SMT-based compile-time verification of safety properties for smart contracts

Leonardo Alt, Alex Beregszaszi and Christian Reitwiessner

Ethereum Foundation
{`leo,axic,chris`}`@ethereum.org`

Abstract. Ethereum smart contracts are programs that run inside a public distributed database called a blockchain. These smart contracts are used to handle tokens of value, can be accessed and analyzed by everyone and are immutable once deployed. Those characteristics make it imperative that smart contracts are bug-free at deployment time, hence the need to verify them formally. In this paper we describe our current efforts in building an SMT-based formal verification module within the compiler of Solidity, a popular language for writing smart contracts. The tool is seamlessly integrated into the compiler, where during compilation, the user is automatically warned of and given counterexamples for potential arithmetic overflow/underflow, unreachable code, trivial conditions, and assertion fails. We present how the component currently translates a subset of Solidity into SMT statements using different theories, and discuss future challenges such as multi-transaction and state invariants.

1 Introduction

The Ethereum [11] 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, via the Ethereum Virtual Machine (*EVM*). 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.

SAT/SMT-based techniques have been used extensively for program verification [4,6,7,2,8,1]. This paper shows how the Solidity compiler which generates EVM bytecode utilizes an SMT solver and a Bounded Model Checking [4] approach to verify safety properties that can be specified as part of the source code, as well as fixed targets such as arithmetic underflow/overflow, and detection of unreachable code and trivial conditions. The main advantage of this system over others for the user is that they do not need to learn a second verification language or how to use any new tools, since verification is part of the compilation process. The Solidity language has requirement and assertion constructs that allow to filter and check a condition at run-time. The verification component tries to verify that the asserted conditions hold for any input, assuming the given requirements.

Section 2 introduces the EVM and smart contracts. Section 3 presents Solidity, the high-level language for smart contracts. Section 4 discusses the translation from Solidity to SMT statements and next challenges. Finally, Section 7 contains our concluding remarks.

Related work. Oyente [9] and Mythril [5] are SMT-based symbolic execution tools for EVM bytecode that check for specific known vulnerabilities, where the former also checks for assertion fails. They simulate the virtual machine and execute all possible paths, which takes a performance toll even though the approach works well for simple programs. Solidity has been translated to the frameworks Why3 [10] and F* [3], but the former requires learning a new annotation specification language and the latter only verifies fixed vulnerability patterns and does not verify custom user-provided assertions.

2 Smart Contracts

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.

⟨⟨**Talk about gas?**⟩⟩

We would like to highlight a special feature of the Ethereum Virtual Machine that makes the control-flow analysis different from other languages. 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.

⟨⟨Talk about "all paths that fail eventually revert"⟩⟩

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.

3 Solidity

⟨⟨Introduce Solidity⟩⟩
⟨⟨Talk about what it's commonly used for⟩⟩
⟨⟨Example of Solidity code, maybe ERC20?⟩⟩
⟨⟨Example of Solidity code with target safety properties, usage of require-assert⟩⟩

4 SMT Encoding

5 Implemented (or almost there)

⟨⟨Write about how we encode the control flow⟩⟩
⟨⟨Write about how we encode types (especially integers)⟩⟩
⟨⟨Write about what checks we perform (overflow, assert, require, unreachable, trivial condition)⟩⟩

Possible extension: Errors are irrelevant if they will result in a state reversion. So we could change the check to e.g. only flag overflows if they do not result in a state reversion. Popular example, the SafeMath contract:

`function add(uint256 a, uint256 b) internal pure returns (uint256) {`uint256 c = a + b; assert(c ¿= a); return c; }

6 Discussions about the future

⟨⟨Multi-tx state invariants, with an example, maybe a figure⟩⟩
⟨⟨Loops?⟩⟩
⟨⟨effective callback freeness⟩⟩

7 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. Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Hifrog: Smt-based function summarization for software verification. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 207–213. Springer (2017)
2. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification. pp. 184–190. Springer (2011)
3. 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)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer (1999)
5. ConsenSys: Mythril (2018), `github.com/ConsenSys/mythril`
6. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Static Analysis. pp. 351–368. Springer (2011)
7. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: Computer Aided Verification. pp. 846–862. Springer (2013)
8. Kroening, D., Tautschnig, M.: Cbmc – c bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014)
9. 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)
10. Why3: Why3 (2018), `why3.lri.fr`
11. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2018), `ethereum.github.io/yellowpaper/paper.pdf`