Towards a Formally Verified EVM in Production Environment

Xiyue Zhang[1], Yi Li[1] and Meng Sun[1,2]

[1]School of Mathematical Sciences, Peking University, Beijing, 100871, China
[2]Center for Quantum Computing, Peng Cheng Laboratory, Shenzhen, 518055, China
{zhangxiyue, 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. 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 [10], 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 [15], 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 [3] extends Bitcoin's design, which can process not only transactions but also complex programs and *smart contracts*. Smart contracts running on the blockchain make it possible to use blockchain techniques in many other application domains besides cryptocurrencies, and have attracted a lot of attention from government, finance, health, entertainment and industry. This feature makes Ethereum a popular ecosystem 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* [12] and then compiled into EVM bytecode, which can be mapped into a list of machine instructions (*opcodes*). Ethereum Virtual Machine

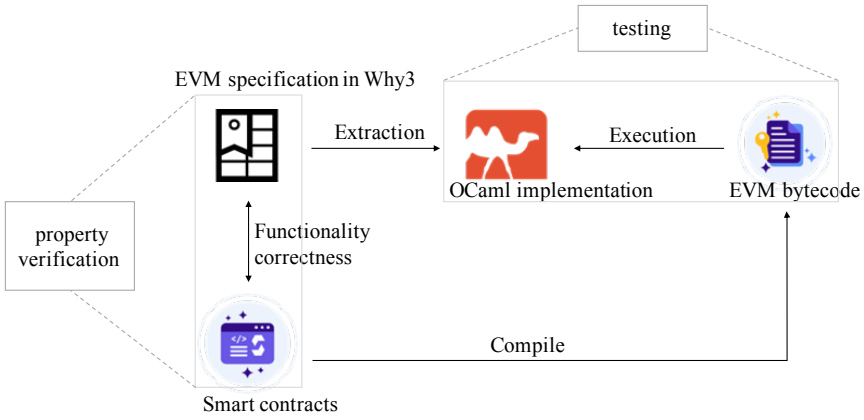

Fig. 1. the Framework of Generating Verified EVM for Production Environment

(EVM) is a quasi-Turing complete machine. It provides a runtime environment for smart contracts to be executed. Given a sequence of bytecode instructions, which are compiled from smart contracts by an 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. A taxonomy of vulnerabilities and related attacks against Solidity, the EVM, and the blockchain is presented in [1]. To deal with the security challenges against EVM, we propose 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 [5].
- 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 summarize this paper in Section 4.

2 The Framework of Generating Verified EVM for Production Environment

In this section, we present the framework of generating verified EVM for production environment in detail. The framework is as shown in Fig. 1 and the main idea is to combine verification and testing techniques towards developing more

secure EVM implementations. It also provides a platform to verify the functionality properties of smart contracts. This framework is mainly comprised of two parts: (1) EVM specification and property verification in Why3; (2) experimental testing based on OCaml extraction and Rust connection. This approach leverages formal methods and engineering approaches, allowing us to perform both rigorous verification and efficient testing for EVM implementations and 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 [14]. More specifically, the EVM implementation is translated into WhyML, the programming and specification language of Why3. Verification conditions can be further generated based on the pre- and post-condition specification. Generated verification goals are solved directly through the supported solvers or go through a sequence of transformations first. In cases when the automatic SMT solvers cannot deal with, users can resort to interactive theorem provers for the remaining unsolved proof goals.

EVM is essentially a stack-based machine. The memory model of EVM is a word-addressed byte array and the storage model is a word-addressed word array. These three components form the infrastructure of EVM. Based on the formalization of the infrastructure, the most important aspect in this framework 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. The main function *Interpreter* provides the specification of transition results for the instructions.

Type Definition To formalize the infrastructure of EVM, we need to first provide the formalization of commonly-used types in EVM, such as the types of machine words and the addresses in the EVM. Hence, we developed a series of type modules such as `UInt256` and `UInt160` to ease the representation of corresponding types in EVM. Type alias supported by Why3 are also used 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`. 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. Furthermore, the record type `machine_state` is defined to represent the overall machine state which consists of stack, memory, storage, program counter, vmstatus, the instruction list, etc.

Instruction Definition The infrastructure has been built above to specify the state of the virtual machine. Inspired by the instruction formalization in Lem [7], the instruction set is defined in multiple groups, such as *arithmetic operations* and *stack operations*, then these groups are integrated 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 [14]. The information related instructions including environmental and block information are defined in type `info_inst`, except `CALL` and `CODE` instructions, such as `CALLDATACOPY`, `CODECOPY` and `CALLDATALOAD`. These instructions are more closely related to memory and stack status. Therefore, they are added to the memory and stack instruction groups. In case when some illegal command occurs, the instruction `Invalid` is included in the `instruction` definition. The specification of the remaining instruction groups are basically the same as the corresponding instruction subsets in [14].

Interpreter Definition The specification of `interpreter` formalizes the state transition result of different instructions. For a specific instruction, the interpreter determines the result machine state developing from the current state. Some auxiliary functions are defined to make the definition of the interpreter more concise and compact.

```
let interpreter (m: machine_state): machine_state =
let inst = get_inst m in
match inst with
  | Some (Arith ADD) -> let (st', a) = (pop_stack (m.mac_stack)) in
    let (st'', b) = (pop_stack st') in ...
    {m with mac_stack = push_stack st'' (a' + b'); ...}
```

In the above code snippet, `get_inst` is used to obtain the next instruction to be executed. It is obtained from the instruction list following the program counter. In the case of `Arith ADD` instrcution, the numbers to perform the add operation on are popped out of the stack first and the result is pushed into the stack after the calculation. As a result, the stack state is updated as a component of the machine state. In this process, functions `push_stack` and `pop_stack` are defined to control the push and pop manipulations for the state transition of stack. With the support of pre-defined auxiliary functions, the definition of the interpreter function is essentially comprised of machine state update with regard to the instructions.

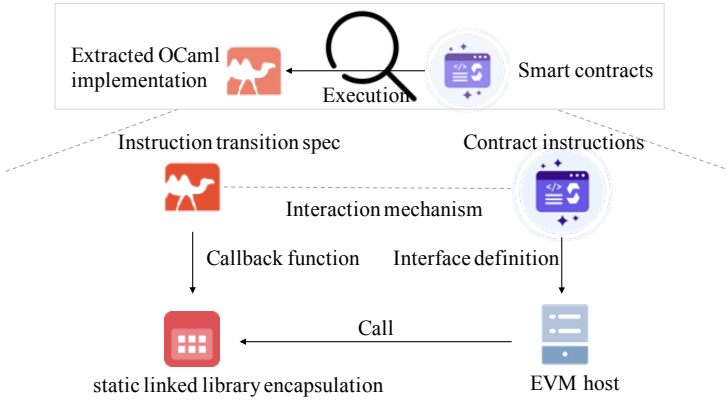

Fig. 2. Running EVM in Production Environment

2.2 Running EVM in Production Environment

Fig. 2 shows the second phase of the framework: deploy the extracted OCaml implementation from 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.

Rust is a multi-paradigm system programming language which is designed to provide better memory safety while maintaining high performance [8]. The framework provides the interaction mechanism between Rust and Why3. By gluing them together, verified models can be directly executed in production environments for further testing. The coupling between Rust and extracted OCaml implementation enables us to perform VM tests to test the basic workings of the verified VM. Information of the overarching environment is obtained through the interface of Rust implementation, and the test can be performed on the execution of the OCaml implementations to check the operations in different transactions.

2.3 Examples of Property Verification and Tests

We now show some examples of property verification towards smart contracts and tests against Ethereum test suites. Specifically, we present the specification

and verification of *SafeMath* library and *SimpleAuction* contract. For the tests, we perform the testing of arithmatic operations against the Ethereum test suite.

Overflow/Underflow property verification. We first take the example of *SafeMath* from Solidity library. Overflow/Underflow problems often occur when we deal with number operations. For EVM, the unsigned integer type we perform arithmatic operations on range from 0 to 2^{256}, which is specified as `uint256` in the WhyML specification. The properties we verify are to guarantee that overflow and underflow problems would not occur in the number operations. Besides, the correctness of the operation results is also specified in the postconditions and further verified, for example, the last postcondition in the function `div_safe`.

As can be seen from the following definition of `div_safe`, the function body is comprised of three parts, as a Hoare triple, preconditions, program expressions and postconditions. The first precondition specifies that the divisor should be greater than zero. The first postcondition states that the returned value should satisfy the required property with no underflow issues. The other two postconditions are to guarantee the correctness of the operation result.

```
let div_safe (a:uint256) (b:uint256): uint256
requires {to_int b > 0}
ensures {to_int result >= 0}
ensures {to_int a = 0 -> to_int result = 0}
ensures {to_int a <> 0 ->
to_int a = (to_int result) * (to_int b) + mod (to_int a) (to_int b)}
= a / b
```

We now proceed to the verification of the properties. The verification conditions can be obtained through running `why3 prove` on the WhyML file. The proving goals for `div_safe` are derived as follows:

```
goal VC div_safe :
forall a:uint256, b:uint256.
  to_int b > 0 -> (not b = 0 && in_bounds (div a b)) /\
  (forall result:uint256. result = div a b -> to_int result >= 0 &&
  (to_int a = 0 -> to_int result = 0) &&
  (not to_int a = 0 ->
  to_int a = (to_int result * to_int b + mod (to_int a) (to_int b))))
```

To prove the goals, we first apply the `split VC` transformation and then call theorem provers *alt-ergo* and *cvc4* to prove the subgoals automatically. The proof session state will be stored in an XML file, which includes the proved WhyML file, the applied transformations, the used provers and the proof results. Complete proving goals derived from the functions and proof sessions can be found at [4].

Open Auction contract verification. The open auction contract is mainly comprised of three functions: (1) Everyone can send their bids through the `bid` function when the bidding period is not finished. When the bid sent by one

bidder exceeds the current recorded highest bid, the auction state including the highestBidder and highestBid would be updated. Then the withdrawal amount of the previous highest bidder should be increased by the previous highest bid. (2) When one bid is beaten by another higher raised bid, the previous bid should be returned back to the corresponding bidder. Bidders can call the `withdraw` function to get the money/Ether back. (3) The auction is ended by the `auctionEnd` function. If current time is already greater than the auctionEndTime, then the auction `end_state` should be set to True. As the bidding ended, the beneficiary would receive the final highest Bid.

In the WhyML specification, `auction_status` records the current state of the auction including the current highest bidder, the highest bid and the auction ended state. `auction_constant` records the beneficiary and the auctionEndTime and `auction_ended` records the final bidder, bid and the beneficiary claimed money/Ether amount. The properties to be verified are to guarantee the correctness of the functionality. For example, in the `auctionEnd` definition, the postcondition specifies the constraints of auction ended state and beneficiary claimed amount that the returned result should satisfy. Complete specification of the functions can be found at [4]. The generated verification conditions can be discharged through *alt-ergo* and *cvc4* automatically.

```
let auctionEnd (current_time: uint) (auc_st: auction_status)
(auc_const: auction_constant) (auc_end: auction_ended):
(auction_status, auction_ended)
... ensures {let (_auc_st, _auc_end) = result in
_auc_st.end_state = True
&& _auc_end.finalBidder = auc_st.highestBidder
&& _auc_end.finalBid = auc_st.highestBid
&& _auc_end.bene_amount.benefici = auc_const.beneficiary
&& _auc_end.bene_amount.benefit_amount = _auc_end.finalBid} = ...
```

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 vulnerabilities have also appeared. Since EVM and smart contracts deal directly with the transactions of valuable cryptocurrency units among multiple parties, the safety of smart contracts and EVM implementations is of paramount importance. To address these challenges, researchers resorted to the techniques of formal methods and program analysis.

Specification and Verification. An executable formal semantics of EVM has been created in the K framework by Everett et al. [6]. Compared with KEVM with the support of matching logic for verification, we use Hoare logic, which serves as a good framework for verification condition specification, to avoid the complex definitions of the operational semantics. A framework to analyze and verify the safety and the correctness of Solidity smart contracts in F* was presented in [2]. Hirai [7] proposed an EVM implementation in Lem, a language that

can be compiled for a few interactive theorem provers. Then, safety properties of smart contracts can be proved in proof assistants like Isabelle/HOL. While in our work, we use WhyML for specification and programming, which supports both logical theories and programming data structures. Moreover, both automated and interactive external theorem provers can be relied on to discharge verification conditions.

Testing and Debugging. The `hevm` project [13] is implemented in Haskell for unit testing and debugging of smart contracts. Sergey et al. [11] provided a new perspective between smart contracts and concurrent objects, based on which existing tools for understanding and debugging concurrent objects can be used on smart contract behaviors. In [9], 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. [9] also implemented a symbolic execution tool `OYENTE` to find security bugs. While in our work, executable OCaml programs can be directly extracted from WhyML programs for further tests with the support of customized drivers and extraction mechanism.

4 Conclusion

We propose a framework to enable formal specification, verification and testing towards EVM. In this framework, the formalization of EVM is specified in WhyML, based on which, automatic SMT solvers and interactive theorem provers can be employed for verification. The OCaml implementation of EVM is extracted from the WhyML specification and then glued with Rust implementation based on the coupling framework. The coupling framework provides the interaction mechanism between OCaml and Rust, which allows us to perform tests on the new implementation without additional interface implementation.

Acknowledgement

This work has been supported by the National Natural Science Foundation of China under grant no. 61772038 and 61532019, and the Guangdong Science and Technology Department (Grant no.2018B010107004). Thanks to members of Cryptape, especially Jan and Zhiwei, for the helpful discussions during the development of this framework.

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., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: Proceedings of PLAS@CCS 2016. pp. 91–96. ACM (2016)

3. Ethereum: `https://github.com/ethereum`, last accessed April 2, 2019
4. Examples: `https://github.com/Xiyue-Selina/coordination20`
5. Filliâtre, J.C., Paskevich, A.: Why3-where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
6. 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)
7. 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)
8. Lin, Y., Blackburn, S.M., Hosking, A.L., Norrish, M.: Rust as a language for high performance GC implementation. In: Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016. pp. 89–98. ACM (2016)
9. 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)
10. Nakamoto, S., et al.: Bitcoin: A peer-to-peer electronic cash system (2008)
11. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: International Conference on Financial Cryptography and Data Security. pp. 478–493. Springer (2017)
12. Solidity Documentation: `https://solidity.readthedocs.io/en/v0.5.6/`, last accessed April 2, 2019
13. The Hevm Project: `https://github.com/dapphub/dapptools/tree/master/src/hevm`, last accessed April 2, 2019
14. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
15. 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)