Verification of Smart Contracts on the Ethereum Blockchain

HENRY FISHER, Grinnell College, United States
PETER-MICHAEL OSERA, Grinnell College, United States

ACM Reference Format:
Henry Fisher and Peter-Michael Osera. 2018. Verification of Smart Contracts on the Ethereum Blockchain. 1, 1 (October 2018), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rise in popularity of the Ethereum blockchain poses some interesting challenges for programmers, in particular because they implement smart contracts.

Smart contracts were first formalized by Nick Szabo in 1997 [Szabo 1997]. The basic idea is that one formally specifies the terms of a contract digitally, and then outsources its execution (and in some cases its verification) to a program. Insurance policies, for example, are particularly well suited to smart contracts. As long as the condition for insurance can be formally specified (e.g., the presence of an earthquake), the policy can be written up such that it executes immediately upon verifying its condition.

This has a number of advantages over traditional contracts. First, the execution of the contract is entirely trustless. Contracts, at their core, are simply methods of lowering the trust required for two parties to enter mutually beneficial circumstances. Enforcement via contract law allows for very little trust required between the parties, but smart contracts lower the trust requirement to zero for cases where a smart contract can actually be made. Second, smart contracts can be set up to execute instantly, meaning bureaucratic delay of contract execution can be eliminated entirely. Last, smart contracts can anonymize information, or at least remove the need for third parties to know about the circumstances of the contract.

The Ethereum blockchain [Buterin 2013] [Wood 2018] has become one of the most popular implementations of smart contracts. In addition to normal user addresses where one can send ether (the currency of the Ethereum blockchain), Ethereum also has smart contract addresses. One can send ether and arbitrary data (i.e., parameters) to a smart contract address and the contract will then execute according to how it was written. To write contracts, Ethereum has a built in Turing complete programming language.

However, as Ethereum's popularity has grown, a number of problems with its implementation of smart contracts (and smart contracts in general) have surfaced [Buterin 2016]. Smart contracts mean that code is law—if a smart contract is implemented incorrectly, or if the conditions for the execution of the contract are imperfectly specified, it will still execute exactly how it was written, not how it was intended. Placing a smart contract on a blockchain makes this problem even worse.

Authors' addresses: Henry Fisher, Computer Science, Grinnell College, 1115 8th Ave, Grinnell, IA, 50112, United States, fisherhe@grinnell.edu; Peter-Michael Osera, Computer Science, Grinnell College, 1115 8th Ave, Grinnell, IA, 50112, United States, fisherhe@grinnell.edu.

XXXX-XXXX/2018/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

You cannot change code at a contract address on the Ethereum blockchain, so any problems are irreversible without careful planning from the outset. Additionally, all smart contracts are usable by anyone at any time. Thus, sufficiently popular smart contracts will essentially be getting fuzz tested all the time. To make matters worse, Ethereum users are highly incentivized to find and exploit contract vulnerabilities. Thus, if they contain vulnerabilities, they will be exploited given enough time.

Given this, anyone building a contract on the Ethereum blockchain that they expect to work must be very confident that they have exactly zero serious bugs, and very few non-serious bugs. Currently, there aren't many tools available to developers looking to gain this level of confidence about their code. Industry standard is to pay thousands of dollars for external auditors to review your code, which is expensive, unreliable, and slow.

The solution for anyone serious about building working smart contracts is to use formal verification techniques to prove properties about their code. There have already been a few projects in this direction [Amani et al. 2018] [Bhargavan et al. 2016] [Luu et al. 2016] [Mueller 2017], but for the most part they just analyze code to catch common, albeit serious bugs. Our project aims to push the field forward on the formal verification front by extending one of these projects which uses the Z3 SMT solver [De Moura and Bjørner 2008] to enable assertion checking in the popular smart contract language Solidity. This allows the tool to catch not just a pre-specified set of bugs, but any problem that can be described by an assertion.

Part 2 of this paper describes how our tool works with an example that will probably be quite common in industry. Part 3 goes into detail on our design. In part 4, we evaluate how useful this tool is and describe next steps for people looking to push further.

2 BACKGROUND AND PRELIMINARY EXAMPLES

The initial example chosen to illustrate how the tool works is a highly simplified Solidity contract called SimpleAssert:

```
contract SimpleAssert {
  uint256 n;

  function SimpleAssert() {
    n = 100;
  }

  function subtract_one() public {
    require(n <= 100);
    n -= 1;
    assert(n < 100);
  }
}
```

SimpleAssert has a single variable, n, which is set to 100 in the constructor (run when the contract is deployed, and never run again). It has a publicly callable function that subtracts one from n.

The expected use case of this tool is to require your preconditions at the beginning of a function you want to verify and assert your postconditions at the end. The analysis tool (which we will describe in more detail later) cannot reason about the contract's history, so it assumes at the beginning of this analysis that n could be anything. In general, you must require your preconditions even if they're technically implied by the other logic in your contract.

As it stands, the assertion will not fail. It would fail if we changed subtract one to look like this:

```
function subtract_one() public {
  require(n <= 100);
  n -= 1;
  assert(n == 100);
}
```

In this case, the tool will output that there's a possible assertion failure, it will list the offending line, and then give the example values it used to make this determination.

Let's use a more realistic example. The motivation behind this tool is that we would like to use it our own project. We are trying to build a stable cryptocurrency (on top of Ethereum) for worldwide use, meaning we need our smart contracts to be robust given at least millions of users and millions of dollars (and thus highly incentivized adversarial actors). We'll focus on perhaps the key contract in the set of smart contracts making up our protocol: the token.

The token contract is useful to analyze for a couple reasons. First, it's quite simple relative to most other contracts. It stores a mapping from addresses to balances, and implements a transfer function (for transferring tokens between accounts) and an approval function (for allowing another user to transfer tokens out of your account). Second, it's an extremely popular contract. Part of the reason Ethereum has grown so big is that it allows people to build currencies on top of their platform, and although smart contracts have many interesting use cases, to date the platform has mostly been used for projects building tokens for their own projects. Third, it's been standardized [Vogelsteller and Buterin 2015], so everyone's tokens will look quite similar, minus a few implementation details.

A simple token might look like this:

```
contract ERC20 {
  mapping (address => mapping (address => uint256)) public allowance;
  mapping (address => uint256) public balanceOf;
  uint256 public totalSupply;

  string public name = "ERC20";
  string public symbol = "ERC";
  uint8 public decimals = 18;

  event Transfer(address indexed _from, address indexed _to, uint256 _value);
  event Approval(address indexed _owner, address indexed _spender, uint256 _value);

  function ERC20() public {
    totalSupply = 0;
  }

  function _transfer(address from, address to, uint256 value) internal {
    balanceOf[from] -= value;
    balanceOf[to] += value;
    Transfer(from, to, value);
  }

  function transfer(address to, uint256 value) public returns (bool) {
    require(balanceOf[msg.sender] > value);
    _transfer(msg.sender, to, value);
    return true;
```

```
  }

  function transferFrom(address from, address to, uint256 value) public returns (bool) {
    require(allowance[from][msg.sender] >= value);
    allowance[from][msg.sender] -= value;
    _transfer(from, to, value);
    return true;
  }

  function approve(address spender, uint256 value) public returns (bool) {
    allowance[msg.sender][spender] = value;
    Approval(msg.sender, spender, value);
    return true;
  }
}
```

This example illustrates how even quite a simple smart contract can have bugs. We'll focus on the _transfer function, as it's the core piece of the contract. The postconditions of this function are that the balance of from decreased by value, and the balance of to increased by value. We can capture this with four asserts: that the balance of from decreased, that the balance of to increased, and that each difference is equal to value. Here's the function with those asserts:

```
function _transfer(address from, address to, uint256 value) internal {
  uint256 prev_from = balanceOf[from];
  uint256 prev_to = balanceOf[to];

  balanceOf[from] -= value;
  balanceOf[to] += value;
  Transfer(from, to, value);

  assert(balanceOf[from] <= prev_from);
  assert(prev_from - balanceOf[from] == value);
  assert(balanceOf[to] >= prev_to);
  assert(balanceOf[to] - prev_to == value);
}
```

Running the tool on this contract shows that the following lines potentially have assert failures:

```
assert(balanceOf[to] >= prev_to);
```

and

```
assert(balanceOf[from] <= prev_from);
```

What this tells us is that it's possible for `_transfer` to decrease the balance of the account it's sending to and increase the balance of the account it's sending from. Both of these bugs result from integer overflow/underflow errors. If `balanceOf[to]` were $2^{256} - 1$ (around 10^{76}), the maximum of a uint256 in Solidity, transferring 10 tokens would yield a balance of 9. Quite the decrease.

Catching these assert fails tells us of new preconditions we did not necessarily know about beforehand: that `balanceOf[to]+value>=balanceOf[to]`, and that `balanceOf[from]-value<=balanceOf[from]`. Codifying these as requires at the beginning of the function yields no more assertion failures.

3 DESIGN AND TECHNICAL DETAILS

This assertion checker is built on top of mythril, a concolic analysis tool for Ethereum smart contracts written in Python. Mythril has a set of modules aimed at discovering particular security flaws in smart contracts (integer overflow, reentrancy bugs, etc), and these modules are built on a symbolic execution engine (also in Python) called laser-ethereum along with a Python implementation of Z3, a constraint solver. The assertion checker is an additional module in mythril.

The symbolic execution engine works as follows: first, it acquires the bytecode for the contract it's analyzing. If it needs to get the contract from an address, it pings the Ethereum network for the bytecode. If it's given a Solidity contract, it compiles that contract. With the bytecode it begins its execution. Since the EVM is a stack based language, the execution is essentially just tracking the program stack and using it to create a control flow graph for the contract. This is a directed graph where each node is a program state (or, more accurately, a set of instructions that the program will unconditionally execute upon arriving in that state), where a node n_0 will have an edge towards another node n_1 if in some possible world n_1 is reached after the steps in n_0 are executed. There are a set of opcodes that prompt a new node to get created. These are generally jumps (which send you to a new index in the bytecode, and can be conditional) and calls (which are, for the most part, calls to other smart contracts). Obviously, this recursive node creation can run into infinite loops. For that reason, the engine only creates nodes to a specified maximum depth, with the default being 12.

Conditionals and conditional jumps are important here. When the symbolic execution engine encounters a conditional, it appends the whole condition (including the expressions it compares) onto the stack. When a conditional jump is found, the engine creates two nodes, one where the condition evaluated to true and one where the condition evaluated to false, and evaluates the rest of the program in each state. Each node in the graph also keeps track of a list of constraints for execution to reach that state. For example, if there was a conditional jump based on an if statement that checked if a variable `n` was equal to 0, every node in the "true" branch will have as a constraint that `n==0`.

The symbolic execution engine returns this graph to the main program, and the modules use it to run their analyses. Most modules simply iterate through nodes checking for a particular opcode, possibly imposing an additional constraint on the list of constraints in a node, and then running the constraints through Z3 to find satisfying assignments. This is largely how the assertion checker works as well. When an assertion fails, an INVALID opcode is hit. Thus, the module runs through all nodes checking for INVALID, and if such a node is found (there should be if there was an assert in the original program), the module checks for variable assignments that satisfy the constraints in that node. The relevant piece of code for the module is this:

```
for instruction in node.instruction_list:
  if (instruction['opcode'] == "INVALID"):
  try:
    model = solver.get_model(node.constraints)
    issue = Issue(node.module_name, node.function_name, instruction['address'], \\
      "Assert Fail", "Warning")
    issue.description = "A possible assert failure exists in the function " + \\
      node.function_name
    issues.append(issue)

    for d in model.decls():
      print("[INTEGER_UNDERFLOW] model: %s = 0x%x" % (d.name(), model[d].as_long()))
```

```
  except UnsatError:
    logging.debug("Couldn't find constraints to reach this invalid")
```

For any node, we iterate through its instruction, and check to see if that node's opcode is INVALID. If it is, then we try to find a satisfying model for the constraints in the node. If we can find a satisfying model for the constraint, it means there's a set of inputs such that we could actually have the assertion fail. If the solver fails to get a model, then this assertion cannot fail.

Let's return to the SimpleAssert contract from earlier and analyze what would happen when the symbolic execution engine reaches the function subtract_one. The assembly for that section is:

```
/* "SimpleAssert.sol":177:178  1 */
0x1
/* "SimpleAssert.sol":172:173  n */
0x0
dup1
/* "SimpleAssert.sol":172:178  n -= 1 */
dup3
dup3
sload
sub
swap3
pop
pop
dup2
swap1
sstore
pop
/* "SimpleAssert.sol":199:202  100 */
0x64
/* "SimpleAssert.sol":195:196  n */
sload(0x0)
/* "SimpleAssert.sol":195:202  n < 100 */
lt
/* "SimpleAssert.sol":188:203  assert(n < 100) */
iszero
iszero
tag_7
jumpi
invalid
tag_7:
/* "SimpleAssert.sol":131:210  function subtract_one() public {... */
jump  // out
```

The first few instructions are simply for subtracting 1 from n (placing n and 1 on the stack, subtracting them, and then storing that value in n). The assertion comes when 100 and n are loaded on the stack, and then compared with an lt call. The result of the lt call is negated twice with iszero, then tag_7 (the end of the function) is placed on the stack. If lt returned a 1, jumpi will go to tag_7. If it didn't (meaning the assertion failed), then an invalid opcode is reached.

When the symbolic execution engine reaches this jumpi, it will get the condition off the stack and create a node with that constraint, and a node with its negation. In this case, the condition is

the lt. Thus, the new nodes will have either $n - 1 < 100$ or $n - 1 >= 100$. Since this jumpi represents the result of an assert, the negation of the condition leads to a node with only an invalid opcode (the program halts upon reaching invalid). So when the assertion checker module runs through these nodes, it will find an invalid opcode with $n - 1 >= 100$ in its list of constraints (in this case, that is the only constraint). This constraint solved for by Z3, which finds that n could have been 101 at the beginning of the function and this assert would fail.

Compare this to what happens when the precondition $n \leq 100$ is required. This results in a very similar opcode, with the addition of this snippet at the beginning of the function:

```
/* "SimpleAssert.sol":185:188  100 */
0x64
/* "SimpleAssert.sol":180:181  n */
sload(0x0)
/* "SimpleAssert.sol":180:188  n <= 100 */
gt
iszero
/* "SimpleAssert.sol":172:189  require(n <= 100) */
iszero
iszero
tag_7
jumpi
0x0
dup1
revert
```

(Note that the tag numbers have changed between these two compilations, so the two tag_7s are not the same).

Here, upon reaching jumpi, the engine would create nodes and append the constraint $n > 100$ to the node with the revert, and the constraint $n \leq 100$ to the node that continues the execution. When the engine reaches the jumpi for the assert, it will create a node for the invalid opcode and append the constraint $n - 1 \geq 100$. Z3 then solves for n where $n - 1 \geq 100 \wedge n \leq 100$. At first glance this is logically impossible, but as the assertion checker will tell us, if $n = 0$ this constraint is satisfied, since when you subtract 1 from a uint256 with value 0, it becomes $2^{256} - 1$. If we require that $n > 0$ at the beginning of this function, this assertion cannot fail.

4 EVALUATION

This module was tested using a combination of tests around known Ethereum bugs, tests around suspected edge cases, tests around known edge cases, and general functionality tests to make sure it works on easy inputs. In the following table we document tests we ran, whether or not they passed (i.e., did they report every error that exists and no more?), and if they failed in this regard what type of error that constitutes.

Description	Result	Error Type
SimpleAssert without requires	Failed	False Positive
SimpleAssert with requires	Passed	
ERC-20 Token	Passed	
Loop heavy contract	Failed	False Negative
Re-entrancy vulnerable contract	Failed	False Negative
Integer under/overflow	Passed	
Adversarial miner	Failed	False Negative
Adversarial miner with preconditions	Passed	

This demonstrates that the tool largely has the ability to find assertion failures, but will be incorrect in a couple known cases.

False positives aren't a big problem as they aren't common and will usually be discovered when they occur. The only case where we run into false positives is where the assertion checker has no knowledge of the state of a contract before the transaction it's analyzing. This is the simple assert contract without requires. In this contract, `n` is initialized to 100 and can't possibly increase since the only publicly callable function decreases it by 1. However, the symbolic execution engine doesn't know what `n` was initialized to, and doesn't know that there's no other method that increases `n`. Thus, it won't impose a constraint that `n<=100`, and it will find that the assertion that `n<100` could fail. When asserts are added, the assertion checker handles SimpleAssert correctly.

False negatives are more sinister. From our testing, there are two types of false negatives. The first comes from problems due to not discovering a node past the max_depth of the engine. As mentioned before, the node depth increases when the execution engine encounters a jump or a call. Jumps are likely to be bigger problems, as they happen for coding patterns used regularly. If/else statements, for instance, jump once. Loops are especially jump heavy. If a for loop runs m times, the program will jump m times. If there is a for loop in a function running, a far higher than normal max_depth must be supplied or the execution engine will not go deep enough to encounter assertion failures. This behavior around loops is certainly a problem, though its severity is mitigated by the strict bound on computation time imposed by Ethereum gas fees, which cause most contracts to have either no loops or very short ones.

The second type of false negatives come from problems with the particularly weird execution environment of a blockchain. As it stands, there are two common bugs of this type that the assertion checker does not catch. The first is a problem best classified as "miners are not your friends" problems. An example of this occurs within some token contracts: if I've allowed a miner to transfer 10 tokens out of my account, and I change that allowance to 9, they can place a transaction in a block that withdraws 10 tokens from my account before the allowance is set to 9, then withdraw again after the allowance is set to 9, taking 19 total when I wanted them to only take 9. The assertion checker can't catch bugs like this because its symbolic execution engine only executes over a single transaction, and this is a problem stemming from the ordering of multiple transactions.

The second problem is known as a re-entrancy bug. This occurs when a contract sends funds to an outside account. That outside account can call the same method that called it, potentially draining more funds than intended. The symbolic execution engine is not currently set up to handle calls like this. It's possible it could be made to respond correctly, though it's a difficult problem. External calls open up the potential for all public methods in a contract to be called. Thus the symbolic execution engine could be changed so that any external call creates a node going to the beginning of every publicly callable function in the contract. While this would work, it may run into issues with exceeding the maximum depth of the execution engine. In all likelihood, a better solution from a program verification perspective is to prove by hand that functions with external

calls work. Most contracts won't have any methods with external calls, and the ones that do will likely have very few anyway.

There are also limitations worth mentioning beyond the correctness of the protocol itself. As we have noted, the expected use of this tool is to require your preconditions, and assert your postconditions. If your preconditions and your code do not imply your postconditions, the tool will tell you.

However, this still leaves quite a bit up to the programmer. Take the ERC-20 token as an example. The postconditions we stated were that the balance of the sender decreased by value and the balance of the receiver increased by value. It's easy to miss that the propositions `prevSenderBal-curSenderBal==value` and `curReceiverBal-prevReceiverBal==value` do not fully specify the postconditions, as they don't account for integer overflow. You need to actually be aware of the possibility for integer overflow in order to realize that there's a problem with it here.

The key point here is that this tool will not tell you whether your preconditions or postconditions are correctly specified for what you want a procedure to do, only whether or not a particular function actually results in the postconditions given the preconditions.

Although it does not technically run a formal verification process, this assertion checker lays the groundwork for a formal verification process for smart contracts. This process, roughly, involves proving by hand first that if your contract's methods work correctly, your contract will do what you intend it to do (e.g. that a token with a correct transfer and approval method is actually the intended token), and second proving that each of your methods works correctly. This tool handles the latter responsibility, and leaves the former up to the programmer.

Obviously, this process is error prone. However, constraints in the computation environments on blockchains mean smart contracts are likely to be highly simple, and thus the burden of the work done by hand is lower than what one might expect. Additionally, while many programs seem to work quite trivially, the blockchain is such an adversarial environment that it's still worth it to go from high confidence in program correctness to extreme confidence. Last, if it's done correctly, this process provides highly legible proof that a contract works correctly. Given the smart contract community has no easy to use tool that accomplishes this, this process and tool is a moderate upgrade over current verification processes for any project that has the time and leeway to do it correctly.

5 CONCLUSION

This tool represents a solid step towards legible program verification in a manner actually usable by projects writing smart contracts on the Ethereum blockchain. It is by no means perfect. You must use it in conjunction with proofs by hand, it may not catch errors in code with many loops and it doesn't handle "blockchain" bugs (re-entrancy bugs and problems with adversarial miners), we believe it can serve a useful role within a project trying to develop robust code on the Ethereum blockchain if one plans around its limitations.

Thanks to Bernhard Mueller, ConsenSys, and the Reserve team for their contributions to this project.

REFERENCES

Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/3167084

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

Vitalik Buterin. 2013. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. (2013). https://github.com/ethereum/wiki/wiki/White-Paper

Vitalik Buterin. 2016. Blog. (2016). https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. Cryptology ePrint Archive, Report 2016/633. (2016). https://eprint.iacr.org/2016/633.

Bernhard Mueller. 2017. Security analysis tool for Ethereum smart contracts. (2017). https://github.com/ConsenSys/mythril

Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997). http://firstmonday.org/ojs/index.php/fm/article/view/548

Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20 Token Standard. (2015).

Gavin Wood. 2018. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. (2018). https://ethereum.github.io/yellowpaper/paper.pdf