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 [15] 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 [5,7,8,3,9,1]. This paper shows how the Solidity compiler which generates EVM bytecode utilizes an SMT solver and a Bounded Model Checking [5] 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.

Sec. 2 introduces the EVM and smart contracts. Sec. 3 presents Solidity, the high-level language for smart contracts. Sec. 4 discusses the translation from Solidity to SMT statements and next challenges. Finally, Sec. 5 contains our concluding remarks.

Related work. Oyente [10] and Mythril [6] 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 [14] and F* [4], 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

Solidity is a programming language specifically developed to write smart contracts which run on the Ethereum Virtual Machine. It is a statically-typed curly-braces language with a syntax similar to Java. The main source code elements are called *contracts* and are similar to classes in other languages.

Smart contracts in Solidity can be used to enforce agreements between mutually distrusting parties as long as all conditions can be fully formalized and do not depend on external factors. Typical use-cases are decentralized tokens which can have a currency-like aspect, any mechanisms that build on top of these tokens like exchanges and auctions or also decentralized tamper-proof registry systems like a domain name system.

The following source code shows a minimal example of a token contract. Users are identifier by their addresses and initially, all tokens are owned by the creator of the contract, but anyone who owns tokens can transfer an arbitrary amount to other addresses. Authentication is implicit in the fact that the address from which a function is called can be accessed through the global variable `msg.sender`. In practice, this is enforced by checking a cryptographic signature on the transaction that is sent through the network.

```
contract Token {
    /// The main balances / accounting mapping.
    mapping(address => uint256) balances;

    /// Create the token contract crediting ‘msg.sender‘ with
    /// 10000 tokens.
    constructor() public {
        balances[msg.sender] = 10000;
    }
    /// Transfer ‘_value‘ tokens from ‘msg.sender‘ to ‘_to‘.
    function transfer(address _to, uint256 _value) public {
        require(balances[msg.sender] >= _value);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
    }
}
```

The `require` statement inside the function `transfer` is used to check a precondition at run-time: If its argument evaluates to false, the execution terminates and any previous change to the state is reverted. Here, it prevents tokens being transferred that are not actually available.

In general, invalid input should be caught via a failing `require`. A similar `assert` statement can be used to check postconditions. The idea behind is that it should never be possible to reach a failing assert. `assert` essentially has the same effect as `require`, but is encoded differently in the bytecode. Verification tools on bytecode level (as opposed to the high-level approach described in this article) typically check whether it is possible to reach an assert in any way.

We now show how an `assert` can be introduced into the `transfer` function to perform a simple invariant check.

```
function transfer(address _to, uint256 _value) public {
    require(balances[msg.sender] >= _value);
    uint256 sumBefore = balances[msg.sender] + balances[_to];
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    uint256 sumAfter = balances[msg.sender] + balances[_to];
    assert(sumBefore == sumAfter);
}
```

The `assert` checks that the sum of the balances in the two accounts involved did not change due to the transfer. Currently, the `assert` statement is not removed by the compiler, even if the formal analysis module can prove that it never fails.

Note that in the general case, `balances[_to]` can overflow and thus an analysis tool might flag this assert as potentially failing. In this specific example, though, the amount of available tokens is too small for this to happen.

4 SMT Encoding

4.1 Implemented (or almost there)

The SMT encoding is computed in a top-down way traversing the AST of the Solidity program. The context regarding the SMT solver, contract storage, and local variables of functions is cleared before each function of a contract is visited.

Inside each function, a variable declaration leads to a correspondent SMT variable that is assigned the default value of the declared type. Function parameters are initialized with a range of valid values for the given type, since their value is unknown. For instance, a parameter `uint32 x` is initialized as $0 \leq x < 2^{32}$ (32 bits), and a parameter `address a` is assigned the range $0 \leq a < 2^{(8*20)}$ (20 bytes). The encoding of variable assignment follows the *Single Static Assignment* (SSA) where each assignment to a program variable introduces a new SMT variable that is assigned to only once. The encoder currenty supports *Bool* and the various sizes of *Integer* variables.

Because the Solidity code is visited via its AST, we can maintain a stack of conditions that lead to the piece of code being analyzed: whenever an *if-statement* is visited its condition is pushed onto the stack, the branch's body is encoded and the condition is popped again. Similarly, the negation of the condition is pushed onto the stack when visiting an *else* branch, and popped afterwards. Control flow is then encoded using the *if-then-else* operator (ite) as the SSA ϕ function, with the head of the stack as the condition of the if-then-else expression.

Every arithmetic operation is checked against underflow and overflow according to the type of the values, and an example is given if there is an underflow or overflow. We also check whether branch conditions are constant, warning the user about unreachable blocks or trivial conditions. A `require`'s condition is checked for triviality and reachability. This is important since `require` conditions are meant to be used as filters for unwanted input values when they are unknown, for example, in public functions. Therefore they act like preconditions for the rest of the scope, implied by the conjunction of the conditions from the conditions stack. Finally, `assert` conditions represent target postconditions that the Solidity programmer wants to ensure at runtime and are verified statically. If it is possible to disprove the assertion, the user is given a counterexample.

Figure 4.1 shows a Solidity sample and its SMT encoding with emphasis on the control flow handling while verifying an assertion. Since variables `uint a` and `uint b` are function parameters, they are initialized with the valid range of values for their type (`uint256`). The requirement condition about `b` is only true if the first condition about `a` is true. The next two assignments to `b` create the new SSA variables b_1 and b_2. Variable b_3 encodes the second and third conditions, and b_4 encodes the first condition. Finally, b_4 is used in the assertion check. Note that the nested control-flow is implicitly encoded in the *ite* variables b_3 and b_4. We can see that the target assertion is safe within its function.

```
contract C
{
  function f(uint a, uint b) {
    if (a == 0)
      require(b <= 100);
    else if (a == 1)
      b = 1000;
    else
      b = 10000;
    assert(b <= 100000);
  }
}
```

$$
\begin{aligned}
&a_0 \geq 0 \wedge a_0 < 2^{256} \wedge \\
&(a_0 = 0) \rightarrow (b_0 \leq 100) \wedge \\
&b_1 = 1000 \wedge b_2 = 10000 \\
&b_3 = ite(a == 1, b_1, b_2) \wedge \\
&b_4 = ite(a == 0, b_0, b_3) \wedge \\
&\neg b_4 \leq 100000
\end{aligned}
$$

Fig. 1. SMT encoding of an assertion check highlighting control-flow.

It is important to highlight that errors are irrelevant if they result in a state reversion (Sec. 2). The user is warned about checks such as overflow only if they do not result in a state reversion. One popular example is the SafeMath [12] contract:

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

Although the tool sees an overflow in the computation of `a + b`, since both have type `uint256` which is the largest integer, any execution that containts the overflow reverts due to the `assert`. In this case the user is not warned of the error, since no erroneous cases exist in accepted executions.

As described above, the component performs several local checks during a single run, therefore it is critical that the used SMT solver has support to incremental checking. Moreover, we do not abstract difficult operations such as multiplication between variables, rather trying to give precise answers when possible. Therefore we combine various quantifier-free theories, such as Linear Arithmetics, Uninterpreted Functions and Nonlinear Arithmetics. Solidity has integrated Z3 [11] and CVC4 [2] via their C++ APIs. The two SMT solvers are used together to increase solving power. This has been important especially for the programs that require Nonlinear reasoning, since often one solver is able to prove a property that the other cannot. In case both Z3 and CVC4's APIs are not available but binaries are, the component generates `smtlib2` [13] formulas and invokes the solver binaries.

4.2 Discussions about the future

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

5 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. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer Aided Verification. pp. 171–177. Springer (2011)
3. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification. pp. 184–190. Springer (2011)
4. 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)
5. 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)
6. ConsenSys: Mythril (2018), `github.com/ConsenSys/mythril`
7. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Static Analysis. pp. 351–368. Springer (2011)
8. 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)
9. 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)
10. 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)
11. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
12. OpenZeppelin: Safemath (2018), `github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol`
13. SMT-LIB: Smt-lib (2018), `smtlib.cs.uiowa.edu`
14. Why3: Why3 (2018), `why3.lri.fr`
15. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2018), `ethereum.github.io/yellowpaper/paper.pdf`