Modeling EVM in the K framework

Deepak Kumar Everett Hildenbrandt Manasvi Saxena Zane Ma
University of Illinois Urbana-Champaign

ABSTRACT

Ethereum is growing in size and value rapidly, due to the key property that ethereum enables a programmatic interface by which users can interact with the currency by way of smart contracts. Because contracts inherently handle money, there is much scrutiny to the contracts that are placed on the ethereum blockchain. Perhaps more worrying, bugs in these contracts can have catastrophic consequences, which can jeopardize the integrity of the currency as a whole. In this work, we model the Ethereum Virtual Machine (EVM) in K, a language built specifically for extensible, formal verification of programming languages, and use our tool to verify a set of EVM programs for correctness.

1. INTRODUCTION

The growing popularity of smart contracts has led to an increased scrutiny of the security of smart contracts. Contracts inherently involve money, so bugs in such programs can often be devastating to the involved parties. An example of such a catastrophe is the recent DAO attack [2], where $150 million worth of ether was stolen, prompting and unprecedented hard fork of the ethereum blockhain. While devastating, the bug that caused such an attack is well known as a common bug in ethereum smart contracts. In fact, many such classes of bugs exist in smart contracts, ranging from transaction-ordering dependence to mishandled exceptions. Even a contract running out of gas before the full completion of a function can interfere with the correctness of smart contracts, and so it is important to do so in order to protect the integrity of ethereum to its many users.

As the smart contract ecosystem grows in complexity, there is an additional reliance on code from external sources in order to enhance the development cycle for smart contracts. While this code is ostensibly correct, there is in fact no mechanism for verifying the correctness of these programs, as implementation details can differ drastically from source to source. from For example, developers often include code from external libraries or even from sources such as StackOverflow to include in their ethereum contracts.

Smart contracts are an attractive method for formal program verification, which can theoretically provide guarantees on smart contract execution safety, liveness, and other forms of application logic. In particular, K is an extensible semantic framework that is used to model programming languages, type systems, and formal analysis tools [4]. While there are a number of semantic frameworks to choose from, we are choosing to use K in our formal analysis primarily because of its simplicity and extensibility.

In this paper, we build a K model of the Ethereum Virtual Machine (EVM) for use in verifying properties on smart contracts. The need for verification of these programs is well motivated, and to our knowledge, no such verification has been done using the power of modern verification tools in the academic literature. The contributions of this paper are as follows:

- **EVM Semantics in K** We implement EVM semantics in the K framework, and cover a large subset of EVM instructions and their properties.
- **EVM Program Verification** We create a simple EVM program that sums values from 1 to some input N, and verify that the program for correctness. We then show how this can be extended to any given EVM program.
- **EVM Property Checking** We show that certain properties, for example minimum gas required for proper execution, can be posited by way of our system.

2. BACKGROUND

2.1 Ethereum + EVM

TODO: Zzzzma

2.2 Verification with K

TODO: Everett + Manasvi

3. METHODOLOGY

We used K and modeled the EVM, to say interesting things about EVM programs.

3.1 How K Works

3.2 Our configuration and design

3.3 Implementing Instructions

After designing our K configuration, we then implemented a large subset of the EVM instructions detailed in the Ethereum Yellow Paper [**?**]. At the point of writing, we have implemented almost 60% of the instructions in EVM, and can run a vast number of EVM programs using our subset of instructions. In this section, we discuss the classes of instructions we have already implemented, as well as outstanding instructions that need to be implemented.

Throughout the model, we implement several classes of instructions. The first kind of instruction deals with operations on the stack. We group instructions based on how many elements they pull off of the stack, for example, a *BinaryStackOperation* pulls two elements off of the stack, performs some computation on those elements, and then pushes the elements back onto the stack.

Other instructions may use the word stack but also make use of other parts of the K configuration, for example, the programCounter or localMemory. A JUMP instruction touches the programCounter to jump to previous instructions; a MLOAD instruction will load a word from a program's local memory and move it to the stack.

TODO: Something about function calls

4. EVALUATION

We evaluated our model for correctness based on our written tests and by use of some contracts written for specific purposes on the Ethereum blockchain.

5. RELATED WORK

Luu et al. [3] formalized a subset of EVM semantics, and used a symbolic execution tool based on their formalized semantics to detect problems in 8519 existing smart contracts. Their semantics however, are incomplete, and can only be used to detect a limited class of problems. For instance, their semantics do not consider the role of gas in Ethereum transactions, and cannot be used to prove properties concerning gas on smart contracts.

There exists extensive literature on the use of language semantics for program verification. Stefanescu et al. [1] in their work presented a language independent verification framework for program verification. The verification infrastructure presented in their work however, has to be initialized with the operational semantics of the language. Hence, having complete semantics of EVM in K would allow for using existing infrastructure for smart contracts verification.

6. FUTURE WORK

While we have made significant efforts towards formal verification of smart contracts, there is still much work to be done to verify all of the nuanced properties of smart contracts. Our future work is split into a few stages,

7. CONCLUSION

8. REFERENCES

[1] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, Nov 2016.

[2] P. Daian. Dao attack, 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. http://eprint.iacr.org/2016/633.

[4] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.