Short Paper: Towards Formal Analysis of Ethereum Smart Contracts

Abstract. Part of Ethereum's rapid growth in value and adoption is due to smart contract, a programmatic interface to the currency. Because bugs in these contracts can be financially devastating, there is much scrutiny to the contracts that are placed and interacted with on the Ethereum blockchain. To mitigate concerns regarding malicious or poorly programmed smart contracts, the Ethereum community has recently turned to formal verification as a means for verifying properties of smart contracts. In this work, we model the Ethereum Virtual Machine (EVM) [16] in $\mathbb{K}$, a language built specifically for extensible, formal verification of programming languages. We build formal semantics for EVM in $\mathbb{K}$, and use the semantics to verify an EVM program with inter-contract transactions.

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 [5], where $50 million worth of Ether was stolen, prompting an unprecedented hard fork of the Ethereum blockchain. 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 identify such problems order to protect the integrity of Ethereum to its 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.

Smart contracts are an attractive use case for formal program verification, which can theoretically provide guarantees on contract execution safety, liveness, and other forms of application logic. In particular, $\mathbb{K}$ is an extensible semantic framework that is used to model programming languages, type systems, and formal analysis tools [15]. While there are a number of semantic frameworks to choose from, we are choosing to use $\mathbb{K}$ in our formal analysis primarily because of its simplicity and extensibility.

In this paper, we build a $\mathbb{K}$ model of the Ethereum Virtual Machine (EVM) for use in verifying properties on smart contracts. The contributions of this paper are as follows:

- **EVM Semantics in $\mathbb{K}$:** We implement EVM semantics in the $\mathbb{K}$ framework, and cover a substantial subset of EVM instructions and their properties.
- **EVM Program Verification:** We create a simple EVM program that contains an inter-contract transaction and verify the transaction for program correctness.

We intend to release the semantics in $\mathbb{K}$ and the associated derived tools at the time of publication.

2 Background

2.1 Ethereum Virtual Machine (EVM)

Ethereum is a public, distributed ledger based on blockchain technology first proposed and popularized by Bitcoin. Whereas Bitcoin's blockchain only stores transactions that exchanges Bitcoin between addresses, Ethereum's blockchain stores the complete execution state of distributed programs known as Ethereum Virtual Machine (EVM) *smart contracts*. Smart contracts, which are often written in higher level languages such as Solidity [9] or Serpent [8], are then compiled to EVM bytecode, a stack based, Turing-complete language, which consists of 65 unique opcodes [16]. Smart contracts consist of a *contract address*, contract *balance*, and private program execution code and state.

2.2 The $\mathbb{K}$ Framework

Developing tools for each new language is labor intensive and error-prone. $\mathbb{K}$ defines its tools parametrically over the input language, avoiding some of this start-up cost. Parsers, interpreters, debuggers, and verifiers are derived directly from the syntax and operational semantics of the language. This infrastructure is developed once, then instantiated to specific languages. [4]

Developing a new language in $\mathbb{K}$ requires the definition of that language's syntax (given as a BNF-style grammar), as well as the operational semantics of that language (as transition rules). $\mathbb{K}$ provides several facilities for making this definition easier, including *configuration abstraction*; configuration abstraction allows each transition rule to only mention the parts of execution state needed for that transition.

Once a language is defined, $\mathbb{K}$ can read and execute programs in that language both on concrete and symbolic inputs, producing an interpreter and a symbolic execution engine, respectively. These can be regarded as a reference implementation of the language. $\mathbb{K}$'s Reachability Logic prover can also be used to verify functional correctness and safety properties of programs. [3] Reachability queries are provided to the prover using the same syntax as a $\mathbb{K}$ rule, and reduced to a query to the SMT solver Z3. [6]

The semantics of EVM in $\mathbb{K}$ lay the groundwork for rigorous and practical debugging and analysis of both EVM smart contracts and the EVM network. Tools for checking EVM contracts for common bugs can be developed in a high-level yet semantically-rigorous way. This is valuable to users of the EVM network, where errors in contracts often directly mean fiscal loss.

3 Methodology

Instantiation of $\mathbb{K}$ to EVM requires modeling transaction execution state and network evolution using the *configuration*, and detailing changes in transaction execution and network evolution using transition *rules*. In this section, we outline the design of our $\mathbb{K}$ model and the design decisions implemented for our tool.

3.1 EVM Execution State

The state of EVM is broken into two components: the state of an active transaction (smart contract execution), and the state of the network as a whole (account information). To mirror this, our configuration consists of two pieces.

Transaction Execution Each transaction has an associated account, a program counter, a word-stack, and a volatile local memory.

```
<k> ... </k>
<id> ... </id>
<pc> ... </pc>
<gas> ... </gas>
<wordStack> ... </wordStack>
<localMem> ... </localMem>
<callStack> ... </callStack>
```

The `<k>` cell holds the next execution step of the EVM. The `<id>` cell holds the ID of the account the EVM contract belongs to and the `<pc>` cell holds the current program counter. The `<gas>` cell holds the current gas consumption of the executing contract. The `<wordStack>` is the simple stack available to the EVM program being executed, and the `<localMem>` cell holds the volatile auxiliary memory available to the program. The `<callStack>` cell records the calls made across a transaction.

Network State The network is modeled as a map from account ids to their associated program, cold storage, and current balance. This can be thought of as the blockchain state at any given time.

```
<accounts>
    <account multiplicity="*">
        <acctID> ... </acctID>
        <program> ... </program>
        <storage> ... </storage>
        <balance> ... </balance>
    </account>
</accounts>
```

The `<accounts>` cell holds information about the accounts on the blockchain. Each `<account>` holds the program associated with it in the `<program>` cell, its permanent storage in the `<storage>` cell, and the balance associated with the account in the `<balance>` cell.

3.2 EVM Transition Rules

EVM smart contracts consist of a sequence of commands for effecting the local execution state and the global network state. Commands are grouped using $\mathbb{K}$'s sorts; for example we have the `StackOp` sort for stack operators, the `ControlFlowOp` sort for

operations which affect control flow (e.g. calling other contracts), and the `LocalOp` sort for other operations that may effect the local transaction state. At the point of writing, we have implemented almost 60% of the EVM opcodes.

Semantics of EVM instructions are implemented using the Ethereum Yellow Paper as a reference [16]. They are provided as $\mathbb{K}$ rules over the execution state. For example, below are the rules for `MLOAD` and `MSTORE`:

```
rule MSTORE INDEX VALUE => #updateLocalMem INDEX VALUE

rule <k> MLOAD INDEX => VALUE ~> #push ... </k>
     <localMem> ... INDEX |-> VALUE ... </localMem>
```

The first rule states that if the next instruction is `MSTORE` with arguments `INDEX` and `VALUE`, then the next instruction should be `#updateLocalMem INDEX VALUE` (an auxiliary internal command for working with the local memory).

The second rule states that if the next instruction is `MLOAD` with argument `INDEX`, and somewhere in the local memory we have the binding `INDEX |-> VALUE`, then `VALUE` should be pushed onto the top of the word-stack. Notice that we take advantage of $\mathbb{K}$'s *configuration abstraction*, which allows each rule to only specify the parts of the configuration (defined above) relevant to it.

Inter-contract Execution EVM allows contracts to call each other, accounting for the rich dynamics possible in the Ethereum network but also providing a wealth of potential bugs and pitfalls in contract design. The next two rules model one contract making a `CALL` to another. Similar rules for `RETURN`, which ends execution of the current function call and passes some values back to the caller, are defined in our semantics.

```
rule <k> CALL ACCT ETHER INIT SIZE
      => #processCall { ACCT | ETHER | #range(LM, INIT, SIZE) }
      ... </k>
     <localMem> LM </localMem>
```

`CALL` takes four arguments: which process to call (`ACCT`), the amount of Ether to transfer (`ETHER`), and the region of local memory to use as arguments (`#range(LM, INIT, SIZE)`). `#processCall` (an internal helper command) is used to complete the call:

```
rule <k> #processCall {ACCT | ETHER | WL}
      =>    #decreaseAcctBalance CURRACCT ETHER
         ~> #increaseAcctBalance ACCT ETHER
         ~> #pushCallStack
         ~> #setProcess {ACCT | 0 | .WordStack | #asMap(WL)}
     ... </k>
     <id> CURRACCT </id>
```

The current accounts balance is reduced (`#decreaseAcctBalance CURRACCT ETHER`), then the callee's account balance is increased, the current execution state is pushed onto the `<callStack>` using internal command `#pushCallStack`, and the callee is set as the current process. The operator `~>` can be read as "followed by", similar to the semicolon in many programming languages.

4 Evaluation

In order to utilize the generated deductive program verifier, we wrote an EVM program (referred to as the SUM program) that given number $n \geq 0$, computes the sum from 0 to n. We then verified that when called from another contract, the program returns $\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$.Inputs to the $\mathbb{K}$ prover are provided as a specification file, which we detail below.$\mathbb{K}$ uses the underlying reachability logic based proof system [4] to prove the claims provided in the specification file. The intuition behind our claim is as follows — if the system starts in a configuration (with symbolic values representing concrete inputs), it will eventually end in a state where correctness constraints hold over the symbolic values. For instance, we prove that if the system starts in a state with a "CALL" to the SUM program, and symbolic value Z is passed to it, it will eventually end in a state where the value returned by the SUM program is $\frac{Z \times (Z+1)}{2}$ and the program reaches a STOP instruction or diverges.

```
<T>
    <k> CALL => STOP </k>
    <id> 1 </id>
    <pc> 10 => 11 </pc>
    <gas> _ => _ </gas>
    <gasPrice> 1 </gasPrice>
    <callStack> .CallStack </callStack>
    <wordStack> 2 : ( 4000
            : ( 0 : ( 1 : ( 1 : ( 1 : .WordStack ) ) ) ) )
               => _
    </wordStack>
    <localMem> (0 |-> X)
               => (0 |-> X 1 |-> (X *Int (X +Int 1)) /Int 2)
    </localMem>
    <accounts>
        <account>
            <acctID> 1 </acctID>
            <program> Account1Pgm </program>
            <storage> .Map </storage>
            <balance> 40 </balance>
        </account>
        <account>
            <acctID> 2 </acctID>
            <program> Account2Pgm </program>
            <storage> .Map </storage>
            <balance> 40 </balance>
        </account>
    </accounts>
</T>
```

The second rule establishes our circularity, which is required to complete the proof in reachability logic - the proof system $\mathbb{K}$ implements [4].

```
rule
<T>
    <k> JUMP1 => STOP </k>
    <id> 2 => 1 </id>
    <pc> 7 => 11 </pc>
    <gas> _ => _ </gas>
    <gasPrice> 1 </gasPrice>
    <callStack>   1 | 10 | _ | 1 :
             ( 1 : .WordStack ) | 0 |-> X  .CallStack
        => .CallStack
    </callStack>
    <wordStack> 10 : ( A : .WordStack ) => .WordStack </wordStack>
    <localMem> 0 |-> A 1 |-> B
        => (0 |-> X 1 |-> (B +Int (A *Int (A +Int 1)) /Int 2))
    </localMem>
    <accounts>
        <account>
            <acctID> 1 </acctID>
            <program>
                Account1Pgm
            </program>
                <storage> .Map </storage>
            <balance> 40 </balance>
        </account>
        <account>
            <acctID> 2 </acctID>
            <program>
                Account2Pgm
            </program>
            <storage> .Map </storage>
            <balance> 40 </balance>
        </account>
    </accounts>
</T>
```

EVM Contract code that calls the sum 0 to n program.

Account1Pgm

```
0 |-> ( PUSH X ) 1 |-> ( PUSH 0 ) 2 |-> MSTORE
3 |-> ( PUSH 1 ) 4 |-> ( PUSH 1 ) 5 |-> ( PUSH 1 )
6 |-> ( PUSH 0 ) 7 |-> ( PUSH 4000 ) 8 |-> ( PUSH 2 )
9 |-> CALL 10 |-> STOP
```

Contract code that we verified using $\mathbb{K}$.

Account2Pgm

```
0 |-> ( PUSH 0 ) 1 |-> ( PUSH 1 ) 2 |-> MSTORE
3 |-> ( PUSH 0 ) 4 |-> MLOAD 5 |-> ( PUSH 10 )
6 |-> JUMP1 7 |-> ( PUSH 1 ) 8 |-> ( PUSH 1 )
9 |-> RETURN 10 |-> ( PUSH 1 ) 11 |-> MLOAD
12 |-> ( PUSH 0 ) 13 |-> MLOAD 14 |-> ADD
15 |-> ( PUSH 1 ) 16 |-> MSTORE 17 |-> ( PUSH 1 )
18 |-> ( PUSH 0 ) 19 |-> MLOAD 20 |-> SUB 21 |-> ( PUSH 0 )
22 |-> MSTORE 23 |-> ( PUSH 3 ) 24 |-> JUMP
```

5 Related Work

Formal verification of programming languages is not a particularly new idea, and our design decisions presented in this work are certainly influenced by prior work in this area. For example, $\mathbb{K}$ semantics exist for several large languages, most notably C [7], Java [2], and JavaScript [14], as well as for many smaller languages.

There has been recent effort and interest in formally verifying properties of smart contracts. Luu et al. [13] formalized a subset of the EVM, called EtherLite, and built a symbolic execution tool to check for common bugs in smart contracts. Why3 [10] is a tool included in the Solidity IDE that supplies formal verification of Solidity contracts. Bhargavan et al. [1] propose a method for formally verifying EVM smart contracts by converting Solidity programs or EVM bytecode to F*, a functional programming language focused on program verification.

6 Future Work

While we have made initial efforts towards formal verification of smart contracts, there is still much work to be done to verify all of their nuanced properties. Our primary goal is to finish implementing the remainder of the instructions in EVM, and including more contract properties beyond functional correctness. A simple example of this may be determining gas bounds on EVM programs, or detecting common classes of bugs in EVM programs. To this end, we also plan to build semantics for mining blocks; without such semantics, we would not be able to detect bugs in the class of transaction order dependence. We aim to use our full EVM semantics to then model higher level languages, such as Serpent [8] and Solidity [9], which smart contracts are almost ubiquitously written in. Doing so will enable verification of the compile step from these higher level languages to EVM. Ultimately, we aim to routinely apply our verification techniques to the *entire Ethereum blockchain*, and publish the resultant dataset back to the security and cryptocurrency communities.

7 Conclusion

In this paper, we contribute a formalization of the EVM semantics in $\mathbb{K}$, demonstrate verification of correctness on a simple EVM program that contains inter-contract inter-

action. As interest in formal verification of EVM programs grows, we hope that our contributions will pave the way for consistent and routine verification of smart contracts, ultimately leading to a bug-free and secure environment for further cryptocurrency development.

References

1. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.
2. D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.
3. A. Ştefănescu, c. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.
4. 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.
5. P. Daian. Dao attack, 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.
6. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.
7. C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.
8. Ethereum. Ethereum serpent documentation. https://github.com/ethereum/wiki/wiki/Serpent.
9. Ethereum. Ethereum solidity documentation. https://solidity.readthedocs.io/en/develop/.
10. J.-C. Filliâtre and A. Paskevich. Why3âĂŤwhere programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.
11. D. Guth, C. Hathhorn, M. Saxena, and G. Rosu. Rv-match: Practical semantics-based program analysis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *LNCS*, pages 447–453. Springer, July 2016.
12. C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.
13. 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.
14. D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.
15. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
16. G. Wood. Ethereum: A secure decentralised generalised transaction ledger.