Solidity's SMT-based formal verification of Ethereum smart contracts

Leonardo Alt and Christian Reitwiessner

No Institute Given

Abstract. This is an abstract.

1 Introduction

This is an introduction [1,3,2].

Cite other papers in the area.

We show how the Solidity compiler utilizes an SMT solver to verify conditions that can be specified as part of the source code. The main advantage of this system over others for the user is that they do not need to learn a second verification language. The Solidity language has an assertion construct that allows to check a condition at run-time. The verification component tries to verify that these conditions hold for any input.

2 Ethereum

The Ethereum platform is a system that appears as a singleton networked computer usable by anyone, but is actually built as a distributed database that utilizes blockchain technology to achieve consensus. One of the features that sets Ethereum apart from other blockchain systems is the ability to store and execute code inside this database. In contrast to traditional server systems, anyone can inspect this stored code and execute functions. Furthermore, blockchains typically store ownership relations of valuable goods (for example cryptocurrencies). Therefore, malicious actors have a monetary incentive to analyze the inner workings of such code. Because of that, testing (i.e. dynamic analysis of some typical inputs) does not suffice and analyzing all possible inputs by utilizing static analysis or formal verification is recommended.

3 Smart Contracts

Due to space constraints, we will only give a very brief description of smart contracts on Ethereum and leave out some details.

Programs in Ethereum are called *smart contracts*. Each smart contract is stored at its *address* with its *code* and a key-value store of data (*storage*). The code is fixed after the creation phase and only the smart contract itself can modify the data stored at its address.

Users can interact with a smart contract by sending a *transaction* to its address. This cases the smart contract's code to execute inside the so-called *Ethereum Virtual Machine* (EVM), which is a stack-based 256-bit machine with a minimalistic instruction set. Each execution environment has a freshly initialized *memory area* (not to be confused with the persisting storage). During its execution, a smart contract can also call other smart contracts synchronously, which causes their code to be run in a new execution environment. Data can be passed and received in calls. Furthermore, smart contracts can also create new smart contracts with arbitrary code.

We would like to highlight a special feature of the Ethereum Virtual Machine which will be essential for the design of the SMT encoding we will use. Whenever the EVM encounters an invalid situation (invalid opcode, invalid stack access, etc.), execution will not only stop, but all effects on the state will be reverted. This reversion takes effect in the current execution environment (i.e. roughly a function call), and the environment will also flag a failure to the calling environment (if present). Typically, high level languages will cause an invalid situation whenever a call fails and thus the reversion affects the whole transaction.

There is also an explicit opcode that causes the current call to fail, which is essentially the same as described above, but as an intended effect.

Very briefly, the SMT encoding assumes that no intended failure happens and tries to deduct that no unintended failure can occur. This allows the programmer to state preconditions using intended failures and postconditions using unintended failures.

4 Solidity

5 SMT Encoding

6 Conclusion

Outlook: Add storage variables which hold their state across function calls. This is more complicated, because typically, the smart contract is not designed to assume all possible states over its potential lifetime. Therefore, the system should be able to restrict the set of possible states by analyzing its code.

Apart from these explicit conditions, other conditions like overflows, divisions by zero or array access out of bounds (planned) are checkd.

References

1. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96. PLAS '16 (2016)

2. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers. pp. 520–535 (2017)
3. 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. CCS '16 (2016)