Using a Semantic Framework for the Security Analysis of Ethereum smart contracts

Illya Grishchenko
Matteo Maffei
Clara Schneidewind

An IFIP SELECT paper for IT-professionals based on the research paper:

Publisher’s information: https://link.springer.com/chapter/10.1007/978-3-319-89722-6_10#citeas

1. Introduction

Smart contracts are distributed programs running on a blockchain and controlling money flows within a cryptocurrency. The concept of such self-enforcing contracts that do not rely on any trusted third party gives rise to many practical applications such as auctions [7], data management systems [1], trading platforms [11, 13], permission management [2] or verifiable cloud computation [4]. Due to its significant practical implications, not only the industry recently showed increased interest in the topic, but also academia started to study the usage and the security of smart contracts. This is particularly relevant as the financial nature of smart contracts makes them a desired target for several attacks that exploit bugs in smart contracts for financial benefits or damage. Most prominently, attacks on smart contracts for the cryptocurrency Ethereum have caused users of the system to lose several millions of dollars in the last years [17, 20]. Such bugs are particularly devastating as, due to the immutable nature of the blockchain, they cannot be dynamically patched when they occur.

Motivated by this threat scenario, our work presents a systematic study of the semantic behavior of EVM bytecode – the low-level language in which Ethereum smart contracts are deployed. We define a complete and formal EVM bytecode semantics which we also implement in the proof assistant F* [19]. In order to give better insights into the subtleties of the presented semantics and the common misconceptions of developers concerning smart contract execution behavior, we present semantic characterizations of generic security properties that are motivated by real-world smart contract bugs.
Semantic Framework in Practice

We believe that a semantic framework for Ethereum smart contracts is of interest to those IT professionals who deal with developing, maintaining, reviewing, and analyzing distributed smart contract applications for the cryptocurrency Ethereum.

2. Practical Relevance

For exemplifying the immediate practicality of our framework, we will give in the following two concrete usage scenarios:

Smart Contract Auditing

First, consider an IT professional who is in charge of auditing a security-critical smart contract. As the presented semantics is formalized in the proof assistant F* [19], the framework allows for characterizing the contract’s behavior by logical statements on the smart contract execution. Those statements can then be formally proven with the help of the proof assistant, ensuring the correctness of the proof and at the same time, providing a certain degree of proof automation that facilitates the proving task for the user. Even though such a proof admittedly requires some technical expertise, it ensures a degree of certainty on the contract’s correctness that could never be met by a purely manual audit. Hence proving specific particularly critical smart contracts correct in the presented framework can add a new assurance layer to the auditing process.

Proving local optimizations correct

Ethereum smart contracts differ in their execution from regular programs in that they are bounded by a specific resource called *gas*. The idea behind the gas mechanism is to make the initiator of a transaction pay for the computational cost caused by the subsequent code execution and to thereby ensure the execution’s termination. As gas ultimately corresponds to monetary cost, there is an immediate economic incentive for smart contract programmers to reduce the number of computation steps in the execution of their smart contracts and also to shorten their size in order to decrease the price for the deployment. In practice, it is hence beneficial to decrease the number of operations (which correspond to bytecodes) or to replace costly operations by cheaper ones. However, a smart contract is generally written in high-level languages, such as Solidity [22], and then compiled to EVM bytecode. This often results in unnecessarily expensive smart contracts in bytecode format. For a smart contract developer, it is hence very interesting to manually optimize the output from the Solidity compiler. When doing so, however, it needs to be ensured that the resulting code replacement does not result in a change of the contract’s behavior. Again, the semantic framework can help to ensure that such modifications are semantics preserving. For example, a programmer could conduct small and simple proofs showing that the replaced code fragments locally show the same behavior.
3. Utilization

In the previous subsection, we discussed the immediate practical impact of the semantic framework presented in our paper. Next, we want to assess the long-term impact of our work on solving the existing problems in smart contract usage, development, and auditing.

As motivated in the introduction, the development of a smart contract is error-prone. This is not only problematic because smart contract bugs can lead to immediate financial losses, but because on account of this, the general trust in smart contract platforms is jeopardized, and therefore, their large-scale deployment is hindered. The appealing idea behind smart contracts of having self-executing contracts whose behavior is only governed by public, pre-determined code is reduced to absurdity if contract creators fail to make their code express the conditions that they have in mind or if contract participants cannot recover the contract's meaning from its code. De facto, transparency is lost, and the 'code is law'-principle\(^1\) only plays into the hands of very few experts who have sufficient knowledge of the underlying language and who can easily misuse this knowledge for exploiting buggy smart contracts or for creating malicious smart contracts with hidden, harmful behavior.

For increasing the trust in smart contracts, it is therefore essential to provide developers and users with tools for validating existing code and for creating secure smart contracts from ground up.

Checking existing code

For checking existing code, no matter whether this code stems from the blockchain or whether it was self-created (e.g., by compilation from Solidity), users need tool assistance. This is as a smart contract’s behavior is ultimately determined by the barely human-readable bytecode deployed on the blockchain. For making this application scenario of our work more concrete, we shortly discuss two categories of supportive tools and how they can profit from our semantic framework:

Code exploration tools.

Tools for code exploration assist users in manually investigating a smart contract’s structure and semantics and encompass, e.g., visual debuggers [12] and decompilers [5, 16]. Such tools help to recover the contract’s meaning in a form that is more accessible to humans and are thereby of particular interest when users are confronted with somebody else’s code (whose sources are potentially not available). An exploration tool though, can only give meaningful guidance if the provided insights reflect the correct semantics of the underlying EVM bytecode.

\(^1\) The term 'code-is-law' was initially introduced by the law professor Lawrence Lessing [10] for characterizing the rules of cyberspace and their social implications. Nowadays it is heavily adapted by the blockchain community to characterize the self-enforcing and immutable character of smart contracts.
**Code verification tools.**

The aim of verification tools is to prove specific (semantic) properties of a smart contract. The formal verification of smart contracts is particularly important in the development of security-critical contracts and such contracts that are meant to control significant money quantities. The spectrum of possible verification tools is broad: they range from formalizations in proof assistants (as provided in our work or in [8, 9]) – which allow for semi-automated proofs of hand-crafted contract specifications – to fully automated tools of generic security properties that do not require any user-interaction [18, 21]. In general, there is always a trade-off in the time and expertise required from a user to conduct a formal security (or correctness) proof and the coverage of contract-relevant properties that can be accomplished with the help of a verification tool. We believe that over the long term the goal should be to have tools in all areas of this spectrum which complement each other and that such tools should all be built on top of a solid semantic foundation (meaning that their produced results should be provably connected to a faithful execution model of EVM bytecode) – a property that most current works are lacking. With our work we want to provide such a foundation and with the conducted study on generic security properties we particularly want to lead the development of fully automated tools for proving such generic properties: Such automated tools have the advantage that they are easily accessible to all users of Ethereum smart contracts and not only to those with a background in conducting formal proofs. In line with this, we already made a first step towards developing an automated EVM bytecode analysis tool which can be proven sound with respect to the semantics presented in this work [6].

**Writing secure code**

For enabling the creation of secure smart contracts on a long-term basis, we believe that the development of new languages that are designed with security in mind is necessary. Such languages could distinguish themselves by stronger type systems, or language constructs that are specific to non-standard smart contract functionality. In the last years, quite some progress in this area has been made [3, 14, 15]. For making such advances meaningful, however, we think that it is of utmost importance that the developed languages have (in contrast to Solidity) a clearly defined semantics as well as a verified compilation to EVM bytecode. Without a formal compiler correctness proof, arguing about programs in terms of their high-level language semantics will never yield the level of certainty which is needed in the context of smart contracts. Having a clearly defined semantics of the compilation target (here EVM bytecode) is a prerequisite for conducting such a proof in the first place.

**References**


