Internals of Solidity's SMTChecker

By Atharv Singh Patlan
BT-CSE, 190200
Supervisor: Prof. Sandeep Shukla

1. Abstract

Solidity smart contracts are the most prominent method of writing smart contracts used to interact with the global computer, which is the Ethereum blockchain. These contracts are mostly immutable and open-source and carry high financial incentives, which make them really prone to attacks. The importance of assuring their security is thus of utmost importance. In our work, we analyze SMTChecker, the formal verification engine shipped with the Solidity compiler, looking at its working, along with comparing it with popular 3rd party static and dynamic analysis tools for Ethereum smart contracts. We try to find similarities and differences between them and find advantages of using the SMTChecker over these, looking at the SWC registry for reference. A benchmark suite with vulnerable smart contracts was also developed for evaluation and comparison with other verification tools.

2. Introduction

In spite of the plethora of smart contract verification tools available, the rapid development of Solidity as a language and its applications makes it difficult to keep the tools in sync with the language.

Using flexible verifiers based on model checking and satisfiability modulo theories (SMT) for checking smart contracts, integrated directly into the language compiler, allows the precise modeling of features specific to Solidity as and when the new changes are launched. Due to it being based on formally verifying contract properties, it can automatically generate ways to exploit vulnerabilities, something which will have to be added manually to other types of tools.

3. SMTChecker in the Solidity Compiler

An overview of the compilation process is depicted in Fig. 1. Essentially, using SMTChecker becomes another pass over the source code in the normal compilation process that starts after parsing and Abstract Syntax Tree (AST) generation.

3.1 SMT Encodings

Figure 2 shows a Solidity sample showing the SMT encodings. As illustrate by [Isola] 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.

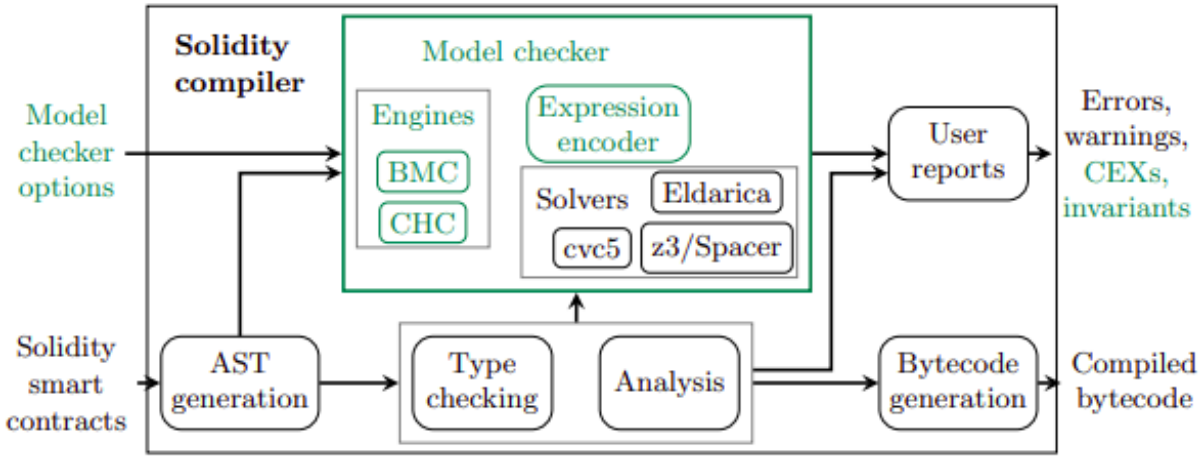

Fig 1: The Solidity compiler stack with SMTChecker (in green). Illustrated by [Alt]

```
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: Sample SMT encoding of a contract C

3.2 Verification Targets

The SMTChecker has two kinds of automatic verification targets:

1. **Arithmetic:** Overflow/underflow, zero division, insufficient transfer balance
2. **Structural:** Assertions, popping empty array, out-of-bounds access

SMTChecker makes use of the assert and require statements in solidity to capture the program logic: require

statements are considered as assumptions and assert statements are used as verification targets, and the model attempts to prove them or provide a counterexample to them.

Using these statements, it can also model reentrant calls and reachability of self-destruct statements

4. Related Smart Contract Analysis Tools

Along with Formal Verification based tools like SMTChecker, we majorly have two other popular types of tools, based on Static and Dynamic Analysis respectively.

Static Analysis – Slither : A powerful static analyzer, Slither does not provide formal guarantees but can detect many vulnerabilities, whose detection methods have been built into the system. Thus, it can't detect new vulnerabilities.

Dynamic Analysis – Mythril : This is also an SMT-based symbolic execution tool, however it works on the EVM bytecode and checks only for specific, known vulnerabilities.

5. Experimental Setup

We had two main goals in our work:

1. Build a pipeline to perform automatic smart contract analysis of both on-system and on main-chain contracts using different SOTA smart contract analysis tools
2. Compare the performance of Formal verification based tool : SMTChecker with two SOTA tools : Slither and Mythril

5.1 Automated Evaluation Pipeline

We built a pipeline, the first goal of our work, to allow a smooth analysis of any smart contract. These were the features of our pipeline

1. Automatic compiler selection: Many Solidity contracts allow for the use of different compiler versions, however, contract analysis tools require a specific contract version. Thus, our pipeline enabled for the automatic detection of the best version of the compiler to be used from the code, even if there are multiple require statements in the contract.
2. Support for multi-file contracts with automatic contract flattening : A large number of contracts are present as multi-file contracts which have been uploaded onto the Blockchain. Certain tools require all the contracts to be present in a particular file, with a common compiler version. Our pipeline automatically performs this for a smoother contract security analysis.
3. Support for analyzing contracts from the Ethereum main-chain : Our pipeline can directly download and analyze contracts deployed on the main Ethereum chain, and report vulnerabilities.

5.2 Benchmarking of tools

A benchmark suite was developed to evaluate the performance of SMTChecker and compare it with the

performance of Slither and Mythril.

We make use of known vulnerabilities from the SWC registry, along with some more complex examples for cases which are not in the automatic verification targets of SMTChecker. We sample our test cases from **here.**

6. Results

6.1 Automatic Verification Targets

6.1.1 Insufficient Transfer Balance

Consider the following contract

```
contract C {
      function f(address payable a) public {
            a.transfer(200);
      }
}
```

SMTChecker detects insufficient balance by modeling the contract to initially have zero balance, which makes this function fail.

Slither also detects this by mentioning that amount is being transferred to an arbitrary address which might lead to the balance being wiped out.

Mythril does not detect this

6.1.2 Out of bounds access

Consider the below code, accessing an array out of bounds

```
contract MyContract {
      uint8 i;
      function dyn_array_oob_loop(uint8 n) public {
            uint8[] memory a = new uint8[](n);
            for (i = 0; i < n; ++i){
                  a[i] = 100;
            }
            assert(a[0] == 100);
      }
}
```

As can be seen from the code, it is possible that `a[0]` might not exist, as it won't be allocated if the value of `n` is passed as `0`.

SMTChecker is able to detect this possibility

```
Warning: CHC: Out of bounds access happens here.
Counterexample:
i = 0
n = 0
a = []

Transaction trace:
MyContract.constructor()
State: i = 0
MyContract.dyn_array_oob_loop(0)
  --> test_suite/dynamic_array_oob_loop.sol:11:12:
   |
11 |     assert(a[0] == 100);
   |            ^^^^
```

Slither and Mythril do not detect out of bounds access.

It is important to detect the above two kinds of vulnerabilities to prevent the contract from crashing and to avoid unexpected situations to occur once the contract has been deployed.

6.1.3 Results on Automatic Verification Targets

If we run all our tools on the various Automatic Verification Targets presented by SMTChecker, we observe that most of these vulnerabilities can not be detected by Slither and Mythril. Along with that, SMTChecker is able to provide a counter example to these problems and help in better error reduction.
In the table, CE stands for whether the tool was able to generate Counter Examples

Target	**SMTChecker**		**Slither**		**Mythril**	
	Detection	**CE**	**Detection**	**CE**	**Detection**	**CE**
Overflow/underflow	Y	Y	N	-	Y	N
Zero Division	Y	Y	N	-	N	-
Insufficient transfer balance	Y	Y	Y	N	N	-
Assertions	Y	Y	N	-	N	-
Popping empty array	Y	Y	N	-	N	-
Out of bounds access	Y	Y	N	-	N	-

Table 1: Results of the tools on Automatic Verification Targets

6.2 Complex Verification Targets

We study the issue of Reentracy, which is one of the complex verification targets handled by SMTChecker. Reentrancy is one of the most studied smart contract vulnerabilities. A reentrancy attack occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.or make changes to the state variables.

6.2.1 External Calls for Reentrancy

The below contract ExtCall is prone to reentrancy. As can be seen, the `u.callme()` function can actually call `ExtCall.setX()` and thus can change the value of `x` to make the assertion fail.

```
interface Unknown {
      function callme() external;
}

contract ExtCall {
      uint x;

      function setX(uint y) mutex public {
            x = y;
      }

      function xMut(Unknown u) public {
            uint x_prev = x;
            u.callme();
            assert(x_prev == x);
      }
}
```

SMTChecker clearly detects ths vulnerability and can exactly model the reentrant calls.

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1
u = 0
x_prev = 0

Transaction trace:
ExtCall.constructor()
State: x = 0
ExtCall.xMut(0)
    u.callMe() -- untrusted external call, synthesized as:
        ExtCall.setX(1) -- reentrant call
  --> test_suite/non_mut_reentrancy.sol:18:3:
   |
18 |       assert(x_prev == x);
   |       ^^^^^^^^^^^^^^^^^^^
```

Mythril also shows the possibility of a re-entrant call due to arbitrary external call to a user address. However, slither does not detect any vulnerabilities as it only detects reentrant calls involving transfer of funds.

6.2.2 Blocking reentrancy with mutex

However, if we block these reentrant calls using mutex, like in the following code,

```
interface Unknown {
      function callme() external;
}

contract ExtCall {
      uint x;

      bool lock;
      modifier mutex {
            require(!lock);
            lock = true;
            _;
            lock = false;
      }

      function setX(uint y) mutex public {
            x = y;
      }

      function xMut(Unknown u) mutex public {
            uint x_prev = x;
            u.callme();
            assert(x_prev == x);
      }
}
```

SMTChecker is able to prove that the invariant in this case holds as the mutex blocks reentrancy, and displays that no error has been detected.

Mythril, however, again shows the possibility of a re-entrant call due to arbitrary external call to a user address, not taking into account the mutex. It hence is a false-positive in this case. Slither, again, does not detect any vulnerabilities

6.3 Uninterpreted Verification Targets

There are a large number of functions in Solidity, that SMTChecker does not have an explicit set of SMT rules. For example, using delegatecall and timelock are known vulnerabilities for which SMTChecker does not have explicit rules yet.

However, it is still able to detect the vulnerabilities due to these as these are treated as function calls to unknown code, and SMTChecker models them as uninterpreted functions (UFs) over the arguments, and tries proving the invariant using these UFs.

6.3.1 Delegatecall

The below code uses the Attack contract to change the owner of Hackme

```
contract Lib {
    address public owner;

    function pwn() public {
        owner = msg.sender;
    }
}

contract HackMe {
    address public owner;
    Lib public lib;

    constructor(Lib _lib) {
        owner = msg.sender;
        lib = Lib(_lib);
    }

    fallback() external payable {

        address o_owner = owner;
        address(lib).delegatecall(msg.data);
         assert(o_owner == owner);
    }
}

contract Attack {
    address public hackMe;

    constructor(address _hackMe) {
        hackMe = _hackMe;
    }

    function attack() public {
        hackMe.call(abi.encodeWithSignature("pwn()"));
    }
}
```

Mythril and Slither detect the presence of a delegatecall using a user supplied address.

SMTChecker does not currently provide SMT rules for delegatecall, however it is still able to detect a case where the owner can be changed.

```
Warning: Assertion checker does not yet implement this type of function
call.
  --> test_suite/delegatecall.sol:48:9:
   |
48 |         address(lib).delegatecall(msg.data);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning: CHC: Assertion violation happens here.
Counterexample:
owner = 0x01, lib = 0
o_owner = 0x0

Transaction trace:
HackMe.constructor(0){ msg.sender: 0x0 }
State: owner = 0x0, lib = 0
HackMe.fallback(){ msg.data: [0x1c, 0x1c, 0x1c], msg.value: 2437 }
  --> test_suite/delegatecall.sol:49:2:
   |
49 | assert(o_owner == owner);
   | ^^^^^^^^^^^^^^^^^^^^^^^^
```

It can be seen that SMTChecker can model the delegatecall and can disprove the assertion.

6.3.2 Timelock

Similar to delegatecall, if a contract uses the timestamp to make decisions, Mythril and Slither detect the presence of this precisely. Again, SMTChecker does not implement SMT rules for timestamp, but is still able to prove the contract unsafe by finding a series of parameter values and call which result in errors for the lock time.

7. Comparison of tools in the SWC-registry

Here is a comparison of the various vulnerabilities in the SWC-registry, whether they have an SMT implementation in the SMTChecker, and finally, how the three tools detect them. D stands for whether the vulnerability was detected, CE stands for whether a counter-example was shown by the tools, and Y* means that the bug was detected, although with false positives.

SWC-ID	Title	Verification Target Type for SMTChecker	SMTChecker		Slither		Mythril	
			D	CE	D	CE	D	CE
SWC-100	Integer Overflow / Underflow	Automatic	Y	Y	N	-	Y	N
SWC-105	Unprotected Ether Withdrawal	Uninterpreted	Y	Y	Y	N	Y	Y
SWC-106	Unprotected SELFDESTRUCT	Uninterpreted	Y	N	Y	N	Y	N
SWC-107	Reentrancy	Complex	Y	Y	Y	N	Y*	N
SWC-110	Assert Violation	Automatic	Y	Y	N	-	N	-
SWC-112	Delegatecall to untrusted callee	Uninterpreted	Y	Y	Y	N	Y	N
SWC-115	Authorization through tx.origin	Uninterpreted	N	N	Y	N	Y	N
SWC-116	Block values as a proxy of time	Uninterpreted	Y	Y	Y	N	Y	N
SWC-132	Unexpected Ether balance	Automatic	Y	Y	Y	N	Y	N

Table 2: Comparison of the tools on the SWC-registry

This shows the power of SMTChecker as it detects a wide variety of SWC bugs, without having any explicit implementations of a lot of these issues.

8. Running SMTChecker on a live contract

The SMTChecker majorly operates on assert statements, and not many contracts are deployed with assert statements.

A popular contract, the Eth2.0 Deposit contract, is deployed with an assert(false) statement.

We run our SMTChecker on this by downloading the contract at address `0x00000000219ab540356cbb839cbe05303d7705fa`. As the current version of SMTChecker works only on solc version >= 0.8.14, we change the version of the contract forcefully

SMTChecker times out on this contract; thus, most likely, it is safe.

9. Conclusion and Future Work

Clearly, the SMTChecker is a powerful tool.

It provides developers with a quick access verification tool, which is very powerful and can even model complex function call orders.

It does not require a lot of new modifications when new Solidity features are launched, as is shown by its ability to detect and provide examples for Uninterpreted verification targets (for which rules haven't been

built into SMTChecker).

The downside of the SMTChecker is that it depends on assertions, and thus, every programmer needs to take care of the conditions they want to be satisfied after a function performs its tasks. Relying on tools such as Slither and Mythril allows the developers to transfer the responsibility of detecting possible faulty locations to the developers of the tools. Thus, there is an ease of use in them.

As some future goals, we would want to compare the functioning and performance of different formal verification engines supported by SMTChecker, and also compare Bytecode-based SMT tools (such as Mythril) with SMTChecker from a theoretical perspective and detect advantages and disadvantages of either method.

10. Codebase

Codebase is available here.

11. References

https://arxiv.org/pdf/2111.13117.pdf

http://verify.inf.usi.ch/sites/default/files/solcmc.pdf

Beacon Deposit Contract | Address 0x00000000219ab540356cbb839cbe05303d7705fa | Etherscan

https://github.com/chriseth/solidity_isola/blob/master/main.pdf

Solidity by Example