Keywords: Smart Contract, Bounded Model Checker, Condition Coverage, Solidity Compiler.

Abstract: Advances in blockchain technologies enable society toward trust-based applications. Smart contracts are the scripts holding the properties to perform the activities in Blockchain. Smart contracts are prepared between the parties to hold their requirements and promises. If the deal held by a smart contract is huge and expensive, then there is a high chance of attracting issues and loss of assets. This necessitates the verification and testing of a smart contract. In this paper, we demonstrate an approach for generating test cases to satisfy the condition coverage of smart contracts using a solidity-bounded model checker. We show the annotation of the original smart contract as per the condition coverage specification and drive the bounded model checker to prove the feasibility of the asserted properties. Finally, we collect all feasible targets and show the condition coverage score. Also, the proposed approach generates test input values for each feasible atomic condition. The approach presented has been tested with 70 smart contracts, resulting in 57.14% of contracts with good condition coverage scores. Our work can be utilized to certify any smart contract to check whether the Optimal or Maximal condition coverage is achieved or not.

1 INTRODUCTION

Blockchains are distributed data structures to store the agreed sequence of transactions in a user network. They can be employed in many applications (e.g., such as banking, insurance, health applications, vehicle networks, shipping, logistics, and cyber-security) that need data exchange between different users. The actions are stored in the form of a block and the data is distributed over individual nodes after acceptance by the respective user. One of the major advantages of blockchain technologies are to avoid tempering data (that is, immutability) by anonymous users, which in turn increases transparency and security. In addition, blockchain has several unique and desirable properties including decentralization, auditability, anonymity, and autonomously enforcing logic via a smart contract.

Blockchains (such as Hyperledger (Buterin et al., 2014) and Ethereum (Dannen, 2017; etherscan, 2021)) enforce consensus, if any, by the users involved, as defined in the smart contract. A smart contract is a computer program, comprising executable codes, residing on the blockchain and executed once specific (pre-defined) conditions are met. The transactions are programmable by smart contracts. A smart contract pays attention to transactions sent to it, executes application logic upon receipt of a transaction, and depending on the need can generate other transactions that can be received by participating users. Thus, a smart contract includes code and data on which the smart contract operates. Also, a smart contract can control other smart contracts.

Smart contract-enabled blockchains guarantee that conditions in a smart contract are not modified once they have been written and published. The code for smart contracts is typically written in a high-level programming language such as Go\(^1\) for Hyperledger\(^2\) and Solidity\(^3\) for Ethereum. Similar to traditional software applications, there may be deviations, errors, and vulnerabilities in the smart contract logic code. Thus, coding smart contract logic as per the applica-

\(^1\)“The Go programming language,” https://golang.org/.
\(^2\)“Hyperledger project,” https://www.hyperledger.org/.
\(^3\)“Solidity smart-contract language,” https://solitude.readthedocs.io/.
tion requirements and ensuring the correctness of that code is very important and challenging. To the best of our knowledge, there is very limited work in the literature on the verification and testing of the smart contract logic.

Our approach enables the annotation of a smart contract with goal constraints or targets in the form of "asserts" that ensures the specification of condition coverage. The solidity compiler with a bounded model checking feature detects the targets in annotated smart contracts. Later, we extract useful information from the execution report and show the coverage information. The main contribution of this work is to provide a system for the contributors of smart contracts to test their scripts and certify the smart contracts with three classes viz. Optimal, Maximal, and Incomplete.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. Section 3 shows basic concepts. Section 4 presents the proposed Condition Coverage analysis for Smart Contracts using the Solidity Bounded Model Checker (CC-SolBMC). Section 5 describes the experimental results. Finally, We conclude the paper with future insights in Section 6.

2 RELATED WORK

The first blockchain platform that supports smart contracts is Ethereum (Dannen, 2017; Wood et al., 2014), in which Solidity scripting language is used for developing smart contracts. There are some works (e.g. (Sánchez-Gómez et al., 2019; Liu et al., 2020)) in the domain of smart contract testing.

Wen and Miller (Wen and Miller, 2016) described an automated bug identification in smart contracts, however, our work proposed to measure the condition coverage. Praithheeshan et al (Praitheeshan et al., 2019) presented a survey on vulnerability analysis and formal verification methods in Ethereum smart contracts. They surveyed 16 kinds of vulnerabilities in smart contracts and presented the existing security issues, analysis tools, and detection methods along with their limitations. They have categorized the analysis method into three categories viz. formal verification, static, and dynamic. These categories are compared based on accuracy, vulnerability detection, and performance.

Benitez et al. (Benitez et al., 2020) considered the program for smart contracts as a chain of typed programs (typechain) and provided verified properties to ensure the smart contract correctness.

Andesta et al. (Andesta et al., 2020) have proposed 10 classes of mutation operators and presented a method of testing solidity smart contracts using mutation testing. Their proposed method is capable of regenerating the real bug in ten contracts out of fifteen contracts and they have also provided their mutation operators with a universal mutator tool.

Manticore (Mossberg et al., 2019) is a Dynamic Symbolic Executor, used for analyzing smart contracts. It uses z3 a constraint solver. However, it is unable to analyze smart contracts if there are multiple contracts in a single file. Due to its dynamic nature, it takes more execution time compared to others.

Solanalyser (Akca et al., 2019) uses both static and dynamic analyses of vulnerabilities in smart contracts. It can also inject faults into the existing Smart contract for testing purposes.

sFuzz(Nguyen et al., 2020) is a combination of AFL fuzzer and a multi-objective adaptive strategy that is useful for the verification of smart contracts. This tool is prone to false positives and excludes the view function.

ContractFuzzer (Jiang et al., 2018) uses Contract Application Binary Interface (ABI) specifications for generating test cases by fuzzing. However, it does not check for Integer overflow/underflow, and in many cases, it shows false negatives.

3 BASIC CONCEPTS

In this section we discuss some important terminologies used in this paper.

As per IBM4 “Smart contracts are scripts stored on a blockchain that run when all the previously essential requirements are full-filled. Smart contracts are used to automate the process of an agreement so that all contributors can be immediately certain of the outcome, without any intermediary’s involvement or time loss. These can be automated in a workflow, triggering the next action when requirements are met.”

In Bounded Model Checking (BMC)(Clarke et al., 2003; Clarke et al., 2004), a Boolean formula is constructed which is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reach certain states of interest.

Condition Coverage is the percentage of conditions within decision expressions in a script that has been evaluated to be both true and false by NIST5

---

4https://www.ibm.com/topics/smart-contracts
5https://csrc.nist.gov/glossary/term/Condition_coverage
CC-SolBMC: Condition Coverage Analysis for Smart Contracts Using Solidity Bounded Model Checker

4 PROPOSED APPROACH

In this section, we discuss our proposed framework and provide a detailed explanation with a working example.

4.1 Framework of CC-SolBMC

Fig. 1 shows a schematic representation of our proposed approach CC-SolBMC. This framework mainly contains three components namely 1. Source Contract Annotator, 2. SolBMC, and 3. Extractor. The flow starts with supplying the Original Smart Contract into Source Contract Annotator to produce Annotated Smart Contract. This modified version of the smart contract has markings for true and false branch edges for each atomic condition. These markings are in the form of assertions.

Next, the Annotated Smart Contract is supplied into a smart verifier SolBMC. Since SolBMC follows the Satisfiability Modulo Theories (SMT) technique using Z3 constraint solvers, the reachability and feasibility of each marked assertion can be done. The SolBMC generates a detailed execution report. This execution report contains the log of each assertion violation with the counter-example (test inputs). The Extractor component analyses the Execution Report and identifies the total number of assertion violations. Each assertion violation represents a branch edge. Also, the Extractor analyses the Original Smart Contract to identify the total number of atomic conditions present in if-else statements and in require statements and counts the total number of calling modifiers. Finally, Condition Coverage Report is generated from Extractor, which contains the condition coverage score.

4.2 Algorithmic Description

In this section, we explain our proposed approach with algorithmic descriptions.

Algorithm 1 shows the implementation of the Source_Contract_Annotator. Here we supply Smart Contract SC as an input and get Annotated Smart Contract ASC as an output. Line 1 of Algorithm 1 shows an iteration for all the conditions (AllConditions) identified from SC. It is to be noted that, for any conditional statement there are two branches viz. True and False. The loop iterates for each condition identified at a specific line. For a TRUE branch of Condition, create “assert(!(condition));”. Similarly, for a FALSE branch of Condition, create “assert(!(!(condition)))).” Line 8 of Algorithm 1 injects the created assertions in Lines 3 and 6 of Algorithm 1 just above the location at line number where the condition was identified. Finally, after iterating the loop for all the conditions present in SC, Algorithm 1 returns ASC.

Algorithm 1: Source_Contract_Annotator.

Input: SC
Output: ASC

\[1\text{ while Condition } \in \text{ AllConditions do} \]
\[2\quad \text{if TRUE Branch of Condition then} \]
\[3\quad \quad \text{Create “assert(!(condition));.”} \]
\[4\quad \text{end} \]
\[5\quad \text{if FALSE Branch of Condition then} \]
\[6\quad \quad \text{Create “assert(!(!(condition))));.”} \]
\[7\quad \text{end} \]
\[8\quad \text{Inject “assert(!(condition));” and “assert(!( !(condition)))) above the line} \]
\[9\quad \text{where the Condition was identified;} \]
\[10\quad \text{return ASC;}
\]

Algorithm 2 shows the logic of CC-SolBMC. We provide a Smart Contract (SC) as an input to CC-SolBMC and produce Condition Coverage\(^6\) Score (CC\%) as an output. Line 1 of Algorithm 2 invokes Algorithm 1 i.e. Source_Contract_Annotator by supplying SC and produces Annotated Smart Contract (ASC). At Line 2, SolBMC is called with argument ASC and generates Execution_Report. This report contains all the useful information for detecting the targets with counterexamples. Finally, the Extractor is called by supplying SC and Execution_Report to produce Condition_Coverage_Report. Using the Condition_Coverage_Report with the values of Detected unique assertions and Total injected assertions, we compute CC\% Line 4.

\[^6\text{It is to be noted that we consider atomic condition from a decision with or without any logical operator(s).}\]
Algorithm 2: CC-SolBMC.
Input: SC
Output: CC%
1 ASC ← Source_Contract_Annotator(SC);
2 Execution_Report ← SolBMC(ASC);
3 Condition_Coverage_Report ← Extractor(SC, Execution_Report);
4 CC% = (Detected unique assertions / Total injected assertions)X100

4.3 Working Example

In this section, we consider an example smart contract namely Token.sol from the set of 70 smart contracts. The original smart contract for Token.sol is shown in Listing 1. The characteristics for Token.sol wrt. size is 37 LOCs, 4 functions, and 3 atomic conditions. In a smart contract the predicate and conditions can be written in if-else, if-else-if, for-loop, while-loop and require statements. It means, for the statements with boolean logical operators and relational operators, we consider those statements to be analysed. So, in this contract, we have three require conditions as shown below:

\[
\text{balanceOf}(msg\_sender) \geq value
\]
\[
\text{balanceOf}(from) \geq value
\]
\[
\text{allowance}[from][msg\_sender] \geq value
\]

If the first two require conditions are true then the “balance too low” message will be printed. Similarly, for the third require condition “allowance too low” message will be printed. So, it is very important to check the other Branch of the conditions.

Next, using Source Contract Annotator component of our proposed approach, we generate the Annotated Smart Contract version i.e Token_mod.sol as shown in Listing 2. In this version, we inject the targets using the “assert” syntax just above the predicates or conditions identified in the contract. We prepare the targets as shown below:

For I\textsuperscript{st} Condition:
\[
\text{balanceOf}(msg\_sender) \geq value
\]

TRUE branch of I\textsuperscript{st} Condition:
assert(!(balanceOf(msg\_sender) >= value));

FALSE branch of I\textsuperscript{st} Condition:
assert(!(balanceOf(msg\_sender) >= value));

For II\textsuperscript{nd} Condition:
\[
\text{balanceOf}(from) \geq value
\]

TRUE branch of II\textsuperscript{nd} Condition:
assert(!(balanceOf(from) >= value));

FALSE branch of II\textsuperscript{nd} Condition:
assert(!(balanceOf(from) >= value));

For III\textsuperscript{rd} Condition:
\[
\text{allowance}[from][msg\_sender] \geq value
\]

TRUE branch of III\textsuperscript{rd} Condition:
assert(!(allowance[from][msg\_sender] >= value));

FALSE branch of III\textsuperscript{rd} Condition:
assert(!(allowance[from][msg\_sender] >= value));

Note that there is a total number of 6 targets which represent the six branch edges for three atomic conditions.
Listing 1: Original Smart Contract Token.sol

```solidity
pragma solidity ^0.8.2;
contract Token {
    mapping(address => uint) public balances;
    mapping(address => mapping(address => uint)) public allowance;
    uint public totalSupply = 100000000000 * 10 ** 18;

    string public name = "ELONBUCCS";
    string public symbol = "BUC";
    uint public decimals = 18;

    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);

    constructor() {
        balances[msg.sender] = totalSupply;
    }

    function balanceOf(address owner) public view returns (uint) {
        return balances[owner];
    }

    function transfer(address to, uint value) public returns (bool) {
        assert((balanceOf(msg.sender) >= value));
        assert(((balanceOf(msg.sender) >= value);
        require((balanceOf(msg.sender) >= value, 'balance too low');
        balances[to] += value;
        balances[msg.sender] -= value;
        emit Transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint value) public returns (bool) {
        assert((balanceOf(from) >= value));
        assert(((balanceOf(from) >= value));
        require((balanceOf(from) >= value, 'balance too low');
        assert(((allowance[from][msg.sender] >= value));
        assert(((allowance[from][msg.sender] >= value));
        require((allowance[from][msg.sender] >= value, 'allowance too low');
        balances[to] += value;
        balances[from] -= value;
        emit Transfer(from, to, value);
        return true;
    }

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

Listing 2: Annotated Smart Contract Token_mod.sol

The annotated Smart Contract is supplied into SolBMC which is a solidity compiler with SMTChecker engine, that is, Bounded Model Checker. This component generates the Execution Report as shown in Listing 3. If a target is reachable and is feasible the SMTChecker engine will detect it and give the counter example for that target. Here, the counter example shows that the constraint has the model and the test input values for the variables in that constraints can be generated. The warning message “Warning: BMC: Assertion violation happens here,” highlights the detected target along with the specific line number in the source code of the smart contract.

```solidity
pragma solidity ^0.8.2;
contract Token {
    mapping(address => uint) public balances;
    mapping(address => mapping(address => uint)) public allowance;
    uint public totalSupply = 100000000000 * 10 ** 18;

    string public name = "ELONBUCCS";
    string public symbol = "BUC";
    uint public decimals = 18;

    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);

    constructor() {
        balances[msg.sender] = totalSupply;
    }

    function balanceOf(address owner) public view returns (uint) {
        return balances[owner];
    }

    function transfer(address to, uint value) public returns (bool) {
        assert((balanceOf(msg.sender) >= value));
        assert(((balanceOf(msg.sender) >= value));
        require((balanceOf(msg.sender) >= value, 'balance too low');
        balances[to] += value;
        balances[msg.sender] -= value;
        emit Transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint value) public returns (bool) {
        assert((balanceOf(from) >= value));
        assert(((balanceOf(from) >= value));
        require((balanceOf(from) >= value, 'balance too low');
        assert(((allowance[from][msg.sender] >= value));
        assert(((allowance[from][msg.sender] >= value));
        require((allowance[from][msg.sender] >= value, 'allowance too low');
        balances[to] += value;
        balances[from] -= value;
        emit Transfer(from, to, value);
        return true;
    }

    function approve(address spender, uint value) public returns (bool) {
        allowance[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true;
    }
}
```
Note: Counterexample:
-false
balances[owner]=9725, decimals=0,
from=32278, owner=32278, ...

Warning: BMC: Assertion violation happens here.
-->.Results/Token/Token_mod.sol:29:2:
assert(!{(balanceOf(from) =>
value)});

Warning: BMC: Assertion violation happens here.
-->.Results/Token/Token_mod.sol:31:2:
assert(!{(allowance[from][msg.sender] =>
value)});

Warning: BMC: Assertion violation happens here.
-->.Results/Token/Token_mod.sol:33:2:
assert(!{(allowance[from][msg.sender] =>
value)});

Note: Counterexample:
-allowance[from][msg.sender]=2331,
balances[owner]=14136, decimals=0,
from=22114, owner=22114, to=0,
totalSupply=0, value=0

Warning: BMC: Assertion violation happens here.
-->.Results/Token/Token_mod.sol:35:2:
assert(!{(allowance[from][msg.sender] =>
value)});

Note: Counterexample:
-allowance[from][msg.sender]=2747,
balances[owner]=2748, decimals=0,
from=28871, owner=28871, to=0,
totalSupply=0, value=0

---

Table 1: Generated test cases for Listing 2.

<table>
<thead>
<tr>
<th>Value</th>
<th>TC1</th>
<th>TC2</th>
<th>TC3</th>
<th>TC4</th>
<th>TC5</th>
<th>TC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>38</td>
<td>2057</td>
<td>2785</td>
<td>2163</td>
<td>14136</td>
<td>22114</td>
</tr>
<tr>
<td>owner</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>totalSupply</td>
<td>0</td>
<td>20537</td>
<td>2748</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>decimals</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>


In our analysis, we have also captured the execution time of testing the smart contract. The time analysis for the working analysis is shown in Listing 5. We can observe that SolBMC took 1.36 sec for this example. It is to be noted that our experiment has a 1-hour timeout, so to certify any smart contract as Optimal or Maximal it is required to finish the execution within 1 Hr without any out of resources.

```
***Time Analysis Report - Start***
1 assert inserted : 6
2 assertion violation detected (dynamic) : 6
3 assertion violation detected (unique) : 6

***Time Analysis Report - End***
```

Listing 5: Time Analysis for Listing 2.

Finally, we extract all the counter-examples or test inputs as shown in Table 1. The variables of the smart contract are taken as a working example. TC1 to TC6 represents all the 6 test cases corresponding to the 6 targets found. The dash “-” symbol shows that the value of the variable is missing because the variable was not present in the constraint checked. These test cases are very much useful for post-processing, and performing a mutation testing analysis. Also, there may be many applications that might require test cases.

5 EXPERIMENTAL RESULTS

In this section, we discuss the setup and benchmarks tested, and demonstration scenarios for our system.

We used an Intel® Core™ i7-9700 CPU @ 3.00GHz Linux box (64-bit Ubuntu 20.04.2 LTS)
with 8 GB RAM and llvmpipe (LLVM 11.0.0, 256 bits) graphics in Oracle Virtualisation. We have used PPAs for Ubuntu with the latest stable version of Solidity Compiler. We have used the following command setting as shown in Listing 6:

```bash
solc ASC.sol --model-checker-engine bmc
```

Listing 6: Command Line for SolBMC.

There are a total of 70 smart contracts were collected and tested. 60 smart contracts were taken from etherscan, 2021. These contracts are real-time smart contracts and we found them suitable to work on. Second, we consider a set of 6 contracts from (Azrune, 2021). Other 4 contracts that we considered are: Ballot.sol (Ballot.sol, 2021), escrow.sol (escrow.sol, 2021), payments.sol (payments.sol, 2021), and ptest.sol (ptest.sol, 2021).

We divide the results on 70 smart contracts with 1 Hr timeout, into three different groups:

- **Group 1**: In this group, we have contracts with condition coverage lesser than 100%, and the SolBMC finishes the execution within the 1 Hr timeout. The rest conditions are uncovered or dead. It means the condition coverage achieved is Optimal.

- **Group 2**: In this group, we have the contracts with 100% condition coverage, and the SolBMC finishes the execution within the 1 Hr timeout. It means the condition coverage achieved is Maximal.

- **Group 3**: In this group we have the contracts with 0% condition coverage and the SolBMC could not finish the execution due to resource/time restrictions for quarries. It means the condition coverage achieved is Incomplete.

Tables 2, 3, and 4 show the detailed results for 70 smart contracts. #LOCs and #MLOCs present the size of the contract before and after the annotations for targets respectively. Here, #LOCs show the Lines of Code, and #MLOCs show Modified Lines of Code. Note that the targets or goal constraints are injected into contracts, in a way that the semantics of the contract will not be affected. #TB shows the total branch edges for the conditions in the contract. #TDBC and #TUBC show the covered branch edges. The #TDBC shows the branches covered dynamically after executing the conditions more than once, whereas, #TUBC shows the unique branches covered. CC% shows the Condition Coverage score calculated using Eq. 1.

$$\text{CC\%} = \frac{\# \text{TUBC}}{\# \text{TB}} \times 100$$  \hspace{1cm} (1)

<table>
<thead>
<tr>
<th>Contracts</th>
<th>#L</th>
<th>#ML</th>
<th>#B</th>
<th>#DB</th>
<th>#UB</th>
<th>CC%</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnySwapV1 ERC20</td>
<td>357</td>
<td>446</td>
<td>88</td>
<td>216</td>
<td>42</td>
<td>47</td>
<td>18.94</td>
</tr>
<tr>
<td>Ballot</td>
<td>79</td>
<td>106</td>
<td>26</td>
<td>26</td>
<td>23</td>
<td>88</td>
<td>3.36</td>
</tr>
<tr>
<td>TokenVesting</td>
<td>177</td>
<td>238</td>
<td>60</td>
<td>93</td>
<td>58</td>
<td>96</td>
<td>5.80</td>
</tr>
<tr>
<td>MiraNit</td>
<td>300</td>
<td>481</td>
<td>90</td>
<td>167</td>
<td>56</td>
<td>62</td>
<td>262.06</td>
</tr>
<tr>
<td>COINNetwork</td>
<td>452</td>
<td>510</td>
<td>72</td>
<td>240</td>
<td>68</td>
<td>94</td>
<td>19.13</td>
</tr>
<tr>
<td>CyberFox</td>
<td>199</td>
<td>248</td>
<td>48</td>
<td>129</td>
<td>46</td>
<td>95</td>
<td>6.30</td>
</tr>
<tr>
<td>wLinSale</td>
<td>196</td>
<td>245</td>
<td>50</td>
<td>97</td>
<td>48</td>
<td>96</td>
<td>17.75</td>
</tr>
<tr>
<td>WrappedToken</td>
<td>470</td>
<td>569</td>
<td>98</td>
<td>180</td>
<td>56</td>
<td>57</td>
<td>262.41</td>
</tr>
<tr>
<td>Address</td>
<td>504</td>
<td>623</td>
<td>118</td>
<td>169</td>
<td>53</td>
<td>44</td>
<td>1267.10</td>
</tr>
<tr>
<td>eMuppy</td>
<td>174</td>
<td>213</td>
<td>42</td>
<td>140</td>
<td>30</td>
<td>90</td>
<td>6.91</td>
</tr>
<tr>
<td>MLComToken</td>
<td>155</td>
<td>184</td>
<td>28</td>
<td>108</td>
<td>26</td>
<td>92</td>
<td>4.50</td>
</tr>
<tr>
<td>Galaxium</td>
<td>174</td>
<td>209</td>
<td>34</td>
<td>92</td>
<td>35</td>
<td>97</td>
<td>5.11</td>
</tr>
<tr>
<td>RIAS</td>
<td>203</td>
<td>248</td>
<td>44</td>
<td>130</td>
<td>42</td>
<td>95</td>
<td>5.45</td>
</tr>
<tr>
<td>kaqanz</td>
<td>38</td>
<td>51</td>
<td>12</td>
<td>15</td>
<td>11</td>
<td>91</td>
<td>3.26</td>
</tr>
</tbody>
</table>

Table 2: Result Analysis for Optimal group.

<table>
<thead>
<tr>
<th>Contracts</th>
<th>#L</th>
<th>#ML</th>
<th>#B</th>
<th>#DB</th>
<th>#UB</th>
<th>CC%</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>payments</td>
<td>36</td>
<td>45</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>100</td>
<td>3.02</td>
</tr>
<tr>
<td>KyreTrader</td>
<td>152</td>
<td>180</td>
<td>30</td>
<td>81</td>
<td>30</td>
<td>100</td>
<td>5.79</td>
</tr>
<tr>
<td>MacyOchi</td>
<td>308</td>
<td>333</td>
<td>24</td>
<td>75</td>
<td>24</td>
<td>100</td>
<td>4.68</td>
</tr>
<tr>
<td>PipeCtn</td>
<td>125</td>
<td>142</td>
<td>20</td>
<td>96</td>
<td>20</td>
<td>100</td>
<td>4.27</td>
</tr>
<tr>
<td>SBC</td>
<td>117</td>
<td>138</td>
<td>20</td>
<td>96</td>
<td>20</td>
<td>100</td>
<td>4.68</td>
</tr>
<tr>
<td>GLISTh</td>
<td>156</td>
<td>179</td>
<td>22</td>
<td>61</td>
<td>22</td>
<td>100</td>
<td>3.91</td>
</tr>
<tr>
<td>Salvador</td>
<td>56</td>
<td>63</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>3.04</td>
</tr>
<tr>
<td>ShibaAstronaut</td>
<td>37</td>
<td>44</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>3.15</td>
</tr>
<tr>
<td>StarNFTProxy</td>
<td>169</td>
<td>196</td>
<td>26</td>
<td>106</td>
<td>26</td>
<td>100</td>
<td>9.47</td>
</tr>
<tr>
<td>UniswapV1MigratoryProxy</td>
<td>16</td>
<td>19</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>3.09</td>
</tr>
<tr>
<td>VampireDoge</td>
<td>37</td>
<td>44</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>2.83</td>
</tr>
<tr>
<td>BasicProvenance</td>
<td>42</td>
<td>53</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>2.86</td>
</tr>
<tr>
<td>ClockRenContact</td>
<td>192</td>
<td>237</td>
<td>37</td>
<td>84</td>
<td>37</td>
<td>100</td>
<td>1.74</td>
</tr>
<tr>
<td>DigitalLocker</td>
<td>128</td>
<td>149</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>100</td>
<td>2.91</td>
</tr>
<tr>
<td>ERC20</td>
<td>308</td>
<td>333</td>
<td>24</td>
<td>78</td>
<td>24</td>
<td>100</td>
<td>4.51</td>
</tr>
<tr>
<td>AssetTransfer</td>
<td>191</td>
<td>236</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>100</td>
<td>4.07</td>
</tr>
<tr>
<td>RefrigeratedTransportation</td>
<td>117</td>
<td>150</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>100</td>
<td>3.45</td>
</tr>
<tr>
<td>escrow</td>
<td>43</td>
<td>52</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>100</td>
<td>3.00</td>
</tr>
<tr>
<td>ptest</td>
<td>17</td>
<td>28</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>2.87</td>
</tr>
<tr>
<td>SimpleMarketplace</td>
<td>61</td>
<td>74</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>100</td>
<td>3.04</td>
</tr>
<tr>
<td>RoomThermistor</td>
<td>42</td>
<td>55</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>100</td>
<td>3.09</td>
</tr>
<tr>
<td>Token</td>
<td>37</td>
<td>44</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>3.47</td>
</tr>
<tr>
<td>DogeMojo</td>
<td>37</td>
<td>44</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>3.11</td>
</tr>
<tr>
<td>Beno</td>
<td>156</td>
<td>185</td>
<td>28</td>
<td>130</td>
<td>28</td>
<td>100</td>
<td>6.29</td>
</tr>
<tr>
<td>FronRunner</td>
<td>265</td>
<td>270</td>
<td>4</td>
<td>14</td>
<td>4</td>
<td>100</td>
<td>3.61</td>
</tr>
<tr>
<td>BurnableERC20</td>
<td>167</td>
<td>185</td>
<td>24</td>
<td>111</td>
<td>24</td>
<td>100</td>
<td>6.71</td>
</tr>
</tbody>
</table>

Table 3: Result Analysis for Maximal group.

Figure 2: Condition Coverage scores for Groups 1 and 2 contracts.

We compute the total execution time (seconds) to analyse the contract. Table 2 shows the result analysis for the Optimal group of 14 smart contracts. Here the optimal means the executor has finished the execution
the executor has finished the execution within the time out given and achieved 100% condition coverage. It means no part of the code is dead. This is the expectation of achieving 100% for any contributor writing a smart contract. In aggregate for 26 contracts, a total of 474 branches were checked and 474 branches were uniquely covered in 105.03 sec. On average Condition Coverage for Group 2 is 100% in an average time of 4.03 sec.

Collectively, in Groups 1 and 2 we have 40 contracts for which we were able to compute condition coverage in reasonable time. Except for 5 contracts out of 40 contracts the condition coverage is below 90%. That is, 87.5% of contracts from Groups 1 and 2 have a good condition coverage score and the level of quality and reliability of the contract written is ensured. Fig. 2 shows the Condition Coverage computed for the contracts from Groups 1 and 2. Fig. 3 shows the chart in logarithm format for Lines of Code vs. Execution time in sec (increasing order). In most cases the bigger contracts take more execution time as shown in Fig. 3.

Table 4 shows the result analysis for the Incomplete group of 30 smart contracts. Here the Incomplete means the executor has not finished the execution due to some resource restrictions. The average LOCs for 30 contracts is 400 with 82 average branches. So, it is showing the complexness of the contracts for which the queries are too long and complex so the solidity bounded model checker can not process, and the process has to be killed. The solidity compiler checks the resources of the machine during the process, and if some resources are out then the query cannot be processed. Also, BMC can only report anything once it is finished. But for Group 3, SolBMC has run out of resources so no coverage information has been generated and the values of #TDBC, #TUBC, and CC% are 0 (highlighted in red color). Note that, for this group, none of the contracts reached 3600 sec. In case any contract would have reached timeout then we would have included that contract in Group 3.

Table 5 shows the aggregate and average results for all the 70 contracts in Groups 1, 2, and 3. So, in aggregate we processed 18.31 KLOCs of original smart contracts and executed 22.06 KLOCs. There was a total number of 3.76K Branch edges out of which 1.07K Branch edges were covered uniquely. Also, the same atomic conditions can be executed several times in the contract due to several calls, hence dynamically 2.95K Branch edges were covered. We have achieved 53.4% of the average condition coverage score for 70 contracts. These 70 contracts took 1.85 Hrs to finish the process. Finally, we claim that for 40 contracts

within the time out given. The condition coverage is lesser than 100%, which means the uncovered code cannot be reached or executed for the contract, hence the dead code. In aggregate for 14 contracts, a total of 810 branches were checked and 600 branches were uniquely covered in 1908.28 sec. On average Condition Coverage for Group 1 is 81.74% in an average time of 136.30 sec.

Table 3 shows the result analysis for the Maximal group of 26 smart contracts. Here the maximal means

<table>
<thead>
<tr>
<th>Contracts</th>
<th>#LOCs</th>
<th>#MLOCs</th>
<th>#TBranches</th>
<th>#TDBC</th>
<th>#TUBC</th>
<th>Avg_CC%</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eNew</td>
<td>329</td>
<td>395</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>189.41</td>
<td></td>
</tr>
<tr>
<td>DogeRocket</td>
<td>475</td>
<td>566</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>118.23</td>
<td></td>
</tr>
<tr>
<td>ChinaCoin</td>
<td>330</td>
<td>413</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>127.87</td>
<td></td>
</tr>
<tr>
<td>BERNIE</td>
<td>356</td>
<td>443</td>
<td>86</td>
<td>0</td>
<td>0</td>
<td>181.43</td>
<td></td>
</tr>
<tr>
<td>shibabread</td>
<td>330</td>
<td>413</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>127.37</td>
<td></td>
</tr>
<tr>
<td>AIBet</td>
<td>301</td>
<td>362</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>158.14</td>
<td></td>
</tr>
<tr>
<td>Amalaka</td>
<td>616</td>
<td>727</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>318.80</td>
<td></td>
</tr>
<tr>
<td>TESTDONTBUY</td>
<td>349</td>
<td>452</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>118.54</td>
<td></td>
</tr>
<tr>
<td>SimpleEER</td>
<td>154</td>
<td>171</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>2.94</td>
<td></td>
</tr>
<tr>
<td>EStack</td>
<td>330</td>
<td>413</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>125.74</td>
<td></td>
</tr>
<tr>
<td>goldiine</td>
<td>335</td>
<td>426</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>230.18</td>
<td></td>
</tr>
<tr>
<td>Kronadou</td>
<td>475</td>
<td>566</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>114.05</td>
<td></td>
</tr>
<tr>
<td>LILY</td>
<td>321</td>
<td>385</td>
<td>68</td>
<td>0</td>
<td>0</td>
<td>182.94</td>
<td></td>
</tr>
<tr>
<td>Lunar</td>
<td>324</td>
<td>395</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>167.65</td>
<td></td>
</tr>
<tr>
<td>PONY</td>
<td>329</td>
<td>395</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>168.77</td>
<td></td>
</tr>
<tr>
<td>PORCUPINE</td>
<td>329</td>
<td>395</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>169.85</td>
<td></td>
</tr>
<tr>
<td>PROGEV2</td>
<td>356</td>
<td>437</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>142.74</td>
<td></td>
</tr>
<tr>
<td>Ryujin</td>
<td>349</td>
<td>452</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>100.71</td>
<td></td>
</tr>
<tr>
<td>SATURNITE</td>
<td>330</td>
<td>413</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>112.18</td>
<td></td>
</tr>
<tr>
<td>Shibalai</td>
<td>469</td>
<td>560</td>
<td>90</td>
<td>0</td>
<td>0</td>
<td>108.29</td>
<td></td>
</tr>
<tr>
<td>Shibakuyo</td>
<td>731</td>
<td>844</td>
<td>112</td>
<td>0</td>
<td>0</td>
<td>161.05</td>
<td></td>
</tr>
<tr>
<td>ShibSamurai</td>
<td>539</td>
<td>616</td>
<td>76</td>
<td>0</td>
<td>0</td>
<td>219.90</td>
<td></td>
</tr>
<tr>
<td>SOTH</td>
<td>330</td>
<td>413</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>119.82</td>
<td></td>
</tr>
<tr>
<td>Thicc</td>
<td>271</td>
<td>326</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>2.62</td>
<td></td>
</tr>
<tr>
<td>TOAD</td>
<td>321</td>
<td>385</td>
<td>68</td>
<td>0</td>
<td>0</td>
<td>175.07</td>
<td></td>
</tr>
<tr>
<td>WickedCrannums</td>
<td>626</td>
<td>745</td>
<td>132</td>
<td>0</td>
<td>0</td>
<td>357.70</td>
<td></td>
</tr>
<tr>
<td>HOTDoge</td>
<td>352</td>
<td>435</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>237.88</td>
<td></td>
</tr>
<tr>
<td>HotinuFinance</td>
<td>730</td>
<td>843</td>
<td>112</td>
<td>0</td>
<td>0</td>
<td>144.59</td>
<td></td>
</tr>
<tr>
<td>KOALA</td>
<td>329</td>
<td>395</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>168.48</td>
<td></td>
</tr>
<tr>
<td>eMastiff</td>
<td>321</td>
<td>414</td>
<td>92</td>
<td>0</td>
<td>0</td>
<td>105.30</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Aggregate and Average Result Analysis for 70 contracts.

<table>
<thead>
<tr>
<th>Groups</th>
<th>BLOCs</th>
<th>DLOCs</th>
<th>Bit</th>
<th>FTDBC</th>
<th>FTUBC</th>
<th>Avg_CC%</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>3568</td>
<td>4711</td>
<td>410</td>
<td>1802</td>
<td>560</td>
<td>81.71</td>
<td>1988.28</td>
</tr>
<tr>
<td>Group 2</td>
<td>3012</td>
<td>3498</td>
<td>474</td>
<td>1151</td>
<td>474</td>
<td>100</td>
<td>105.03</td>
</tr>
<tr>
<td>Group 3</td>
<td>11737</td>
<td>14195</td>
<td>2484</td>
<td>0</td>
<td>0</td>
<td>4657.33</td>
<td></td>
</tr>
<tr>
<td>#Total</td>
<td>18317</td>
<td>22084</td>
<td>3768</td>
<td>2953</td>
<td>1874</td>
<td>53.4</td>
<td>6678.65</td>
</tr>
</tbody>
</table>
(57.14\%) out of 70 contracts we have good condition coverage scores.

6 CONCLUSION

The main objective of this work is to certify the smart contract wrt. the condition coverage analysis. It is very important to test smart contracts by looking at the critical business in the blockchain. If an incorrect contract or bug in the contract exists, then there is a high chance of losing the expensive assets. In this paper, we proposed a novel approach to compute condition coverage for a smart contract using a solidity compiler with a bounded model checker. We tested 70 contracts and showed that for 40 contracts i.e., 57.14\% have optimal or maximal condition coverage scores. However, the rest 30 contracts were incomplete due to resource issues. We will explore other techniques such as Fuzzing and Symbolic execution for a more detailed analysis.

ACKNOWLEDGEMENT

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

REFERENCES


