SMT-based Verification of Solidity Smart Contracts

Leonardo Alt and Christian Reitwiessner

Ethereum Foundation
{leo,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.

This paper is organized as follows: 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, among other things, its *code*, and a key-value store of data (*storage*) are 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, which acts as a *gas limit*. If execution is terminated via the *stop* instruction, any remaining

gas is refunded and the transaction is successful. However, if an exceptional condition or this *gas limit* is reached without prior termination, any effect of the transaction is reverted and it is marked as a failure. 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. 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

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

    /// Create the token contract crediting ‘msg.sender‘ with
    /// 10000 tokens.
    constructor() public {
        totalSupply = 10000;
        balances[msg.sender] = totalSupply;
    }

    /// 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.

it should never be possible to reach a failing assert. `assert` essentially[1] 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.

[1] As opposed to `require`, `assert` will result in all remaining *gas* to be consumed.

Another important feature that we refer to later in this paper are *function modifiers*. These 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. Fig. 2 shows a contract using the former, where the execution of function `f` continues if and only if the original deployer of the contract is the caller. We discuss later how to use modifiers to represent function pre- and postconditions.

```
contract C
{
  address owner;

  // A function using this modifier will be executed only
  // if the require condition holds.
  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }

  // Create the contract setting the deployer as owner.
  constructor() public {
    owner = msg.sender;
  }

  function f() onlyOwner {
    ...
  }
}
```

Fig. 2. Example of modifiers.

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 five 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 *Branch conditions* are the conditions of the current branch of execution and thus grow and shrink as we traverse the AST. The last, *Verification Target*, creates a formula consisting of the verification goal conjoined with the previously mentioned constraints, including the current branch conditions, and queries the SMT solver for satisfiability. The different types of encoding are described below.

Branch conditions. For an if-statement `if (`c`)` T `else` F, we add c to the branch conditions during the visit of T. After that, we replace c by $\neg c$ for the visit of F and also remove that when we are finished with the if-statement.

Control-flow. These constraints model conditional termination of execution. A `require(`r`)` statement (and similar for `assert(`r`)`) terminates execution if r evaluates to false, but of course only if it is executed. Thus, we add a constraint $b \rightarrow r$, where b is the conjunction of the current branch conditions. Note that due to the implication, we can keep this constraint even when we leave the current branch.

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), a parameter `int256 y` is initialized as $-2^{255} \leq y < 2^{255}$, 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 a 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 inside different branches of execution, a new variable is created after the branch to re-combine the different values after the branches. We use the if-then-else-function `ite` to assign the value `ite(c,` x_1`,` x_2`)` (if-then-else), where c is the branch condition and x_1 and x_2 are the two SSA variables corresponding to x at the ends of the branches (cf. the ϕ function in SSA).

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 (i.e. the current branch conditions are satisfiable), 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, acting like preconditions for the rest of the scope. Therefore, 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 five 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. 3. SMT encoding of an assertion check.

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, and rather try 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 Specific Examples

Even though the current implementation of the SMT module supports a small subset of Solidity, it can already be used to detect flaws that might be overlooked by the user. We present now a few examples of buggy code that the compiler is able to detect regarding constant conditions, overflow, and assertion checking.

The following loop is infinite because the author of the code forgot to increment the loop variablee `i`. In that case, the user receives a message about the loop's condition being always true for the case where `owners.length` is not zero.

```
for (uint i = 0; i < owners.length;)
{
    // ...
}
```

Another type of problem that the compiler finds automatically is unreachable code. In the following control-flow expressions, it warns the user that the condition in the `else if` is unreachable.

```
if (a >= 7) { ... }
else if (a >= 10) { ... }
```

Arithmetic operations should be checked against overflow, especially when parameters of public functions are used. The code below may easily lead to an overflow, which the tool reports with a counterexample. The overflow can be prevented with a `require` statement.

```
function addFunds(uint256 _amount) {
  // require((_amount + funds) >= funds);
  funds += _amount;
}
```

One of the most important features is the ability to check safety properties statically, by using Solidity's `assert`. The following example code uses an `assert` to check the equivalence of two computations, once written using control-flow statements, once as a direct Boolean formula.

```
function f(bool a, bool b) public pure {
  bool c;
  if (a) {
    if (b) c = false;
    else c = true;
  }
```

```
  else {
    if (b) c = true;
    else c = false;
  }
  assert(c != ((a && !b) || (!a && b)));
}
```

Note that the assertion will be reported to fail with the valuation `a = false, b = false, c = false`. The safe condition would be `assert(c == ((a && !b) || (!a && b)));`.

4.3 Future Plans

We introduce now the features that we intend to implement in the SMT module, as well as discuss arising research problems where we present simple examples that highlight how the new features will work.

Our current implementation plans for the component involve supporting a larger subset of the language, including more complex data structures such as `mapping`. This is especially important for cases such as token contracts, where properties such as funds leakage and wrong balance could be used as targets. The component is meant to be built as a Bounded Model Checker, unrolling loops up to a constant bound and automatically detecting bounds when possible. We also intend to introduce a loop pre and postconditions syntax to help the unbounded case.

Range restriction of real life values. Some Solidity environment variables have a 256 bit unsigned integer type, although the range of their values is much more restricted in practice. For instance, the UNIX timestamp of the current block in seconds, `block.timestamp` will not exceed 64 bits for the next 500 billion years. To reduce the false positives rate for overflows, it makes sense to restrict the value range for these variables in the SMT encoding. It is an open question how to do this properly, since a straightforward hard cap at some point could create undesired artefacts around that point. Another environment variable that could have a similar behavior is `block.number`.

Revert after Error. Errors are irrelevant if they result in a state change reversion (Sec. 2). The user should be warned about failing 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 should not be warned of the error, since no erroneous cases exist in accepted executions.

Aliasing. In many languages, complex data structures are only assigned by reference, creating two names for the same object and thus changes performed via one name also affect references via the second name. This is of course a big challenge for formal verification and is known as the *aliasing problem*. This is also the case for some aspects of Solidity, but data stored in storage does not have this problem: The structure of storage is determined at compile-time, and all objects are statically allocated; while arrays can grow, their position in storage is fixed at compile-time. Because of that, the aliasing problem is not an issue, as long as we can assume that there are no hash collisions in keccak256 and dynamic arrays are small enough.

```
contract C
{
  uint a;

  constructor () public {}

  function a1() public { a = 1; }
  function a2() public { a = 2; }
  function a3() public { a = 3; }
  function a4() public { a = 4; }

  function plusA(uint x) public view returns (uint) {
    require(x < 1000);
    return a + x;
  }
}
```

Fig. 4. Contract with a storage variable invariant.

Multi-transaction invariants. 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. Fig. 4 presents an example contract with a state variable `a` which can be assigned differently depending on which public function is called. We can see that if we consider all possible paths, `a` is never greater than 4, so the invariant $a \leq 4$ holds. Currently, without

the discovery of the invariant, the SMT module reports an overflow case in the `return` statement of function `plusA`. If the invariant is used as a pre-condition of the function, by adding `require(a <= 4)`, for example, no overflow is reported. The SMT component should in the future be able to automatically infer these invariants.

Post-constructor invariants. A special and restricted case of multi-transaction invariants usage are contracts where a state variable is assigned in the constructor and never modified again. A common example is contract `Token` from Sec. 3. We can see from the constructor that the `totalSupply` of tokens is 10000, which is also the initial amount of tokens given to the deployer of the contract. 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. We can modify function `transfer` to use the invariant about state variable `totalSupply`:

```
function transfer(address _to, uint256 _value) public {
    require(balances[msg.sender] >= _value);
    uint256 sumBefore = balances[msg.sender] + balances[_to];
    totalSupply -= sumBefore;
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    uint256 sumAfter = balances[msg.sender] + balances[_to];
    totalSupply += sumAfter;
    assert(sumBefore == sumAfter);
    assert(totalSupply == 10000);
}
```

As we can see, the number of total tokens never changes and the invariant totalSupply = 10000 holds in the beginning of any function of the contract. Similarly to the previous example, it is not possible to prove the last assertion without the knowledge about the invariant.

Modifiers as pre and postconditions. An orthogonal approach to automatically inferred invariants is to provide a good syntax so that Solidity programmers can explicitly state pre and postconditions of functions. Modifiers (Sec.3) are a natural candidate for that, given their ability to behave as patterns that wrap functions. In the following code, the modifier `safeBalance` states pre and postconditions for the `transfer` function in the `Token` contract (Sec. 3), ensuring that the concrete value of `totalSupply` does not change after a token transfer.

```
modifier safeBalance {
  require(totalSupply == 10000);
  _;
  assert(totalSupply == 10000);
}
```

```
function transfer(address _to, uint256 _value) safeBalance {
  ...
}
```

Function abstraction. If modifiers are used as pre and postconditions as described above, it could be possible to abstract functions based on these modifiers. Let `zeroAccount` be a function from contract `Token` that transfers all the tokens that an account holds to another one of their choice. Function `zeroAccount` should also be sure that the `totalSupply` did not change.

```
function zeroAccount(address _to) {
  transfer(_to, balance[msg.sender]);
  assert(totalSupply == 10000);
}
```

One approach to analyze `zeroAccount` is to abstract function `transfer` by encoding only its modifiers and ignoring its body when trying to prove the assertion. This query is much cheaper for the SMT solver, and in many cases (as it is in this one) it might be enough to prove the assertion.

Effective Callback Freeness. 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 a callback in C can also be caused by an execution that does not have this callback. Straightforward examples include a contract that uses a mutex mechanism to disallow state changes if the function is called as a callback, and the general pattern where all functions perform state changes before they call other contracts. 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 current work and future plans building an SMT-based formal verification module inside the Solidity compiler. The module creates SMT constraints from the Solidity code and queries SMT solvers to statically check for underflow/overflow, division by zero, unreachable/trivial code, and assertion fails, where require statements are used as assumptions. The programmer receives, in compile-time, feedback with counterexamples in case any of the target properties fail, without any extra effort. The SMT constraints and queries are created using theories that model the Solidity program precisely, therefore the given counterexamples are correct.

The features that are currently under implementation aim at extending the subset of Solidity that is supported, as well as improving error reporting. Future

work on the SMT module includes interesting broader research questions, such as computing multi-transaction invariants for state variables, detecting post-constructor invariants, and using modifier-based abstraction for functions.

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