# SmartMuVerf: A Mutant Verifier for Smart Contracts

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

Keywords: Smart Contract, Software Testing, Mutation Verification.

Abstract: Smart contracts are the logical programs holding the properties in Blockchain. These Blockchain technologies enable society towards trust-based applications. Smart contracts are prepared between the parties to hold their deals. If the deal held by a smart contract is complex and non-trivial, then there is a high chance of attracting issues and loss of assets. These contracts also consider expensive assets. This necessitates the verification and testing of a smart contract. Since we have the source code of a smart contract, then it is reasonable to apply verification and testing techniques. From the traditional ways, it has been observed that mutation testing is one of the important testing techniques. But, this testing technique suffers from the issues of time and cost. It is true that fault-based testing is a good mechanism to perform. So, looking at the issues we introduce a new technique for Mutation Verification for Smart Contracts. In this paper, we present an approach for measuring the mutation score using a verification approach. We experimented with a total of 10 smart contracts.

## 1 INTRODUCTION

A blockchain is a continuously expanding list of records, known as blocks, that are securely linked together via cryptography (Morris, 2016)(Popper, 2016). A cryptographic hash of the preceding block, a timestamp, and transaction data are all included in each block. This is typically depicted as a Merkel Tree, with leafs representing data nodes. The timestamp verifies that the transaction data was there at the time the block was released, allowing it to be hashed. Because each block contains information about the one before it, they form a chain, with each new block reinforcing the preceding ones. As a result, blockchains are resistant to data tampering since the data in any one block cannot be changed retrospectively without affecting all subsequent blocks.

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 accepting by the respective user. One of the major advantages of blockchain technologies is to avoid tampering with 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, audibility, 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 they 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.

The code for smart contracts is typically written in a high-level programming language such as Go[1]

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

[1] "The Go programming language," https://golang.org/.



Godboley, S. and Krishna, P.
SmartMuVerf: A Mutant Verifier for Smart Contracts.
DOI: 10.5220/0011822200003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 346-353
ISBN: 978-989-758-647-7; ISSN: 2184-4895

for Hyperledger[2] and Solidity[3] for Ethereum. Similar to any other software application, there may be deviations, errors and vulnerabilities in the smart contract logic code. Smart contract-enabled blockchains guarantee that conditions in a smart contract are not modified once they have been written and published. Thus, coding smart contract logic as per the application 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 smart contract logic.

A Smart Contract is a computer program, sometimes described as a transaction protocol, that is designed to execute, control, or document legally significant events and activities in accordance with the provisions of a contract or agreement (Martin and Boris, 2019). The decrease in the need for trusted intermediaries, arbitration and enforcement costs, fraud losses, and purposeful and inadvertent exceptions are all goals of smart contracts. Scripting languages have been supported by numerous cryptocurrencies since Bitcoin, allowing for more complex smart contracts between untrusted parties.

These Smart Contracts are programs that are collectively run by a network of mutually distrusting nodes that use a consensus protocol like proof-of-work or proof-of-stake to digitally enforce agreements between nodes (Maher and van Moorsel, 2019). The smart contract's code cannot be modified or its execution subverted by the nodes or the smart contract's developer. Smart contracts have been used in a variety of industries, including finance, insurance, identity management, and supply chain management since they were conceived and implemented by blockchain platforms like Ethereum and EOS. In our project, we are using Ethereum as the smart contract platform, with smart contracts written in the Solidity programming language.

Smart contracts are vulnerable to hacking because they are entrusted by users with handling and transferring assets of significant value. Because the contract becomes immutable once deployed on the blockchain, such hacking is more dangerous than on a traditional network system, essentially creating a high-risk, high-stake paradigm: the deployed code is virtually impossible to patch, and contracts collectively control billions of dollars in digital assets. For example, there have been numerous well-publicized attacks on Ethereum smart contracts: the reentrancy attack successfully stole $60 million in tokens from a contract, leading to the hard fork that established Ethereum Classic (ETC).

Testing of Smart Contracts is important. The widespread acceptance of smart contracts has cemented their place in the next-generation blockchain technology ecosystem. Writing a correct smart contract, on the other hand, is notoriously difficult. Furthermore, once a network confirms a state-changing transaction, the result is immutable. As a result, extensive testing of a smart contract application is essential prior to its implementation.

Smart contracts are deployed and executed in a blockchain environment, where the transactions are irreversible. Smart contracts themselves are immutable, which ensures their tamper-proof nature. Hence, once the smart contract is deployed, improving the code or fixing the bugs is not possible.

In this work, we propose an approach that ensures conformance of execution as per the smart contract logic, if any, of the execution trace of a smart contract. To do this we propose the Mutation Verification technique. The original smart contract serves as a master smart contract. We create Logical Operator Replacement (LOR) and Relational Operator Replacement (ROR) mutants for Original Smart Contract (OSC). Our approach allows annotating the smart contract with goal constraints or targets in the form of "asserts" that ensure the specification of all created mutants. 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 mutation analysis report. The main contribution of this work is to provide a platform for the contributors of smart contracts to test their scripts.

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 approach. 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), in which Solidity scripting language is used for developing smart contracts. There are few more blockchain platforms such as Hyperledger Fabric (Androulaki et al., 2018), Bitcoin (Böhme et al., 2015), and RootStock (Reighard et al., 2008), that supports the deployment and execution of smart contracts. There are some work such as (Sánchez-Gómez et al., 2019; Liu et al., 2020a) in the domain of smart contract testing.

[2]"Hyperledger project," https://www.hyperledger.org/.

[3]"Solidity smart-contract language," https://solidity.readthedocs.io/.

(Andesta et al., 2020) have proposed a method of testing solidity smart contract using mutation testing and proposed 10 classes of mutation operators. 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 universal mutator tool. (Peng et al., 2019) suggested a full system for analyzing Solidity contracts, querying code creation, and instrumentation. It helps Solidity contract developers and testers create source-level approaches for analysis, code generation, diagnostics, and optimizations.

(Driessen et al., 2021) introduced a method that generates an automatic test suite for stand-alone contracts using two search algorithms. They tested a collection of 36 real-world Contracts to evaluate both the algorithms, a genetic algorithm, and a random search technique. VerX by (Permenev et al., 2020) is a temporal property verification framework. To assess these features, they use symbolic execution and abstract interpretation-based predicate abstraction, as well as their own policy language. The need of extensive testing of smart contracts before implementation is demonstrated by Xinming Wang et al. (Xinming et al., 2019). They proposed the concepts of full transaction basis path set and bounded transaction interactions to represent the basic control flow behaviours of smart contracts. They developed a set of path-based test coverage criteria based on these two ideas. They also conducted a case study to evaluate the effectiveness of the proposed test coverage criteria to random testing and statement coverage testing, finding that k-bounded transaction coverage testing is nearly 55 percent more effective than statement coverage testing in detecting flaws (Xinming et al., 2019). They finished by recognising the paucity of research on how to test smart contract applications systematically.

Existing approaches could not efficiently perform mutation testing for integer overflow in Ethereum Smart Contracts (ESCs), according to Jinlei Sun, et al. (Jinlei et al., 2020). As a result, they presented five specific mutation operators to address those flaws in ESC testing sufficiency detection. The different integer overflow problems can be reliably simulated using these operators. Their empirical study on 40 open-source ESCs to evaluate the effectiveness of the proposed mutation operators revealed that the proposed mutation operators reproduced all 179 integer overflow vulnerabilities in the 40 smart contracts, and the generated mutants had a high compilation pass rate and integer overflow vulnerability generation rate. Furthermore, the generating mutants discovered the flaws in existing testing methodologies for integer overflow vulnerabilities.

Xingya Wang, et al. (Xingya et al., 2019) also emphasise the significance of doing adequate testing of Ethereum Smart Contracts (ESC), and present a multi-objective random and NSGA-II based approach to find cost-effective test-suites. Their approach is the first Pareto minimisation approach to ESC testing, combining the goal of reducing uncovered branches in conventional software with the goal of reducing the time and gas cost of testing ESCs. Their empirical assessment of a collection of smart contracts in eight of the most popular Ethereum Decentralised Applications also confirmed that the proposed ways could drastically cut gas and time costs while maintaining branch coverage.

Gustavo Greico, et al. (Gustavo et al., 2020) introduce Echidna, an open-source smart contract fuzzer that automates the generation of tests to discover assertion and custom property violations. Echidna's major benefit is that it doesn't require any complicated configuration or contract deployment to a local blockchain. It has been used in over ten large-scale paid security audits, with comments from those audits helping to enhance the usability and test generation methodologies. They state that recent research examining and categorising flaws in critical contracts found that fuzzing with custom user-defined properties can detect up to 63 percent of the most severe and exploitable flaws in contracts, implying that smart contract developers and security auditors have a significant need for high-quality, easy-to-use fuzzing. Echidna supports three properties: User-defined, assertion verification, and gas use estimation.

Smart contracts are vulnerable to hacking, according to Jean-Wei Liao et al. (Jian-Wei et al., 2019), because they are difficult to fix and lack assessment standards to ensure their quality. They present SoliAudit, a smart contract vulnerability assessment tool that employs machine learning and fuzz testing. They developed a Gray-box fuzz testing mechanism that includes a fuzzer contract and a simulated blockchain environment for online transaction verification. The accuracy of SoliAudit can approach 90 percent, and the fuzzing can assist detect possible weaknesses, such as reentrancy and arithmetic overflow concerns, according to their real-world evaluation utilising nearly 18k smart contracts from the Ethereum blockchain and CTF samples.

Erfan Andesta et al. (Erfan et al., 2020) suggested a mutation-based testing technique for smart contracts in the Solidity language. They looked at a long list of known defects in Solidity smart contracts and came up with ten different mutation operators based on the actual flaws. Classic Mutation Operators and Solidity Mutation Operators were used (as the Classic Muta-

tion Operators were not enough). Their tests revealed that their operators can regenerate real defects for 10 out of 15 well-known problematic smart contracts.

Model-based Software development, and modelling techniques, such as use cases models and activity diagram models based on Unified Model Languages, are combined by N. Sánchez-Gómez et al. (Sanchez et al., 2019) to simplify and improve the modelling, management, and execution of collaborative business processes between multiple companies in the Blockchain network. Current testing and analysis methods, according to Ye Liu et al. (Liu et al., 2020b), lack support for contracts that are frequently larger and more complicated. They demonstrate ModCon, a model-based testing platform that uses user-defined models to design test oracles, drive test generation, and assess test adequacy. ModCon was able to reach all states and transitions for each scenario in roughly 500 test cases in their experiments.

Purathani Praitheeshan, et al. (Purathani et al., 2019) investigated 16 smart contract vulnerabilities and discovered that some of them need adequate solutions. They projected that numerous attacks are yet to be exploited by associating 16 Ethereum vulnerabilities with 19 software security concerns. They discuss smart contract security issues, as well as the existing analytical tools and detection approaches. They identify and explain the primary weaknesses in smart contracts that could cause serious issues.

# 3 BASIC CONCEPTS

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

**Definition 3.1** (Smart Contract)**.** 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."

**Definition 3.2** (Bounded Model Checking (BMC))**.** "In Bounded Model Checking (BMC), 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. If such a path segment cannot be found at a given length, k, the search is continued for larger k. The procedure is symbolic, i.e., symbolic Boolean variables are utilized; thus, when a check is done for a specific path segment of length k, all path segments of length k are being examined. The Boolean formula that is formed is given to a satisfiability solving program and if a satisfying assignment is found, that assignment is a witness for the path segment of interest" by (Clarke et al., 2004).

**Definition 3.3** (Mutation Verification)**.** "A fault-based verification approach where the mutants are targeted as goal constraints and the verification approach proves the targets with counter-examples as to show the killed mutants."

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

# 4 PROPOSED APPROACH

In this section, we discuss the framework of the proposed approach with an algorithmic description. Also, we provide a detailed explanation with a working example.

## 4.1 Framework of SmartMuVerf

Fig.1 shows a schematic representation of our proposed approach SmartMuVerf. This framework mainly contains four components 1. Mutator 2. *Mutants Annotator*, 3. *SolBMC*, and 4. *Extractor*. The flow starts with supplying *Original Smart Contract* into *Mutator* to create LOR and ROR types mutants. Now, these created mutants along with *Original Smart Contract* supplied into *Mutants Annotator* to produce *Annotated Smart Contract*. This modified version of the smart contract has a goal constraint for each mutant. These goal constraints have been designed and injected in the form of *assertions*.

Next, the *Annotated Smart Contract* is supplied into a smart verifier *SolBMC*. Since *SolBMC* follows 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 killed mutant. Finally, *Killed* and *Alive* mutants reports are generated from *Extractor*.

## 4.2 Algorithmic Description

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

Algorithm 1 shows the implementation of the *SmartMuVerf*. The main input to this algorithm is

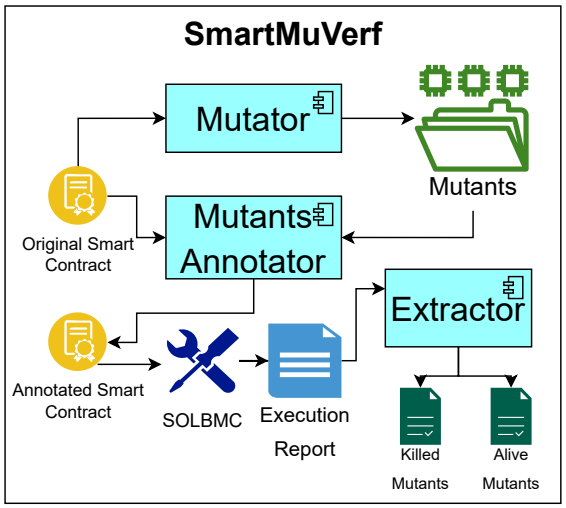


Figure 1: The framework for the proposed approach SmartMuVerf.

*Original Smart Contract (OSC)* to produce *Mutation Score Percentage (MScore%)*. Line 3 of Algorithm 1 invokes Algorithm 2 which shows the implementation of *Mutator* component. Line 4 of Algorithm 1 invokes Algorithm 5 which shows the implementation of *Mutants Annotator* to produce *Annotated Smart Contract (ASC)* as output. Line 5 of Algorithm 1 shows the processing of Solidity Compiler with Bounded Model Checker (SolBMC) by supplying *ASC* and generation *Execution_Report*. Line 6 of Algorithm 1 shows the processing of *Extractor* component. It requires the *Execution_Report* to identify *Killed* and *Alive* mutants. Line 7 of Algorithm 1 shows the formula to compute MScore% using *Killed Mutants* and *Total Mutants*.

Next, let's see the detail of Algorithm 2 named *Mutator* which takes *Original Smart Contract (OSC)* and produces mutants viz. {M1,M2,M3..Mn}. Line 3 of Algorithm 2 invokes Algorithm 3 RM (ROR Mutator). Similarly, Line 4 of Algorithm 2 invokes Algorithm 3 LM (LOR Mutator). Line 5 of Algorithm 2 renames all the collected mutants so that they can be identified uniquely and annotated in a smart contract. Algorithm 3 takes OSC and produces $\{RM1, RM2, RM3..RMn\}$. Line 3 iterates the loop for each condition in the program and extract the relational operator so that mutants can be created. Lines 6 to 23 show that for each relational operator rest 5 operators create the mutants. Finally, Line 25 return all the create ROR mutants {RM1,RM2,RM3..RMn}. Similarly, Algorithm 4 shows the implementation of LM (LOR Mutator). It takes OSC as input and produces LOR mutants {LM1,LM2,LM3,...,LMn}.

Algorithm 5 shows the implementation of the *Mutator_Annotator*. Here we supply Original Smart Contract *OSC* as an input and get Annotated Smart Contract *ASC* as an output. Line 3 of Algorithm 5 shows an iteration for all the predicates identified from OSC. The loop iterates for each predicate identified at a specific line. For each predicate, we iterate all the created mutants specific to that predicate only to create the goal constraints. Our template is **"assert(!((Pred$_i$)!=(M$_j$)));"** for each mutant. We inject this goal constraint just above the Pred$_i$, where $i$ shows the current predate and $j$ shows the current mutant. The aim of this template is to show whether the considered mutant is Killed or Alive after execution. If SolBMC shows the assertion violation happens then the mutant is considered killed otherwise alive.

Algorithm 1: SmartMuVerf.

1: **Input**: {OSC}
2: **Output**: {MScore%}
3: $\{M1, M2, M3..Mn\} \leftarrow$ Mutator(OSC)
4: $ASC \leftarrow$ Mutants_Annotator($OSC, \{M1, M2, M3, ....Mn\}$)
5: $Execution\_Report \leftarrow$ SolBMC(ASC)
6: $\{KilledMutants, AliveMutants\} \leftarrow$Extractor(Execution_Report)
7: MScore%$\leftarrow$ Div(Killed Mutants,Total Mutants) X 100

Algorithm 2: Mutator.

1: **Input**: {OSC}
2: **Output**: {M1,M2,M3..Mn}
3: $\{RM1, RM2, RM3..RMn\} \leftarrow$ RM(OSC)
4: $\{LM1, LM2, LM3..LMn\} \leftarrow$ LM(OSC)
5: $\{M1, M2, M3..Mn\} \leftarrow$Rename({RM1,RM2,RM3,...,RMn}+{LM1,LM2,LM3,...,LMn})

## 4.3 Working Example

In this section, we consider an example smart contract namely *RoomThermostat.sol* from the set of **10** smart contracts. The original smart contract for *RoomThermostat.sol* is shown in Listing 1(Artifacts:, 2022)[5]. The characteristics for *RoomThermostat.sol* wrt. size is **51** LOCs, **3** functions, **3** predicates, and **6** 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. We only consider two classes of faults viz. **LOR** (Logical Operator Replacement) and **ROR** (Relational Operator Replacement). In LOR, we replace || with && and vice-versa. Similarly in ROR, we take one relational oper-

[5]To save space in the paper we have uploaded all the listings (Listings 1 to 8) at (Artifacts:, 2022).

Algorithm 3: RM (ROR Mutator).

```
1: Input: OSC
2: Output: {RM1,RM2,RM3,RM4,...,RMn}
3: while Condition ∈ AllConditions do
4:     OP ← OperatorExtractor(Condition)
5:     Update Mutant ids
6:     if OP == '<' then
7:         RM1 = '! =', RM2 = '>', RM3 = '<=',
   RM4 = '>=', RM5 = '=='
8:     end if
9:     if OP == '>' then
10:        RM1 = '! =', RM2 = '<', RM3 = '<=',
   RM4 = '>=', RM5 = '=='
11:    end if
12:    if OP == '<=' then
13:        RM1 = '! =', RM2 = '<', RM3 = '>',
   RM4 = '>=', RM5 = '=='
14:    end if
15:    if OP == '>=' then
16:        RM1 = '! =', RM2 = '<', RM3 = '<=',
   RM4 = '>', RM5 = '=='
17:    end if
18:    if OP == '==' then
19:        RM1 = '! =', RM2 = '<', RM3 = '>',
   RM4 = '<=', RM5 = '>='
20:    end if
21:    if OP == '! =' then
22:        RM1 = '==', RM2 = '<', RM3 = '>',
   RM4 = '<=', RM5 = '>='
23:    end if
24: end while
25: return {RM1,RM2,RM3,RM4,...,RMn}
```

ator from six operators (>,>=,<,<=,==,!=) and replace it with other five relational operators.

Listing 2(Artifacts:, 2022) shows the total LOR and ROR mutants created for smart contracts in Listing 1(Artifacts:, 2022). There are three predicates and six atomic conditions at lines 26, 36, and 45 of Listing 1(Artifacts:, 2022). The mutants shown at lines 1 to 11 in Listing 2(Artifacts:, 2022) are for line 26 i.e. "if (Installer != msg.sender || State != StateType.Created)" in Listing 1(Artifacts:, 2022). The mutants shown at lines 12 to 22 in Listing 2(Artifacts:, 2022) are for the line 36 i.e. "if (User != msg.sender || State != StateType.InUse)" in Listing 1(Artifacts:, 2022). Similarly, the mutants shown at lines 13 to 23 in Listing 2(Artifacts:, 2022) are for the line 35 i.e. "if (User != msg.sender || State != StateType.InUse)" in Listing 1(Artifacts:, 2022).

Next, using *Mutants Annotator* component of our proposed approach, we generate the Annotated Smart Contract version i.e *RoomThermostat_mod.sol* as shown in Listing 3(Artifacts:, 2022). In this ver-

Algorithm 4: LM (LOR Mutator).

```
1: Input: OSC
2: Output: {LM1,LM2,LM3,LM4,...,LMn}
3: while Predicates ∈ AllPredicates do
4:     OP ← OperatorExtractor(Predicates)
5:     Update Mutant ids
6:     if OP == '&&' then
7:         LM1 = '||'
8:     end if
9:     if OP == '||' then
10:        LM1 = '&&'
11:    end if
12: end while
13: return {LM1,LM2,LM3,LM4,...,LMn}
```

Algorithm 5: Mutants Annotator.

```
1: Input: OSC
2: Output: ASC
3: while Pred_i ∈ OSC do
4:     while M_j ∈ {M1,M2,M3,M4,...,Mn} do
5:         Create and Inject "assert(!((Pred_i)!=(M_j)));" above the Pred_i
6:     end while
7: end while
8: return ASC
```

sion, we inject the targets or goal constraints using the "*assert*" syntax just above the predicates or conditions identified in the contract. We prepare the target for example "*assert(!((Installer != msg.sender || State != StateType.Created) != (Installer !=msg.sender && State != StateType.Created)));*" for the LOR mutant "Installer != msg.sender && State != StateType.Created)" which is from line 26 i.e. "if (Installer != msg.sender || State != StateType.Created)" in Listing 1(Artifacts:, 2022). Note that the assertion function is used to identify whether the **mutant** is **killed** or **alive**. If assertion violation happens for the target asserted then we claim that the mutant is killed, otherwise alive. Deeper to this, please see the "!=" operator between the first expression and the second expression. The first expression in the assert is the original code whereas the second expression in the assert is the mutated code. Now we use traditional definitions of killed mutants and alive mutants. As we know, if the outputs of the original code and mutated code are different then the mutant is considered as killed otherwise alive. Here we compare the context of the state until the point of the reached line where the mutation is applied. We consider that the line can be reached via. one or more different path(s). So our objective in using "!=" is to show the killing of a mutant. Be-

cause, if this non-equality (different outputs or values) is true then the assertion will be violated. Note that only a single path is sufficient to show the killing of a mutant. Now, consider this non-equality which is false for all the paths reaching to that line and never shows the assertion violation. It means there exists no feasible path that a mutant cannot be killed, hence it is an alive mutant. Finally, taking all the mutants from Listing 2(Artifacts:, 2022) and properly annotate as the goal constraints to produce *Annotated Smart Contract* as shown in Listing 3(Artifacts:, 2022).

Now, the *Annotated Smart Contract* as shown in Listing 3(Artifacts:, 2022) is supplied into *SolBMC* to produce the Execution Report with counter-examples Listing 8(Artifacts:, 2022). This report has all the execution logs with the warning and messages. This execution report generated is supplied into the *Extractor* component to get the Killed and alive mutants information. As we have injected **33** targets, *SolBMC* has detected **30** of them with counterexamples. So for this contract, we have a **90.91%** mutation score. This information can be found in the Killed Mutants report in Listing 4(Artifacts:, 2022). The detailed mutation analysis report is shown in Listing 5(Artifacts:, 2022).

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 6(Artifacts:, 2022). We can observe that SolBMC took **1.36** sec for this example. Also, the execution report in Listing 5(Artifacts:, 2022) contains all the counter-examples or test inputs for each killed mutant (assertion violation). These test cases are very much useful.

Also, our technique can be considered as the guided approach to generate the optimal test inputs subject to the types of faults considered. If we had the same set of test cases and we suppose to use the traditional mutation testing then we needed to replay all these unique 33 test inputs over 33 different mutant versions. We can estimate or predict for sure it cannot be finished in 1.59 sec. But, as we can see using *SmartMuVerf* we got the results in 1.59 sec. This enables the usability of our approach.

## 5 EXPERIMENTAL RESULTS

In this section, we discuss the setup and benchmarks tested, and discuss on results. 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[6]. We have used the following command setting as shown in Listing 7(Artifacts:, 2022):

Table 1: Result Analysis Note: #L: Lines of Code, #ML: Modified Lines of Code, #M: Total Mutants, #K: Total Killed Mutants, #A: Total Alive Mutants, MS%: Mutation Score, T: Execution Time (Sec).

| Contracts | #L | #ML | #M | #K | #A | MS% | T |
|---|---|---|---|---|---|---|---|
| BasicProvenance | 53 | 79 | 27 | 24 | 3 | 88.89 | 1.21 |
| RoomThermostat | 51 | 84 | 33 | 30 | 3 | 90.91 | 1.59 |
| SimpleMarketplace | 73 | 101 | 30 | 29 | 1 | 96.67 | 1.16 |
| Token | 43 | 75 | 31 | 31 | 0 | 100.00 | 1.93 |
| escrow | 78 | 99 | 21 | 21 | 0 | 100.00 | 1.53 |
| DogeMojo | 53 | 85 | 31 | 31 | 0 | 100.00 | 2.29 |
| ShibaAstronaut | 53 | 85 | 31 | 31 | 0 | 100.00 | 2.14 |
| UniswapV3MigratorProxy | 23 | 29 | 5 | 5 | 0 | 100.00 | 0.85 |
| payments | 53 | 73 | 20 | 19 | 1 | 95.00 | 1.31 |
| kia_quiz | 51 | 77 | 26 | 23 | 3 | 88.46 | 1.25 |

Table 2: Aggregated Result Analysis.

| | #L | #ML | #M | #K | #A | Avg_MScore(%) | T |
|---|---|---|---|---|---|---|---|
| Total | 531 | 787 | 255 | 244 | 11 | 95.99 | 15.25 |

Table 1 shows the detailed results for **10** smart contracts taken from (etherscan, 2021). #L and #ML present the size of the contract before and after the annotations for targets or goal constraints for created mutants respectively. Here, #L shows the Lines of Code and #ML shows 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. #M shows the total mutants created for the contract. #K shows the Killed mutants. The #A shows the Alive mutants which have been computed using $\#A = \#M - \#K$. MS% shows the mutation score calculated using $MS\% = \frac{\#K}{\#M}$. Lastly, T shows the total execution time in seconds.

Table 2 shows the aggregated results for **10** smart contracts. In total, we processed 531 and 787 Lines of code for original and modified smart contracts respectively. We created a total of 255 mutants and injected them in proper locations and executed them with SolBMC. The verifier proved a total of 244 killed mutants and 11 as Alive mutants. On an average of 10 smart contracts, we achieved 95.99% in 15.25 seconds total time.

## 6 CONCLUSION

The main objective of this work is to verify the created mutants for the smart contract. 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

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

of losing the expensive assets. In this paper, we proposed a novel approach to computing mutation scores for a smart contract using a solidity compiler with a bounded model checker. We propose to use mutation verification in the industry which replaces the traditional mutation testing methodology. Our ongoing work focuses on a detailed analysis of more types of faults. We will explore other techniques such as Fuzzing and Symbolic execution for a more detailed analysis.

## ACKNOWLEDGEMENT

We would like to thank the Department of Science and Technology (DST), Government of India, NM-ICPS, IBITF IIT Bhilai for sponsoring the project to NITMINER Technologies Private Limited (a startup recognised by GOI) under PRAYAS scheme.

## REFERENCES

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.

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15.

Artifacts: (2022). Raw experimnetal data. https://figshare.com/s/9c0f08be4fbc234e8d66.

Böhme, R., Christin, N., Edelman, B., and Moore, T. (2015). Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives*, 29(2):213–38.

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.

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

Driessen, S., Nucci, D. D., Monsieur, G., and van den Heuvel, W.-J. (2021). Automated test-case generation for solidity smart contracts: the agsolt approach and its evaluation.

Erfan, A., Fthiyeh, F., and Mahdi, F. (2020). Testing smart contracts gets smarter. *10th International Conference on Computer and Knowledge Engineering (ICCKE2020)*.

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

Gustavo, G., Will, S., Artur, C., Josselin, F., and Alex, G. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. *ISSTA '20, July 18–22, 2020, Virtual Event, USA*.

Jian-Wei, L., Tsung-Ta, T., and Chia-Kang, H. (2019). Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*.

Jinlei, S., Song, H., Changyou, Z., and Tingyong, W. (2020). Mutation testing for integer overflow in ethereum smart contracts.

Liu, Y., Li, Y., Lin, S.-W., and Yan, Q. (2020a). 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.

Liu, Y., Lin, S., and Qiang, Y. (2020b). Modcon: A model-based testing platform for smart contracts. *ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA*.

Maher, A. and van Moorsel, A. (2019). Blockchain-based smart contracts: A systematic mapping study.

Martin, F. and Boris, P. (2019). Smart contracts.

Morris, D. Z. (2016). Leaderless, blockchain-based venture capital fund raises $100 million, and counting. *Fortune*.

Peng, C., Akca, S., and Rajan, A. (2019). Sif: A framework for solidity code instrumentation and analysis.

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. (2020). Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677. IEEE.

Popper, N. (2016). A venture fund with plenty of virtual capital, but no capitalist. *NYT*.

Purathani, P., Lei, P., and Jiangshan, Y. (2019). Security analysis methods on ethereum smart contract vulnerabilities — a survey.

Reighard, G. L., Loreti, F., et al. (2008). Rootstock development. *The peach: Botany, production and uses. CABI Publishing, Wallingford, Oxon, UK*, pages 193–220.

Sanchez, G., Morales, T., and Torres, V. (2019). Towards an approach for applying early testing to smart contracts. *15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*.

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.

Xingya, W., Haoran, W., Weisong, S., and Yuan, Z. (2019). Towards generating cost-effective test-suite for ethereum smart contract.

Xinming, W., Zhijian, X., Jiahao, H., and Ruihua, N. (2019). Basis path coverage criteria for smart contract application testing. *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*.