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 [6] 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 that can have stateful effects. Since blockchains are typically used to store ownership relations of valuable goods (for example cryptocurrencies), 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 [5,8,11,3,12,1]. 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, division by zero and detection of unreachable code and trivial conditions. For the user, the main advantage of this system over others 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 conditions at run-time. The verification component builds on top of this and tries to verify at compile-time that the asserted conditions hold for any input, assuming the given requirements.

Sec. 2 introduces the EVM and smart contracts. Sec. 3 gives a very brief overview of Solidity. Sec. 4 discusses the translation from Solidity to SMT statements and next challenges. Finally, Sec. 5 contains our concluding remarks.

Related work. Oyente [13], Mythril [7] and MAIAN [15] are SMT-based symbolic execution tools for EVM bytecode that check for specific known vulnerabilities, where Oyente 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.

Subsets of Solidity have been translated to Why3 [18], F* [4] and LLVM [10], but the first requires learning a new annotation specification language and the latter two only verify fixed vulnerability patterns and do not verify custom user-provided assertions.

2 Smart Contracts

Programs in Ethereum are called *smart contracts*. They 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.

Each smart contract has an *address* under which its *code* and a key-value store of data (*storage*) is stored. 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 causes 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.

Since it would otherwise be easy to stall the network by asking it to execute a complex task, the resources consumed are metered during execution in a unit called *gas*. Each transaction only provides a certain amount of gas, and as soon as this *gas limit* is reached without prior termination, any effect of the transaction is reverted. In every case, the user who requested the execution pays for it with Ethereum's native token Ether proportionally to the amount of gas consumed.

A reverting termination can also happen prior to all gas being consumed. This is a special feature of the Ethereum Virtual Machine, which 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, and the environment will also flag a failure to the calling environment, if present. Typically, when a call fails, high level languages will in turn cause an invalid situation in the caller 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. This leads to the important observation that errors might be detected but they are only relevant if a path that reaches the error does not lead to the transaction being reverted.

Very briefly, the SMT encoding we will discuss later 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. Contract-level variables in Solidity are persisted in storage while local variables and function parameters only have a temporary lifetime. Among others, Solidity has integer data types of various sizes (up to 256 bits, the word size of the EVM), address types and an associative array type called *mapping* which can only be used for contract-level variables.

The source code in Fig. 1 shows a minimal example of a token contract. Users are identified 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.

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`. The related `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

```
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;
    }
}
```

Fig. 1. Example of a token contract

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-based Solidity verification

SMT solvers are powerful tools to prove satisfiability of formulas in different logics which often have the necessary expressiveness to model software in a straightforward manner [11,1,8,3].

We translate Solidity contracts and their functions into SMT formulas using a combination of different quantifier-free theories. We shall name the translated formulas the *SMT encoding* of the Solidity program. The goal of the translation from Solidity to SMT formulas is to verify safety properties from the Solidity program by performing queries to the SMT solver.

4.1 SMT encoding

The SMT encoding is computed during a depth-first traversal of the abstract syntax tree (AST) of the Solidity program and thus roughly follows the execution order. For now, each function is analyzed in isolation and thus the context regarding the SMT solver (contract storage, local variables, etc.) is cleared before each function of a contract is visited. There are four types of formulas that are encoded from Solidity inside each function. Three of them, *Control-flow*, *Type constraint* and *Variable assignment* are simply translated as SMT constraints. The last, *Verification Target*, creates a formula consisting of the verification goal conjoined with the previously mentioned constraints, and queries the SMT solver for satisfiability. The different types of encoding are described below.

Control-flow. Since control-flow does not have to be monotonous, we cannot just add constraints while traversing the AST. We will define a set of control-flow constraints which are satisfiable if and only if the Solidity statement currently under consideration is reachable. Let $R_{\text{pre}}(s)$ be the constraint set to reach the statement s and $R_{\text{post}}(s)$ be the constraint set for control-flow going past n. Naturally, $R_{\text{post}}(s) = R_{\text{pre}}(t)$ if t follows s. Also, $R_{\text{post}}(s) = R_{\text{pre}}(s)$ for a regular statement s.

For s being `require(`r`)` (and similar for `assert(`r`)`), control flow terminates if r evaluates to false, and thus $R_{\text{post}}(s) = R_{\text{pre}}(s) \wedge r$.

For s being `if (`c`)` T `else` F, we have $R_{\text{pre}}(T) = R_{\text{pre}}(s) \wedge c$ and $R_{\text{pre}}(F) = R_{\text{pre}}(s) \wedge \neg c$. Finally, $R_{\text{post}}(s) = R_{\text{post}}(T) \vee R_{\text{post}}(F)$.

The implementation utilizes a stack to realize if-statements.

Type constraint. A variable declaration leads to a correspondent SMT variable that is assigned the default value of the declared type. For example, Boolean variables are assigned false, and integer variables are assigned 0. 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 encoder currently supports Boolean and the various sizes of Integer variables.

Variable assignment. 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. When a program variable is modified at a place that is executed only conditionally (if-else), the *if-then-else* operator (ite) is used as the common SSA ϕ function associated with the branch condition to create a new SMT variable after the branch blocks.

Verification target. 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. The conditions in calls to `assert` represent target postconditions that the Solidity programmer wants to ensure at runtime and are verified statically. If it is possible to disprove the assertion provided that the control flow can reach it, the user is given a counterexample. In contrast, `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. Failing calls to `require` are not treated as errors and are just checked for triviality and reachability.

Figure 4.1 shows on the left a Solidity sample that requires all four types of encoding, shown on the right, in order to verify the intended properties. Since the variables `uint256 a` and `uint256 b` are function parameters, they are initialized (lines 1 and 2) with the valid range of values for their type (`uint256`). If `a = 0`, the `require` condition about `b` is used as a precondition when verifying the assertion in the end of the function (line 3). The next two assignments to `b` create the new SSA variables b_1 and b_2 (line 4). Variable b_3 encodes the second and third conditions, and b_4 encodes the first condition (lines 5 and 6). Finally, b_4 is used in the assertion check (line 7). 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(uint256 a, uint256 b)
  {
    if (a == 0)
      require(b <= 100);
    else if (a == 1)
      b = 1000;
    else
      b = 10000;
    assert(b <= 100000);
  }
}
```

1. $a_0 \geq 0 \wedge a_0 < 2^{256} \wedge$
2. $b_0 \geq 0 \wedge b_0 < 2^{256} \wedge$
3. $(a_0 = 0) \rightarrow (b_0 \leq 100) \wedge$
4. $b_1 = 1000 \wedge b_2 = 10000$
5. $b_3 = ite(a == 1, b_1, b_2) \wedge$
6. $b_4 = ite(a == 0, b_0, b_3) \wedge$
7. $\neg b_4 \leq 100000$

Fig. 2. SMT encoding of an assertion check.

It is important to highlight that errors are irrelevant if they result in a state change 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 [16] contract which is commonly used to turn wrapping arithmetics into overflow-checked arithmetics:

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

Although the tool detects an overflow in the computation of `a + b`, the overflow will result in a truncation of `c` in two's complement and thus any execution that contains the overflow will revert at the `require`. 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 supports 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 [14] 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. The component is also able to generate `smtlib2` [17] formulas in order to interface with additional solvers.

4.2 Future plans

Our current implementation plans for the component involve supporting a larger subset of the language, including more complex data structures such as `mapping`. We intend to build the component as a Bounded Model Checker, unrolling loops up to a constant bound and automatically detecting bounds when possible. This feature is currently being implemented. ⟨⟨**I don't think we should be that specific about the progress of the project. Earlier we already mentioned things that are not yet implemented.**⟩⟩

One of the most interesting aspects we intend to research and support is multi-transaction invariants. The ultimate goal is to compute invariants for state variables (resident in the contract's storage) considering any arbitrary number of calls to the contract. This would enable these invariants to be used as preconditions whenever they are accessed. As an example, take contract `Token` from Sec. 3. We can see from the constructor that the deployer of the contract receives 10000 tokens. The only way to move tokens is via the function `transfer`, which decreases a certain amount of tokens from one account, if it owns enough, and increases the same amount in another account. As we can see, the number of total tokens never changes and the invariant

$$\sum_{a \in \text{balances}} \text{balances}[a] = 10000$$

holds in the beginning of any function of the contract.

Function *modifiers* are Solidity constructs that are used as patterns to change the behavior of functions, and in many cases, to restrict them. Commonly used modifiers are, for example, allowing only the owner of the contract to execute the function, or executing a function if and only if the amount of Ether sent is greater than a certain value. We intend to use properties inferred from modifiers as function preconditions, therefore improving proving power.

The idea of *Effective Callback Freeness* was recently introduced by [9]. A smart contract C is effectively callback free, if any state change caused by an internal callback can also be caused by an execution that does not have this callback. The authors show that most of the contracts deployed on Ethereum have this property. This is a powerful property, since it means that any invariant computed for a contract's state variables still holds even after calling external contracts with unknown behavior. We intend to study how to integrate this approach to our static analysis.

5 Conclusion

We have presented our work in progress building an SMT-based formal verification module inside the Solidity compiler. The component creates SMT constraints from the Solidity code and queries to statically check for underflow/overflow, division by zero, unreachable/trivial code, and assertion fails. The programmer receives, in compile-time, feedback with counterexamples in case any of the target properties fail, without any extra effort.

Our under implementation features intend to extend the subset of Solidity that is supported, and future work includes interesting research questions, such as computing multi-transaction invariants for state variables, and using properties from assertions at the end of functions or from modifiers to compute generic invariants, as discussed in Sec. 4.

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. Buterin, V.: A next-generation smart contract and decentralized application platform (2014), github.com/ethereum/wiki/wiki/White-Paper
7. ConsenSys: Mythril (2018), github.com/ConsenSys/mythril
8. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Static Analysis. pp. 351–368. Springer (2011)
9. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. 2(POPL), 48:1–48:28 (2017)
10. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing safety of smart contracts (2018)
11. 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)
12. 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)
13. 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)
14. 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)
15. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. CoRR abs/1802.06038 (2018), http://arxiv.org/abs/1802.06038
16. OpenZeppelin: SafeMath (2018), github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol
17. SMT-LIB: SMT-LIB (2018), smtlib.cs.uiowa.edu
18. Why3: Why3 (2018), why3.lri.fr