A Formal Verification Tool for Ethereum VM Bytecode

Daejun Park
University of Illinois at Urbana-Champaign, USA
Runtime Verification, Inc., USA
dpark69@illinois.edu

Yi Zhang
University of Illinois at Urbana-Champaign, USA
Runtime Verification, Inc., USA
yzhng173@illinois.edu

Manasvi Saxena
University of Illinois at Urbana-Champaign, USA
Runtime Verification, Inc., USA
msaxena2@illinois.edu

Philip Daian
Cornell Tech, USA
IC3, USA
Runtime Verification, Inc., USA
phil@linux.com

Grigore Roșu
University of Illinois at Urbana-Champaign, USA
Runtime Verification, Inc., USA
grosu@illinois.edu

ABSTRACT

In this paper, we present a formal verification tool for the Ethereum Virtual Machine (EVM) bytecode. To precisely reason about all possible behaviors of the EVM bytecode, we adopted KEVM, a complete formal semantics of the EVM, and instantiated the K-framework's reachability logic theorem prover to generate a correct-by-construction deductive verifier for the EVM. We further optimized the verifier by introducing EVM-specific abstractions and lemmas to improve its scalability. Our EVM verifier has been used to verify various high-profile smart contracts including the ERC20 token, Ethereum Casper, and DappHub MakerDAO contracts.

Demo Video URL: https://youtu.be/4XBcAclq0Vk

CCS CONCEPTS

• **Software and its engineering → Software verification**;

KEYWORDS

Ethereum, smart contracts, formal verification, K framework

ACM Reference Format:
Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roșu. 2018. A Formal Verification Tool for Ethereum VM Bytecode. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264591

1 INTRODUCTION

Smart contract failures have caused millions of dollars of lost funds,[1] and rigorous formal methods are required to ensure the correctness and security of contract implementations.[2] The smart contract is usually written in a high-level language such as Solidity[3] or Vyper[4], and then it is compiled down to the Ethereum Virtual Machine (EVM) bytecode[5] that actually runs on the blockchain.

In this paper, we present a formal verification tool for the EVM bytecode. We chose the EVM bytecode as the verification target language so that we can directly verify what is actually executed without the need to trust the correctness of the compiler. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM [4], a *complete* formal semantics of the EVM, and instantiated the K-framework's reachability logic theorem prover [10] to generate a *correct-by-construction* deductive program verifier for the EVM. While it is sound, the initial out-of-box EVM verifier was relatively slow and failed to prove many correct programs. We further optimized the verifier by introducing custom abstractions and lemmas specific to EVM that expedite proof searching in the underlying theorem prover. We have been using the EVM verifier to verify the *full functional correctness* of high-profile smart contracts including multiple ERC20 token contracts [13], Ethereum's Casper[6] contract, and DappHub's MakerDAO[7] contract. Our verification tool and artifact is publicly available at [11].

Contributions. We describe our primary contributions:

- We present a formal verification tool for the EVM bytecode that is capable and scalable enough to verify various high-profile, safe-critical smart contracts. Moreover, our verifier is the first tool, to the best of our knowledge, that adopts a complete formal semantics of EVM, being able to completely reason about all possible corner-case behaviors of the EVM bytecode. See Section 5 for comparison to other tools.
- We enumerate important, concrete challenges in verifying the EVM bytecode, and propose EVM-specific abstractions and lemmas to mitigate the challenges. (Section 2 & 3)
- We present a case study of completely verifying high-profile ERC20 token contracts. We enumerate divergent behaviors we found across these tokens, illuminating potential security vulnerabilities for any API clients assuming consistent behavior across ERC20 implementations. (Section 4)

[1] https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/
[2] https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/
[3] http://solidity.readthedocs.io/en/v0.4.24/
[4] https://vyper.readthedocs.io/en/latest/index.html
[5] http://yellowpaper.io/
[6] https://eips.ethereum.org/EIPS/eip-1011
[7] https://makerdao.com/

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264591

2 EVM VERIFICATION CHALLENGES

Verifying the EVM bytecode is challenging, especially due to the internal byte-manipulation operations that require non-linear integer arithmetic reasoning, which is undecidable in general [7]. Here we provide a few examples of the challenges.

Byte-Manipulation Operations. The EVM provides three types of storage structures: a local memory, a local stack, and the global storage. Of these, only the local memory is byte-addressable (i.e., represented as an array of bytes), while the others are word-addressable (i.e., each represented as an array of 32-byte words). Thus, a 32-byte (i.e., 256-bit) word needs to be split into 32 chunks of bytes to be stored in the local memory, and those 32 chunks need to be merged back to be loaded in either the local stack or the global storage. These byte-wise splitting and merging operations can be formalized using non-linear integer arithmetic operations, as follows.[8] Suppose x is a 256-bit integer. Let x_n be the n^{th} byte of x in its two's complement representation, where the index 0 refers to the least significant bit (LSB), defined as follows:

$$x_n \stackrel{\text{def}}{=} (x/256^n) \bmod 256$$

Let *merge* be a function that takes as input a list of bytes and returns the corresponding integer value under the two's complement interpretation, recursively defined as:

$$\begin{aligned} \mathit{merge}(x_i \cdots x_{j+1} x_j) &\stackrel{\text{def}}{=} \mathit{merge}(x_i \cdots x_{j+1}) \mathbin{\underline{*}} 256 \mathbin{\underline{+}} x_j \quad \text{when } i > j \\ \mathit{merge}(x_i) &\stackrel{\text{def}}{=} x_i \end{aligned}$$

where $\underline{*}$ and $\underline{+}$ are multiplication and addition over words (modulo 2^{256}). If the byte-wise operations are blindly encoded as SMT theorems, then Z3, a state-of-the-art SMT solver, times out attempting to prove "$x = \mathit{merge}(x_{31} \cdots x_0)$". The SMT query can be simplified to allow Z3 to efficiently terminate, for example, by omitting the modulo reduction for multiplication and addition in *merge* with additional reasoning about the soundness of the omission. Despite these improvements, the merge operation still incurs severe performance penalties as solving the large formula is required for every load/store into memory, an extremely common operation.

Arithmetic Overflow. Since EVM arithmetic instructions perform modular arithmetic (i.e., $+, -, *, / \bmod 2^{256}$), detecting arithmetic overflow is critical for preventing potential security holes due to an unexpected overflow. Otherwise, for example, increasing a user's token balance could trigger an overflow, resulting in loss of the funds as the balance wraps around to a lower-than-expected value. There is no standard EVM-level overflow check, so the overflow detection varies across compilers and libraries. For example, the Vyper compiler inserts the following runtime check for an addition `a + b` over the 256-bit unsigned integers `a` and `b`:

$$\mathtt{b} == \mathtt{0} \;||\; \mathtt{a} \mathbin{\underline{+}} \mathtt{b} > \mathtt{a}$$

where $\underline{+}$ represents addition modulo 2^{256}. It seems straightforward to show that the above formula is equivalent to $\mathtt{a} + \mathtt{b} < 2^{256}$ (where $+$ is the pure addition *without* modulo reduction), but it is no longer trivial once the above is compiled down to EVM. The compiled EVM bytecode of the above conditional expression can be encoded in the SMT-LIB format as follows:

```
(not (= (chop (+ (bool2int (= b 0))
                 (bool2int (> (chop (+ a b)) a)))) 0))
```

where `(chop x)` denotes ($x \bmod 2^{256}$), and `(bool2int x)` is defined by `(ite x 1 0)`. However, Z3 fails (timeout) to prove that the above SMT formula is equivalent to $\mathtt{a} + \mathtt{b} < 2^{256}$.

Hash Collision. Precise reasoning about the SHA3 hash[9] is critical. Since it is not practical to consider the hash algorithm details every time the hash function is called in the EVM bytecode, an abstraction for the hash function is required. Designing a sound but efficient abstraction is not trivial because while the SHA3 hash is not cryptographically collision-free, the contract developers assume collisions will not occur during normal execution of their contracts.[10] A naive way of capturing the assumption would be to simply abstract the SHA3 hash as an injective function. However, it is not sound simply because of the pigeonhole principle, and thus we need to be careful when abstracting the hash function.

3 EVM-SPECIFIC ABSTRACTIONS

K's reachability logic theorem prover can be seen as a symbolic model checker equipped with coinductive reasoning about loops and recursions (refer to [10] for details of the underlying theory and implementation). The prover, in its current form, often delegates domain reasoning to SMT solvers. The performance of the underlying SMT solvers is critical for the overall performance. The domain reasoning involved in the EVM bytecode verification is not tractable in many cases, especially due to non-linear integer arithmetic. We had to design custom abstractions and lemmas to avoid the non-tractable domain reasoning and improve the scalability.

Abstraction for Local Memory. We present an abstraction for the EVM local memory to allow word-level reasoning. As mentioned in Section 2, since the local memory is byte-addressable, the load and store operations involve the conversion between a word and a list of bytes, which is not tractable to reason about in general. Our abstraction helps to make the reasoning easier by abstracting away the byte-manipulation details of the conversion. Specifically, we introduce uninterpreted function abstractions and lemmas for word-level reasoning as follows.

The term $\mathsf{nthByteOf}(v, i, n)$ represents the i^{th} byte of the two's complement representation of v in n bytes (0 being the most significant bit), with discarding high-order bytes when v does not fit in n bytes. Precisely, it is defined as follows:

$$\begin{aligned} \mathsf{nthByteOf}(v, i, n) &= \mathsf{nthByteOf}(\lfloor v/256 \rfloor, i, n-1) && \text{when } n > i+1 \\ \mathsf{nthByteOf}(v, i, n) &= v \mod 256 && \text{when } n = i+1 \end{aligned}$$

However, we want to keep it uninterpreted (i.e., do not unfold the definition) when the arguments are symbolic, to avoid the expensive non-linear arithmetic reasoning.

We introduce lemmas over the uninterpreted functional terms. The following lemmas are used for symbolic reasoning about `MLOAD`

[8] It is also possible to formalize the byte-manipulation using the bit-vector theory, but the formalization using the mathematical integer theory has an advantage of the functional specifications being succinct. Indeed, the KEVM semantics adopted the integer formalization because of the advantage.

[9] https://keccak.team/index.html

[10] The assumption is not unreasonable, as virtually all blockchains rely heavily on the collision-resistance of hash functions.

and MSTORE instructions. They capture the essential mechanisms used by the two instructions: splitting a word into a list of bytes and merging it back into the word. First, we have the bound of $\text{nthByteOf}(v, i, n)$ by definition: $0 \leq \text{nthByteOf}(v, i, n) < 256$. Then we have the following lemma for the merging operation:

$$\textit{merge}(\text{nthByteOf}(v, 0, n) \cdots \text{nthByteOf}(v, n-1, n)) = v \quad \text{if } 0 \leq v < 2^{8n} \text{ and } 1 \leq n \leq 32$$

Refer to [11] for the other lemmas of the memory abstraction.

Abstraction for Hash. We do not model the hash function as an injective function simply because it is not true due to the pigeonhole principle. Instead, we abstract it as an uninterpreted function, hash, that takes as input a list of bytes and returns an (unsigned) integer:

$$\text{hash} : \{0, \cdots, 255\}^* \to \mathbb{N}$$

Note that this abstraction allows the possibility of hash collision.

However, one can avoid reasoning about the potential collision by assuming all the hashed values appearing in each execution trace are collision-free. This can be achieved by instantiating the injectivity property only for the terms appearing in the symbolic execution, in a way analogous to universal quantifier instantiation.

Arithmetic Simplification Rules. We introduce simplification rules, specific to EVM, that capture arithmetic properties, which reduce a given term into a smaller one. These rules help to improve the performance of the underlying theorem prover's symbolic reasoning. For example, we have the following simplification rule:

$$(x \times y) \;/\; y = x \quad \text{if } y \neq 0$$

where / is the integer division.[11] We also have a rule for the masking operation, $\texttt{0xff} \cdots \texttt{f} \;\&\; n$, as follows:

$$m \;\&\; n = n \quad \text{if } m + 1 = 2^{1+\log m} \text{ and } 0 \leq n \leq m$$

where & is the bitwise AND operator, and m denotes a bitmask $\texttt{0xff} \cdots \texttt{f}$. Refer to [11] for other simplification rules.

4 CASE STUDY: ERC20 VERIFICATION

We present a case study of completely verifying high-profile, practically deployed implementations of the ERC20 token contract [13], one of the most popular Ethereum smart contracts that provides the essential functionality of maintaining and exchanging tokens.

4.1 Formal Specification

The ERC20 standard [13] informally specifies the correctness properties that ERC20 token contracts must satisfy. Unfortunately, however, it leaves several corner cases unspecified, which makes it less than ideal to use in the formal verification of token contracts.

We specified ERC20-K [9], a complete formalization of the high-level business logic of the ERC20 standard, in the K framework. ERC20-K clarifies what data (e.g., balances and allowances) are handled by the various ERC20 functions and the precise meaning of those functions on such data. ERC20-K also clarifies the meaning of all the corner cases that the ERC20 standard omits to discuss, such as transfers to itself or transfers that result in arithmetic overflows, following the most natural implementations that aim at minimizing gas consumption. The complete specifications are available at [9].

```
[transfer-success]
callData: #abiCallData("transfer", #address(TO), #uint256(VALUE))
statusCode: _ => EVMC_SUCCESS
output: _ => #asByteStackInWidth(1, 32)
log: ... (. => #abiEventLog(FROM, "Transfer",
   #indexed(#address(FROM)), #indexed(#address(TO)), #uint256(VALUE)))
storage:
 #hashedLocation({BALANCES}, FROM) |-> (BAL_FROM => BAL_FROM -Int VALUE)
  #hashedLocation({BALANCES}, TO) |-> (BAL_TO => BAL_TO +Int VALUE)
  ...
requires:
  andBool FROM =/=Int TO
  andBool VALUE <=Int BAL_FROM
  andBool BAL_TO +Int VALUE <Int (2 ^Int 256)

[transfer-failure]
callData: #abiCallData("transfer", #address(TO), #uint256(VALUE))
statusCode: _ => EVMC_REVERT
output: _ => _
log: ...
storage:
  #hashedLocation({BALANCES}, FROM) |-> BAL_FROM
  #hashedLocation({BALANCES}, TO)   |-> BAL_TO
  ...
requires:
  andBool FROM =/=Int TO
  andBool ( VALUE >Int BAL_FROM
   orBool   BAL_TO +Int VALUE >=Int (2 ^Int 256) )
```

Figure 1: Formal specification of `transfer` function

Figure 1, for example, shows part of the (simplified) specification of `transfer`. It specifies two possible behaviors: success and failure.[12] For each case, it specifies the function parameters (`callData`), the return value (`output`), whether an exception occurred (`statusCode`), the log generated (`log`), the storage update (`storage`), and the path-condition (`requires`). Specifically, the success case (denoted by `[transfer-success]`) specifies that the function succeeds in transferring the VALUE tokens from the FROM account to the TO account, with generating the corresponding log message, and returns 1 (i.e., true), if no overflow occurs (i.e., the FROM account has a sufficient balance, and the TO account has enough room to receive the tokens). The failure case (`[transfer-failure]`) specifies that the function throws an exception without modifying the account balances, if an overflow occurs.

4.2 Formal Verification

For this case study, we consider three ERC20 token implementations: the Vyper ERC20 token[13], the HackerGold (HKG) ERC20 token[14], and OpenZeppelin's ERC20 token[15]. Of these, the Vyper ERC20 token is written in Vyper, and the others are written in Solidity.

We compiled the source code down to the EVM bytecode using each language compiler, and executed our verifier to verify that the compiled EVM bytecode satisfies the aforementioned specification.

[11] Note that Z3 fails to prove this seemingly trivial formula at the time of this writing. Indeed, this issue has been fixed in the develop branch, once reported by the authors of this paper: https://github.com/Z3Prover/z3/issues/1683

[12] `transfer` admits four possible behaviors: success and failure of regular transfer (i.e., FROM ≠ TO), and success and failure of self-transfer (i.e., FROM = TO). Here we omit the self-transfer behaviors due to space limit. Refer to [9] for the complete specification.

[13] https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20_solidity_compatible/ERC20.vy

[14] https://github.com/ether-camp/virtual-accelerator/blob/master/contracts/StandardToken.sol

[15] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

Table 1: Verification time (secs) of ERC20 token contracts

	Vyper	HKG	Zeppln.		Vyper	HKG	Zeppln.
`totalSupply`	36.4	N/A	34.3	`approve`	33.9	48.4	35.4
`balanceOf`	33.3	37.3	37.1	`transfer`	148.5	198.5	219.7
`allowance`	36.4	42.3	39.6	`transferFrom`	174.4	257.6	179.2

During this verification process, we found divergent behaviors across these contracts that do not conform to the ERC20 standard. Due to the deviation from the standard, we could not verify those contracts against the original ERC20-K specification. In order to show that they are “correct” w.r.t. the original specification *modulo* the deviation, we modified the specification to capture the deviation and successfully verified them against the modified specification. Table 1 provides the performance of the verifier. Below we describe the results.

Vyper ERC20 Token. The Vyper ERC20 token is successfully verified against the original specification, implying its full conformance to the ERC20 standard.

HackerGold (HKG) ERC20 Token. In addition to the well-known security vulnerability of the HKG token,[16] we found that the HKG token implementation deviates from our specification as follows:

- *No totalSupply function*: No `totalSupply` function is provided in the HKG token, which is *not* compliant to the ERC20 standard.
- *Returning false in failure*: It returns false instead of throwing an exception in the failure cases for both `transfer` and `transferFrom`. It does not violate the standard, as throwing an exception is recommended but not mandatory according to the ERC20 standard.
- *Rejecting transfers of 0 values*: It does not allow transferring 0 values, returning false immediately without logging any event. It is *not* compliant to the standard. This is a potential security vulnerability for any API clients assuming the ERC20-compliant behavior.
- *No overflow protection*: It does not check arithmetic overflow, resulting in the receiver’s balance wrapping around the 256-bit unsigned integer’s maximum value in case of the overflow. It does not violate the standard, as the standard does not specify any requirement regarding it. However, it is potentially vulnerable, since it will result in loss of the funds in case of the overflow as the receiver’s balance wraps around to a lower-than-expected value.

OpenZeppelin ERC20 Token. The OpenZeppelin ERC20 token is a high-profile ERC20 token library developed by the security audit consulting firm Zeppelin[17]. We found that the OpenZeppelin token deviates from the ERC20-K specification as follows:

- *Rejecting transfers to address 0*: It does not allow transferring to address 0, throwing an exception immediately. It does not violate the standard, as the standard does not specify any requirement regarding it. However, it is questionable since while there are many other invalid addresses to which a transfer should not be made, it is not clear how useful rejecting a single invalid address is, at the cost of the additional gas consumption for every transfer transaction.

[16] https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued Note that the token contract had been manually audited by Zeppelin, but they failed to find the vulnerability, which implies the need of the rigorous formal verification.

[17] https://zeppelin.solutions/security-audits

5 RELATED WORK

While there exist several static analysis tools [5, 6, 8, 12] tailored to check certain predefined properties, here we consider, due to space limit, only the verification tools backed by a full-fledged theorem prover that allows to reason about arbitrary (full functional correctness) properties. Specifically, Bhargavan et al. [2] and Grishchenko et al. [3] presented a verification tool based on the F* proof assistant, and Amani et al. [1] presented a tool based on Isabelle/HOL. These tools, however, adopt only a partial, incomplete semantics of EVM, and thus may miss certain critical corner-case behaviors of the EVM bytecode, which could undermine the soundness of the verifiers. Our EVM verifier, on the other hand, is a verification tool derived from a *complete* and *thoroughly tested* formal semantics of EVM [4], for the first time to the best of our knowledge.

ACKNOWLEDGMENTS

The work presented in this paper was supported in part by NSF grant CCF-1421575, NSF grant CNS-1619275, and an IOHK gift.

REFERENCES

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM International Conference on Certified Programs and Proofs (CPP 2018)*.

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS 2016)*.

[3] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST 2018)*.

[4] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF 2018)*.

[5] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*.

[6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*.

[7] Yuri V. Matiyasevich. 1993. *Hilbert’s Tenth Problem*. MIT Press.

[8] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018).

[9] Grigore Rosu. 2017. ERC20-K: Formal Executable Specification of ERC20. https://github.com/runtimeverification/erc20-semantics.

[10] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*.

[11] Formal Verification Team. 2018. Verified Smart Contracts. https://github.com/runtimeverification/verified-smart-contracts/.

[12] Petar Tsankov, Andrei Marian Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *CoRR* abs/1806.01143 (2018).

[13] Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.