction subsets in [15].

Interpreter Definition The function interpreter maps from some machine state to a new machine state, mainly in charge of formalizing the state transition result of different instructions. In order to make the definition of interpreter more concise, we defined some auxiliary functions. For example, the following function is used to obtain the instruction which needs to be executed from the instruction list according to 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
```

Item pop and push operations are most commonly-used manipulations for the state transition of stack. The auxiliary functions `push_stack` and `pop_stack` are defined to control the stack change. For the formalization of `Swap` instructions, two recursive functions `fetch` and `drop` are defined to manipulate lists and one main function `swap_stack` that implements the stack swap instruction is defined further based on these two functions. With the pre-defined auxiliary functions, the definition of the interpreter function is essentially comprised of component update of the machine state with regard to concrete instructions.

3.3 OCaml Extraction and Rust Connection

We customizedly defined a series of drivers to extract from WhyML programs to 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

0 The research interest of using blockchain technology has exploded since the inception of Bitcoin. As the popularity of the second generation blockchain Ethereum grows, a series of security vulnerabilities has also appeared. Since

EVM and smart contracts deal directly with the transactions of valuable cryptocurrency units between different parties, the security of contract programs and EVM implementations is of paramount importance. To address the security challenges, researchers resort to the techniques of formal methods and program analysis.

– **Formalization Foundation of EVM** An executable formal semantics of EVM have 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, and further proved some safety properties of smart contracts in Isabelle/HOL. The `hevm` project[2] is implemented in Haskell especially 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 Ruby[5], Javascript[3], Go[4] and etc.
– **Verification of Smart Contracts** Sergey et al. [14] provides 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. [12] pointed out several new security problems and proposed a way to enhance the operational semantics of Ethereum to make smart contracts less vulnerable. Due to the difficulty of correcting the semantics of Ethereum, Luu et al.[12] also implements 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

6.1 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation etc. does not need an indent, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

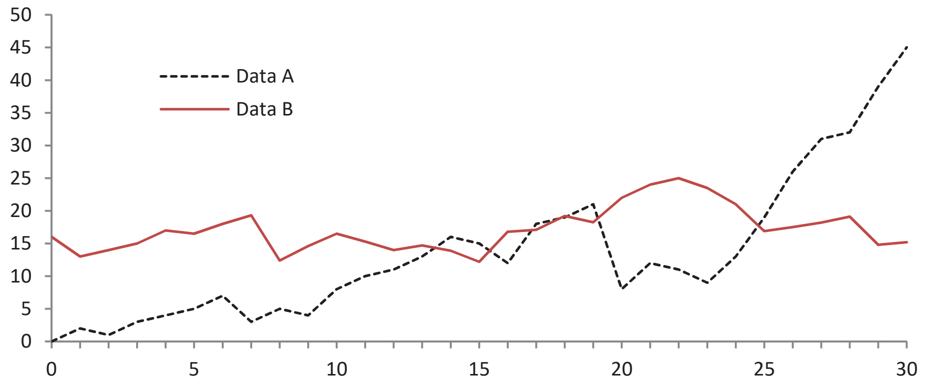
Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 1 gives a summary of all heading levels.
Displayed equations are centered and set on a separate line.

$$x + y = z \tag{1}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 6.1).

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic



Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

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.: Why3where 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)