CC-SolBMC: Condition Coverage Analysis for Smart Contracts Using Solidity Bounded Model Checker

Sangharatna Godboley[a] and P. Radha Krishna[b]
NITMiner Technologies, Department of Computer Science and Engineering,
National Institute of Technology,
Warangal, Telangana, India

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-

[a] https://orcid.org/0000-0002-6169-6334
[b] https://orcid.org/0000-0001-8298-7571

[1] "The Go programming language," https://golang.org/.
[2] "Hyperledger project," https://www.hyperledger.org/.
[3] "Solidity smart-contract language," https://solidity.readthedocs.io/.

Godboley, S. and Krishna, P.
CC-SolBMC: Condition Coverage Analysis for Smart Contracts Using Solidity Bounded Model Checker.
DOI: 10.5220/0012627200003687
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 387-395
ISBN: 978-989-758-696-5; ISSN: 2184-4895
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

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 (eg. (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. Praitheeshan 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 IBM[4] "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 NIST [5]

[4]https://www.ibm.com/topics/smart-contracts

[5]https://csrc.nist.gov/glossary/term/Condition_coverage

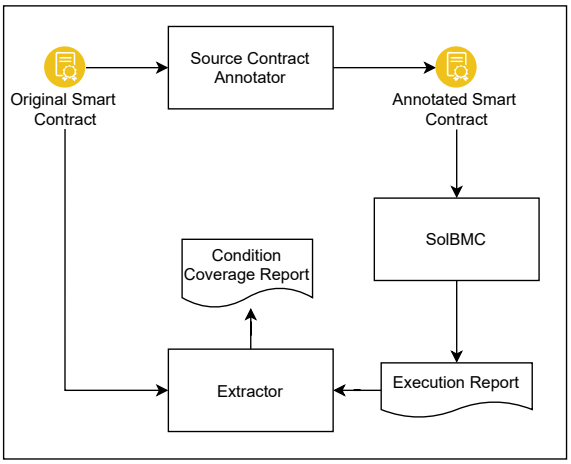

Figure 1: The framework for CC-SolBMC.

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 constraints solver, 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  while Condition ∈ AllConditions do
2      if TRUE Branch of Condition then
3          Create "assert(!(condition));"
4      end
5      if FALSE Branch of Condition then
6          Create "assert(!(!(condition)));"
7      end
8      Inject "assert(!(condition));" and
         "assert(!(!(condition)));" above the line
         where the Condition was identified;
9  end
10 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]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:

balanceOf(msg.sender) >= value

balanceOf(from) >= value

allowance[from][msg.sender] >= 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 Ist Condition:

balanceOf(msg.sender) >= value

TRUE branch of Ist Condition:

assert(!((balanceOf(msg.sender) >= value)));

FALSE branch of Ist Condition:

assert(!(!(balanceOf(msg.sender) >= value)));

For IInd Condition:

balanceOf(from) >= value

TRUE branch of IInd Condition:

assert(!((balanceOf(from) >= value)));

FALSE branch of IInd Condition:

assert(!(!(balanceOf(from) >= value)));

For IIIrd Condition:

allowance[from][msg.sender] >= value

TRUE branch of IIIrd Condition:

assert(!((allowance[from][msg.sender] >= value)));

FALSE branch of IIIrd 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.

```
1  pragma solidity ^0.8.2;
2  contract Token {
3  mapping(address => uint) public
       balances;
4  mapping(address => mapping(address =>
       uint)) public allowance;
5  uint public totalSupply = 100000000000 *
       10 ** 18;
6  string public name = "ELONBUCCS";
7  string public symbol = "BUC";
8  uint public decimals = 18;
9  event Transfer(address indexed from,
       address indexed to, uint value);
10 event Approval(address indexed owner,
       address indexed spender, uint
       value);
11 constructor() {
12    balances[msg.sender] = totalSupply;
13 }
14 function balanceOf(address owner)
       public view returns(uint) {
15    return balances[owner];
16 }
17 function transfer(address to, uint
       value) public returns(bool) {
18    require(balanceOf(msg.sender) >=
          value, 'balance too low');
19    balances[to] += value;
20    balances[msg.sender] -= value;
21   emit Transfer(msg.sender, to,
         value);
22    return true;
23 }
24 function transferFrom(address from,
       address to, uint value) public
       returns(bool) {
25    require(balanceOf(from) >= value,
          'balance too low');
26    require(allowance[from][msg.sender]
          >= value, 'allowance too low');
27    balances[to] += value;
28    balances[from] -= value;
29    emit Transfer(from, to, value);
30    return true;
31 }
32 function approve(address spender, uint
       value) public returns (bool) {
33    allowance[msg.sender][spender] =
          value;
```

```
34      emit Approval(msg.sender, spender,
            value);
35      return true;
36  }
37
```

Listing 1: Original Smart Contract *Token.sol*.

```
1  pragma solidity ^0.8.2;
2  contract Token {
3  mapping(address => uint) public
       balances;
4  mapping(address => mapping(address =>
       uint)) public allowance;
5  uint public totalSupply = 100000000000 *
       10 ** 18;
6  string public name = "ELONBUCCS";
7  string public symbol = "BUC";
8  uint public decimals = 18;
9  event Transfer(address indexed from,
       address indexed to, uint value);
10 event Approval(address indexed owner,
       address indexed spender, uint
       value);
11 constructor() {
12     balances[msg.sender] = totalSupply;
13 }
14 function balanceOf(address owner)
       public view returns(uint) {
15     return balances[owner];
16 }
17 function transfer(address to, uint
       value) public returns(bool) {
18 assert(!(balanceOf(msg.sender) >=
       value));
19 assert(!(!(balanceOf(msg.sender) >=
       value)));
20     require(balanceOf(msg.sender) >=
           value, 'balance too low');
21     balances[to] += value;
22     balances[msg.sender] -= value;
23   emit Transfer(msg.sender, to,
         value);
24     return true;
25 }
26 function transferFrom(address from,
       address to, uint value) public
       returns(bool) {
27 assert(!(balanceOf(from) >= value));
28 assert(!(!(balanceOf(from) >= value)));
29     require(balanceOf(from) >= value,
           'balance too low');
30 assert(!(allowance[from][msg.sender] >=
       value));
31 assert(!(!(allowance[from][msg.sender]
       >= value)));
32     require(allowance[from][msg.sender]
           >= value, 'allowance too low');
33     balances[to] += value;
34     balances[from] -= value;
35     emit Transfer(from, to, value);
36     return true;
37 }
38 function approve(address spender, uint
       value) public returns (bool) {
39     allowance[msg.sender][spender] =
           value;
40     emit Approval(msg.sender, spender,
           value);
41     return true;
42 }
43
```

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.

```
1  Warning: BMC: Assertion violation happens
       here.
2    --> ./Results/Token/Token_mod.sol:19:2:
3     |
4  19 |  assert(!(balanceOf(msg.sender) >=
       value));
5     |  ^^^^^^^^^^^^^^^^^^^^^^^^^
6  Note: Counterexample:
7    = false
8    balances[owner]=38, decimals=0,
         owner=28100, to=0, totalSupply=0,
         value=0
9  ........................skip..
10 Warning: BMC: Assertion violation happens
       here.
11   --> ./Results/Token/Token_mod.sol:20:2:
12    |
13 20 |  assert(!(!(balanceOf(msg.sender)
       >= value)));
14    |  ^^^^^^^^^^^^^^^^^^^^^^^^^
15 Note: Counterexample:
16   =false
17   balances[owner]=20537, decimals=0,
         owner=1323, to=0, totalSupply=0,
         value=20538
18 ........................skip..
19 Warning: BMC: Assertion violation happens
       here.
20   --> ./Results/Token/Token_mod.sol:28:2:
21    |
22 28 |  assert(!(balanceOf(from) >=
       value));
23    |  ^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Note: Counterexample:
  =false
  balances[owner]=9725, decimals=0,
       from=32278, owner=32278, to=0,
       totalSupply=0, value=0
.......................skip..
Warning: BMC: Assertion violation happens
    here.
  --> ./Results/Token/Token_mod.sol:29:2:
   |
29 |  assert(!(!(balanceOf(from) >=
    value)));
   |  ^^^^^^^^^^^^^^^^^^^^^^^^^
Note: Counterexample:
  =false
  balances[owner]=17887, decimals=0,
       from=31597, owner=31597
  to=0, totalSupply=0, value=17888
.......................skip..
Warning: BMC: Assertion violation happens
    here.
  --> ./Results/Token/Token_mod.sol:31:2:
   |
31 |
    assert(!(allowance[from][msg.sender]
    >= value));
   |  ^^^^^^^^^^^^^^^^^^^^^^^^^
Note: Counterexample:
  =false
  allowance[from][msg.sender]=2331,
       balances[owner]=14136, decimals=0,
       from=22114, owner=22114, to=0,
       totalSupply=0, value=0
.......................skip..
Warning: BMC: Assertion violation happens
    here.
  --> ./Results/Token/Token_mod.sol:32:2:
   |
32 |  assert(!(!(allowance [from]
    [msg.sender] >= value)));
   |  ^^^^^^^^^^^^^^^^^^^^^^^^^
Note: Counterexample:
  =false
  allowance[from][msg.sender]=2747,
       balances[owner]=2748, decimals=0,
       from=28871, owner=28871, to=0,
       totalSupply=0, value=2748
```

Listing 3: Execution Report with counter examples Listing 2.

Now, the execution report generated is supplied into the *Extractor* component along with the original smart contract to get the condition coverage report as shown in Listing 4. As we have injected 6 targets, SolBMC has detected all of them with counterexamples. We have extracted the results and found that there were 6 targets found dynamically and uniquely. Here, we provide both the counting of dynamic and unique targets. It is to be noted that, *dynamic* means a target can be reached multiple times through different paths or traces. Whereas, *unique* means a target at least reached through a path or trace once. This counting is important to compute the condition coverage because statically we know how many targets or branch edges we have and out of them how many we have detected. So for this contract, we have a **100%** condition coverage score, which means it is *Maximal*.

Table 1: Generated test cases for Listing 2.

Variables	TC1	TC2	TC3	TC4	TC5	TC6
allowance[from][msg.sender]	-	-	-	-	2331	2747
balances[owner]	38	20537	9725	17887	14136	2748
decimals	0	0	0	0	0	0
from	-	-	32278	31597	22114	28871
owner	28100	1323	32278	31597	22114	28871
to	0	0	0	0	0	0
totalSupply	0	0	0	0	0	0
value	0	20538	0	17888	0	2748

```
1 assertion inserted : 6
2 assertion violation detected (dynamic) : 6
3 assertion violation detected (unique) : 6
```

Listing 4: Condition Coverage Report Listing 2.

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.

```
1 ***Time Analysis Report - Start***
2 ***Total runtime in seconds 1.365783207
3 Total runtime: 0:00:00:1.3658
4 ***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[7]. We have used the following command setting as shown in Listing 6:

```
solc ASC.sol --model-checker-engine bmc
    --model-checker-targets assert
```

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 (Azure, 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.

$$CC\% = \frac{\#TUBC}{\#TB} \quad (1)$$

Table 2: Result Analysis for *Optimal* group.

Contracts	#L	#ML	#B	#DB	#UB	CC%	T(sec)
AnyswapV5ERC20	357	446	88	216	42	47	18.94
Ballot	79	106	26	26	23	88	3.36
TokenVesting	177	238	60	93	58	96	5.80
MiraNft	390	481	90	167	56	62	262.06
COINNetwork	452	510	72	240	68	94	19.33
CyberFox	199	248	48	129	46	95	6.30
wLitiSale	196	245	50	97	48	96	17.75
WrappedToken	470	569	98	180	56	57	262.41
Address	504	623	118	169	53	44	1287.10
eMuppy	174	213	42	140	38	90	6.91
MJCoinToken	155	184	28	108	26	92	4.50
Galaxium	174	209	34	92	33	97	5.11
RIAS	203	248	44	130	42	95	5.45
kiaquiz	38	51	12	15	11	91	3.26

Table 3: Result Analysis for *Maximal* group.

Contracts	#L	#ML	#B	#DB	#UB	CC%	T(sec)
payments	36	45	8	8	8	100	3.02
Kyuseishu	152	180	30	81	30	100	5.77
MayoOcho	308	333	24	73	24	100	4.68
PipiCoin	125	142	20	96	20	100	4.27
RBC	117	138	20	96	20	100	4.68
GOLIATH	156	179	22	61	22	100	3.91
salvador	56	63	6	6	6	100	3.04
ShibaAstronaut	37	44	6	6	6	100	3.15
StarNFTProxy	169	196	26	106	26	100	9.47
UniswapV3MigratorProxy	16	19	2	2	2	100	3.09
VamprireDoge	37	44	6	6	6	100	2.81
BasicProvenance	42	53	10	10	10	100	2.86
ClockBoxContract	192	217	24	74	24	100	4.16
DigitalLocker	128	149	20	20	20	100	2.91
ERC20	308	333	24	73	24	100	4.51
AssetTransfer	191	276	84	84	84	100	4.07
RefrigeratedTransportation	117	150	32	32	32	100	3.45
escrow	43	52	8	16	8	100	3.00
ptest	17	28	10	10	10	100	2.87
SimpleMarketplace	61	74	12	12	12	100	3.04
RoomThermostat	42	55	12	12	12	100	3.09
Token	37	44	6	6	6	100	3.47
DogeMojo	37	44	6	6	6	100	3.11
Benu	156	185	28	130	28	100	6.29
FrontRunner	265	270	4	14	4	100	3.61
BurnableERC20	167	185	24	111	24	100	6.71

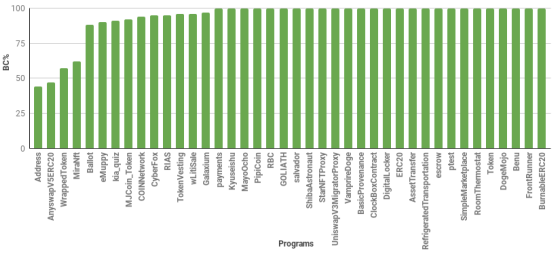

Figure 2: Condition Coverage scores for Groups 1 and 2 contracts.

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

[7]https://docs.soliditylang.org/en/develop/installing-solidity.html#linux-packages

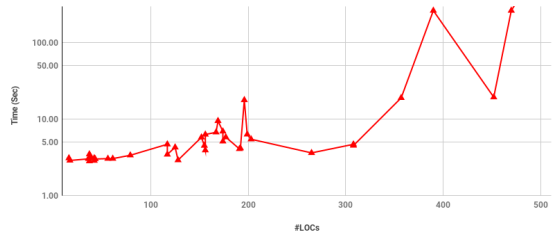

Figure 3: Execution time in sec for Groups 1 and 2 contracts.

Table 4: Result Analysis for *Incomplete* group.

Contracts	#L	#ML	#B	#DB	#UB	CC%	T(sec)
eNew	329	395	70	0	0	0	189.41
DogeRocket	475	566	90	0	0	0	118.23
ChinaCoin	330	413	82	0	0	0	127.87
BERNIE	356	443	86	0	0	0	181.43
shibabread	330	413	82	0	0	0	127.37
AIRBets	301	362	60	0	0	0	158.14
Animalia	616	727	110	0	0	0	318.80
TESTDONTBUY	349	452	102	0	0	0	118.54
SimpleECR20	154	171	22	0	0	0	2.94
EStack	330	413	82	0	0	0	125.74
goldinu	335	426	90	0	0	0	230.18
KizunaInu	475	566	90	0	0	0	114.05
LILY	321	385	68	0	0	0	182.94
Lunar	324	395	70	0	0	0	167.65
PONY	329	395	70	0	0	0	168.77
PORCUPINE	329	395	70	0	0	0	169.85
PROGEV2	356	437	80	0	0	0	142.74
Ryujin	349	452	102	0	0	0	100.71
SATURNITE	330	413	82	0	0	0	112.18
ShibaJail	469	560	90	0	0	0	108.29
ShibaKiyo	731	844	112	0	0	0	161.05
ShibaSamurai	539	616	76	0	0	0	219.00
SOTH	330	413	82	0	0	0	119.82
Thicc	271	326	60	0	0	0	2.62
TOAD	321	385	68	0	0	0	175.07
WickedCraniums	626	745	132	0	0	0	357.70
HOTDOGE	352	435	82	0	0	0	237.88
HotinuFinance	730	843	112	0	0	0	144.59
KOALA	329	395	70	0	0	0	168.48
eMastiff	321	414	92	0	0	0	105.30

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

Groups	#LOCs	#MLOCs	#TB	#TDBC	#TUBC	Avg_CC%	Time (sec)
Group 1	3568	4371	810	1802	600	81.71	1908.28
Group 2	3012	3498	474	1151	474	100	105.03
Group 3	11737	14195	2484	0	0	0	4657.33
#Total	**18317**	**22064**	**3768**	**2953**	**1074**	**53.4**	**6670.65**

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 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

(**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

Akca, S., Rajan, A., and Peng, C. (2019). Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE.

Andesta, E., Faghih, F., and Fooladgar, M. (2020). Testing smart contracts gets smarter. In *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 405–412. IEEE.

Azure (2021). Azure. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.

Ballot.sol (2021). Ballot.sol. https://docs.soliditylang.org/en/v0.4.24/solidity-by-example.html.

Benitez, S., Cogan, J., and Russo, A. (2020). Short paper: Blockcheck the typechain. In *Proceedings of the 15th Workshop on Programming Languages and Analysis for Security*, pages 35–39.

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37).

Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In *TACAS*, pages 168–176. Springer.

Clarke, E., Kroening, D., and Yorav, K. (2003). Behavioral consistency of C and Verilog programs using bounded model checking. In *40th DAC*, pages 368–371. ACM Press.

Dannen, C. (2017). *Introducing Ethereum and solidity*, volume 318. Springer.

escrow.sol (2021). escrow.sol. https://www.geeksforgeeks.org/what-is-escrow-smart-contract/.

etherscan (2021). etherscan. https://etherscan.io/.

Jiang, B., Liu, Y., and Chan, W. (2018). Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE.

Liu, Y., Li, Y., Lin, S.-W., and Yan, Q. (2020). Modcon: a model-based testing platform for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1601–1605.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., and Dinaburg, A. (2019). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE.

Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T. (2020). sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788.

payments.sol (2021). payments.sol. https://github.com/joaquin-alfaro/ethereum-payment-vuejs/blob/develop/dapp/contracts/Payments.sol.

Praitheeshan, P., Pan, L., Yu, J., Liu, J., and Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*.

ptest.sol (2021). ptest.sol. https://www.npmjs.com/package/solfuzz.

Sánchez-Gómez, N., Morales-Trujillo, L., and Valderrama, J. T. (2019). Towards an approach for applying early testing to smart contracts. In *WEBIST*, pages 445–453.

Wen, Z. A. and Miller, A. (2016). Scanning live ethereum contracts for the "unchecked-send" bug. *Hacking Distributed*.

Wood, G. et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32.