# Towards Generating Cost-Effective Test-Suite for Ethereum Smart Contract

Xingya Wang*, Haoran Wu, Weisong Sun, Yuan Zhao
*State Key Laboratory for Novel Software Technology, Nanjing University*
Nanjing, China
xingyawang@nju.edu.cn, {haoranwu, weisongsun}@smail.nju.edu.cn, allenzcrazy@gmail.com

***Abstract*—In Ethereum, many accounts and funds have been managed by smart contracts, thereby making them easy to be targeted. Due to the persistence characteristic of blockchain, revising a deployed smart contract is almost impossible. Both realities heighten the risks of managing funds and thus increase the demand for conducting sufficient testing to Ethereum Smart Contracts (ESC). Different from the conventional software, ESC is a gas-driven program, where developers must charge gases for deploying and testing it. Therefore, it is important to provide a cost-effective yet representative test suite, where its representativeness can be typically measured by its branch coverage. In this paper, we deem the problem of ESC test generation as a Pareto minimization problem, and three objectives, minimizing (1) uncovered branch coverage, (2) time cost, and (3) gas cost are considered. Then, we propose a random based and an NSGA-II based multi-objective approach to seek cost-effective test-suites. Our empirical study on a set of smart contracts in eight of the most widely used Ethereum Decentralized Applications (DApps) verified that the proposed approaches could significantly reduce the gas cost as well as the time cost while retaining the ability to cover branches.**

***Index Terms*—Ethereum smart contract, test-suite generation, Pareto minimization**

## I. Introduction

BitCoin raises the awareness that both cryptocurrency and blockchain have great values such as cost saving and efficiency improvement [1]. Subsequently, more than 2,000 cryptocurrencies were proposed[1]. Among them, Ethereum is one representative cryptocurrency [2]. Smart Contract (SC) can be deemed as an automatable and enforceable agreement. It is firstly proposed by Nick Szabo, who aimed to make the contract that can be automatically and correctly executed without relying on a trusted authority [3]. Blockchain, which possesses the characteristics of decentralization and anonymity [1], provides a feasible technique for building a smart contract supporting platform. Ethereum provides a industry level blockchain within a built-in fully fledged Turing-complete programming language [4], making it easier for practitioner to develop, deploy and apply a smart contract. Currently, more than 1 million smart contracts have been deployed in Ethereum, where 54475 of them had been verified[2].

*Xingya Wang is the correspondence auther, and the work is supported by NSFC (61832009, 61772260, 61673384), Jiangsu Planned Projects for Postdoctoral Research Funds (2018K028C), and Innovation Project for State Key Laboratory for Novel Software Technology (ZZKT2018B02)

[1]Data from CoinMarketCap: 2086 cryptocurrencies on Jan. 6th 2019.

[2]Data from Etherscan: 54475 smart contracts on Jan. 6th 2019.

Though smart contract has attracted an increasing number of practitioners, developing a high-quality SC is still full of challenges. On the one hand, SC is often used in safety-critical areas such as finance and credit, thereby making it an attractive target for attackers. On the other hand, another characteristic of blockchain, persistence [1], increases the difficulty of bug fixing in SC. That is, once an SC has been deployed, there is little chance to revise it. Thus, a minor bug might result in a series of devastating losses, and fixing it might be quite costly. For example, the infamous DAO attack, which causes a loss of 60 million US dollars [5], is caused by a reentrancy bug. To deny the hacker, Ethereum has to conduct a hard fork, making Ethereum no longer be seen as immutable. Moreover, compared to conventional software, environments for developing, testing as well as debugging SC are immature, and SC developers are young and inexperienced, both of which make it harder to develop a high-quality SC. Since software quality has become one dominant success criterion in the software industry [6], and testing remains as the most commonly used technique to ensure the quality of software [6], prior to deployment, SC must be given sufficient testing.

Generally, sufficient testing indicates that a high-quality test-suite must be designed and generated for the SC under test. Subsequently, to make the testing practicable, one important question must be answered, that is, how to measure the quality of an SC test-suite? For conventional software, branch coverage is commonly mandated in industrial testing standards [7] in the industry to evaluate the quality of the generated test cases. The higher the branch coverage, the more the blocks of the program under test are examined. Thus, minimizing the uncovered branches becomes one important objective in test-suite generation. One natural way to decrease uncovered branches is augmenting the existed test-suite by generating additional test inputs for the uncovered branches. However, generating more test inputs indicates that the testers have to spend more time to execute the program under test. Moreover, an Ethereum Smart Contract (ESC), when it is called, will run in each of the nodes in Ethereum. Long-term running ESC inevitably delays the subsequent transactions and degrades the Ethereum performance. In this regard, Ethereum employs a Gas mechanism to limit the executing time and memory consumption [2]. For a deployed ESC, its execution is driven by gases, which must be purchased in ethers. Undoubtedly, conducting sufficient testing for a deployed ESC will cost

plenty of gases and ethers. Therefore, the generated test-suite also should be cost-effective, that is, costing time and gases as little as possible.

As stated, a high-quality ESC test-suite must satisfy the following three objectives, that are: (1) minimizing uncovered branches, (2) minimizing time cost, and (3) minimizing gas cost. Obviously, objectives (1) and (2) as well as (1) and (3) conflict, which mean that finding a solution that can achieve the best result in each of the objectives is unrealistic. Thus, ESC test generation can be deemed as a multi-objective optimization problem, which aims to gain a tradeoff among multiple objectives by generating the Pareto solutions [8]. In this paper, we propose the first multi-objective test generation approach for testing Ethereum smart contract. Specifically, we resort two algorithms, Random Based Multi-objective test generation algorithm (RBM) and NSGA-II Based Multi-objective test generation algorithm (NBM), to seek the cost-effective test-suites. NSGA-II is a representative multi-objective genetic algorithm (with over 26,000 citations at the time of writing[3]) [9]. Representation of the solution as well as the evolution operators (i.e., selection, recombination and mutation) are redesigned for applying NSGA-II into the ESC test-suite generation. To verify the effectiveness of both RBM and NBM, we conducted an empirical study, where two independent variables, objectives and generation algorithm, are considered respectively. This paper makes the following contributions.

1) **Approach**. We introduced the first Pareto minimization approach to ESC testing, combining one objective (i.e., minimizing uncovered branches) used for conventional software and two cost-effective objectives (i.e., minimizing both time cost and gas cost).
2) **Empirical Study**. We conducted an empirical study on a set of smart contracts. Results verified that our approaches could significantly reduce the gas and time cost while retaining the ability to cover branches.

The rest of this study is organized as follows. Section II describes the basics of Ethereum smart contract and multi-objective optimization. Our test-suite generation approaches are described in detail in Section III. Experimental design and results analysis are presented in Section IV. Finally, Section V concludes the article and outlines directions for future research.

## II. Background

### A. Ethereum Smart Contract

The development language (e.g. Solidity[4]) in Ethereum is Turing completeness. Thus, it has conditional branching and the ability to change an arbitrary amount of memory, each of which may block the subsequent transactions, and thus affects Ethereum performance. To restrict them, Ethereum employs a Gas mechanism, where two concepts, $gas_{limit}$ and $gas_{price}$, are defined. Then, you should purchase more ethers if the smart contract you called executes too long, or consume too much memory.

**Definition II.1** ($gas_{limit}$)**.** A scalar value equal to the maximum amount of gas that could be used in executing one transaction. This is paid up-front, before any computation is done and may not be increased later [4].

**Definition II.2** ($gas_{price}$)**.** A scalar value equal to the number of Wei to be paid per unit of gas for all computation costs incurred as a result of the execution of this transaction [4].

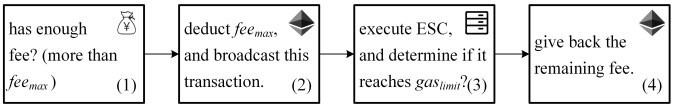


Fig. 1. Workflow of Ethereum Gas Mechanism

Fig. 1 depicted the process of executing an ESC under the Ethereum Gas mechanism:

1) Given $gas_{limit}$ and $gas_{price}$, Ethereum checks if the sender account $sa$ has enough ethers. It continues if $sa$ has (i.e., $fee_{sa} \geq fee_{max} = gas_{limit} \times gas_{price}$), otherwise it stops and throws an out-of-gas exception.
2) Ethereum deducts $fee_{max}$ from $sa$, and broadcast the request of executing ESC to all accounts.
3) Once an account receives the request, its EVM starts to execute ESC and monitor the gases used $gas_{used}$. It continues if $gas_{used}$ consistently lower than $fee_{max}$, otherwise it stops and throws an out-of-gas exception.
4) Once the transactions have been recorded in the blockchain, Ethereum gives back the remaining fee (i.e. $fee_{max} - gas_{used} \times gas_{price}$) to $sa$. Note that no fee will be given back if $fee_{max}$ has been run out.

### B. Multi-objective Optimization

Optimization is a common problem in engineering practice [10]. Problem with only one objective is called Single-objective Optimization Problem (SOP), otherwise, it is called Multi-objective Optimization Problem (MOP) [9]. SOP aims to find a feasible solution that archives the best result (i.e. minimum or maximum) in the given objective function. MOP is quite different from SOP. Objectives may conflict, that is, optimizing one objective needs to sacrifice the other objectives. Thus, MOP aims to gain a tradeoff among the different objectives rather than finding a solution that archives best result in all objective functions. Without loss of generality, in this paper, we resort minimization as the goal for all the objectives. Then, a general MOP can be formally defined as follows:

**Definition II.3** (MOP)**.** The set of all the values satisfying $p$ inequality constraints $g_i(x) \geq 0,\ i \in \{1, 2, \ldots, p\}$ and $q$ equality constraints $h_i(x) = 0,\ i \in \{1, 2, \ldots, q\}$ defines the feasible region $\Omega$ and any point $x \in \Omega$ is a feasible solution. MOP aims to find a solution $x = (x_1, x_2, \ldots, x_n)$ that minimizes the vector function $f(x) = (f_1(x), f_2(x), \ldots, f_m(x))^T$, where $x_1, x_2, \ldots, x_n$ are decision variables.

[3]Data from Google Scholar: 26513 citations on Jan. 6th 2019.
[4]Solidity. https://solidity.readthedocs.io.

**Definition II.4** (Domination)**.** Taking into account the definition of MOP, a solution $x^1 = (x_1^1, x_2^1, \ldots, x_n^1)$ is said to dominate a solution $x^2 = (x_1^2, x_2^2, \ldots, x_n^2)$ denoted with $x^1 \prec x^2$, if and only if $f_i(x^1) \leq f_i(x^2)$ for $i \in \{1, 2, \ldots, m\}$, and there exists at least one $j \in \{1, 2, \ldots, m\}$ such that $f_j(x^1) < f_j(x^2)$.

**Definition II.5** (Pareto Optimal Set)**.** The solution of a given MOP is usually a set of solutions satisfying:

- Every pair of two solutions in the set is non-dominated.
- Any other solution is dominated by at least one solution in the set.

Finding a solution that works best in all but conflicting objectives is virtually impossible. Thus, MOP requires a set of solutions known as a Pareto optimal set [9]. Such a set contains only non-dominating solutions. As stated in Definition II.4, individual $x^1$ dominates $x^2$ if, and only if, $x^1$ is better than $x^2$ in at least one objective, and no worse in all other objectives. Conversely, two points are said to be non-dominated whenever none of them dominates the other.

Test-suite generation aims to automatically generate a high-quality test-suite for alleviating the burden of practitioners from the laborious task of test designation. Thus, we require a precise definition of high-quality to guide the process of test-suite generation and evaluate the generated results. For a generated result, branch uncoverage $uncov_{branch}$ and time cost $cost_{time}$ are two commonly used measurements [11], in which $uncov_{branch}$ measures its adequacy and $cost_{time}$ measures its efficiency. As stated, Ethereum resorts to a Gas mechanism to solve the problems (e.g., long-term execution and high memory consumption) resulted by Turing completeness. With this mechanism, testers must purchase more if the generated result costs too much gases. Thus, the gas cost $cost_{gas}$ should also be deemed as an important measurement. Therefore, a high-quality ESC test-suite should satisfy low $uncov_{branch}$, low $cost_{time}$ and low $cost_{gas}$. Then, the problem of ESC test-suite generation can be deemed as a MOP problem, and the vector function corresponds to $f(ts) = (f_1(ts) = uncov_{branch}(ts), f_2(ts) = cost_{time}(ts), f_3(ts) = cost_{gas}(ts))^T$ with $ts$ refers to a generated test-suite.

## III. Our Approaches

This section presents our proposed multi-objective test generation approaches. Specifically, we propose a Random Based Multi-objective test generation approach (RBM) and an NSGA-II Based multi-objective test generation approach (NBM), in which a typical evolutionary multi-objective optimization algorithm, NSGA-II [9], is selected. In our case, a solution is a test-suite, which is represented as a set $T$ of test cases. Given $|T| = n$, we have $T = \{t_1, t_2, ...t_n\}$. A test case $t$ is a list of values w.r.t the arguments of a method in the smart contract under test.

### A. Random Based Multi-objective Test Generation

Given a smart contract $sc$, RBM requires: (1) its source code; (2) its Application Binary Interface (ABI) [2]; (3) the time budget $t_b$ for a test-suite creation; and (4) the number of solutions $num_{solution}$.

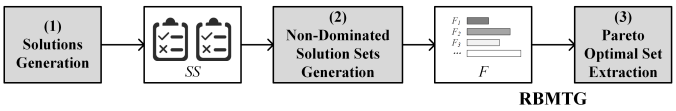


Fig. 2. Workflow of random based multi-objective test generation

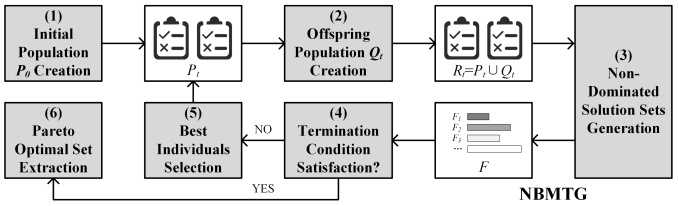


Fig. 3. Workflow of NSGA-II based multi-objective test generation

As Fig. 2 depicted, RBM randomly generates $num_{solution}$ solutions, sorts them based on nondomination [8], and selects the Pareto optimal set as the output [8]. Each solution is given a time budget $t_b$ to be generated. A solution is a set of randomly generated test cases, where each of which corresponds to one method declaration in ABI. Once all solutions have been generated, we estimate their branch coverage, time costs and gas costs, in which all of these are collected by tracing the test cases on the deployed $sc$.

### B. NSGA-II Based Multi-objective Test Generation

NBM is depicted in Fig. 3: Starting with a random population $P_0 = \langle T_0^1, T_0^2, ..., T_0^m \rangle$, evolution is performed until the termination condition (i.e., number of iterations) is satisfied. In each iteration, an offspring population $Q_t$ is created by selecting and evolving (i.e., crossover and mutation in order) the best $m$ individuals in the last generation $P_t$. Both $P_t$ and $Q_t$ are added to the new generation $R_t$. Then, solutions in $R_t$ are ranked based on non-dominated sorting, depending on function $f(T)$. NBM outputs the Pareto optimal set when the maximum iteration has been reached. Fig. 3 depicts the high-level NBM workflow. To adapt it to our test generation problem, we need to define the crossover and mutation operators.

*1) Crossover:* As stated, a solution can be deemed as a list of test cases and thus our crossover works at test-suite level. The adopted selection operator is the conventional binary tournament [8]–[10], and each pair of the selected test-suites has a $\frac{1}{2}$ probability to be crossed. We follow the work of whole test suite generation [12] to design crossover operator. Given two parent solutions, $P_1$ and $P_2$, we first randomly choose a value $\alpha$ from $[0, 1]$ and split $P_1$ into two test-suites, $P_1^\alpha$ and $P_1^{1-\alpha}$, in which they contain the first $\alpha|P_1|$ and the latter $(1-\alpha)|P_1|$ test cases, respectively. Similarly, $P_2$ is split into $P_2^\alpha$ and $P_2^{1-\alpha}$. Then, the first offspring $O_1$ is generated by combining $P_1^\alpha$ and $P_2^{1-\alpha}$, and the second offspring $O_2$ is generated by combining $P_2^\alpha$ and $P_1^{1-\alpha}$.

*2) Mutation:* Different with crossover, our mutation operator works at both test-suite and test case levels. In each iteration, every test-suite has a $\frac{1}{2}$ probability to be mutated.

Then, the test-suite under mutation will be applied the three types of operations, $remove$, $change$, and $insert$, in order. For a test-suite $T$, these three operations work as follows: (1) $remove$ corresponds to randomly delete a test case in $T$. (2) $change$ corresponds to randomly select a test case $t = \langle arg_1, arg_2, ..., arg_l \rangle$ in $T$ and conduct a test case level based mutation for it. Each argument in $t$ is changed with probability $\frac{1}{l}$ and the argument under changed will be randomly reassigned a new value based on its variable type. (3) $insert$ corresponds to randomly generate a test case and insert it at the end of $T$. Each operation is applied with probability $\frac{1}{3}$. Therefore, on average, only one of them is applied.

## IV. Evaluation

We investigated the following two main research questions:

1) Can RBM and NBM generate cost-effective Ethereum test suites?
2) Are RBM and NBM retaining the ability of branch coverage in generating Ethereum test suites?

To answer these research questions, we considered the Completely Random Based approach (CRB) and two single-objective based approaches (RBS and NBS) to compare our proposal with. CRB corresponds to the completely random based test generation, which does not consider any objectives during the test generating process. RBS and NBS take low $uncov_{branch}$ as the optimization objective, and they correspond to random based and NSGA-II based approach, respectively. We selected these baselines because they can reflect if we make a delicate tradeoff among the objectives.

### A. Experimental Subjects and Measures

TABLE I
Subject Programs

| DAPP Name | LOC | BOC | DAPP Name | LOC | BOC |
|---|---|---|---|---|---|
| NEVERDIE | 26 | 6 | 0xBitcoin | 108 | 46 |
| Bancor | 27 | 20 | FuseaNetwork | 120 | 52 |
| UranBank | 60 | 32 | LEE | 137 | 76 |
| OpenSea | 88 | 40 | Etherchicks | 199 | 80 |

We used the smart contracts in eight of the mostly used Ethereum Decentralized Applications (DApps)[5] as experimental subject programs. Table I summarizes the characteristics of these smart contracts. For each DAPP, its name, the number of code lines (excluding comments and blank lines) and the number of branches are described.

In empirical study, we considered the following quantitative criteria to assess the results of the proposed approaches to compare that were obtained by applying three baselines.

$$\begin{cases} uncov_{branch}(T) = (1 - \frac{|\bigcup branch_{covered}(t)|}{|branch_{subject}|}) \times 100\%, t \in T \\ cost_{time}(T) = \sum cost_{time}(t), t \in T \\ cost_{gas}(T) = \sum cost_{gas}(t), t \in T \end{cases} \quad (1)$$

[5]Data from State of the DApps: 1842 Ethereum DAPPs on Jan. 6th 2019.

Equation (1) presents the evaluation metrics in quantitative criteria. Given a test-suite $T$, $uncov_{branch}(T)$ represents the percentage of branches of the $sc$ under test that has not been exercised by the test cases of $T$. $uncov_{branch}$ values range in between 0 and 100. The lower the $uncov_{branch}$ value, the higher the branch coverage is, and the better it is. $cost_{time}(T)$ and $cost_{gas}(T)$ represent the time as well as the gas required for executing the test cases in $T$ on the deployed $sc$, respectively. For each of the latter two measures, the lower the value, the better it is.

### B. Procedure

For each $sc$ under test, we applied the following procedure:

1) *Collecting artifacts.* We collected the following artifacts: source code and its ABI.
2) *Compiling and deploying.* To run $sc$, we compiled it into bytecode, and deployed it on Ethereum testnet.
3) *Running RBM, NBM and baseline approaches, and collecting data.* We ran the random based approaches with the following setup: time budget $tb$=100ms, number of solutions $num_{solution}$=1000, and we ran the NSGA-II based approaches with the following setup: time budget $tb$=100ms, population size $size_{pop}$=20, number of iterations $num_{iteration}$=50. Since all approaches have a non-deterministic behavior, we ran them 10 times and collected all the generated solutions. For each solution, we collected its uncovered branches, time cost and gas cost.

### C. Results and Analysis

We conducted the experiment following the procedure in Section IV-B and present the results of the baselines (i.e., CRB, RBS, and NBS) as well as the proposed approaches (i.e., RBM, NBM) in Table II. Columns 2-6 illustrate the branches that had not been covered per subject program when different approaches are employed. Similarly, the latter two set of columns, 7-11 and 12-16, represent the time cost as well as the gas cost when different approaches are employed, respectively. The last row of table II illustrates the average results on all subjects of each approach.

From table II, we can observe that, the NSGA-II based approaches always achieve better branch covered results than that of the random based approaches, where the completely random approach CRB always performs worst. The average scores of both RBM and NBM are slightly higher than that of the single-objective based approaches, RBS and NBS, by 0.90% and 0.89%, respectively, indicating that our approaches maintain the ability of covering branches. Besides, we can also observe that, the executing time, as well as the used gasses employing our approaches are always lesser than that of the three baseline approaches. For example, the average gas cost of NBM is 1.47 million, which is much lesser than the gas cost when using the test-suites generated by the baselines. It means that in the empirical study, our approaches can generate more cost-effective test-suites.

TABLE II
EXPERIMENTAL RESULTS OF THE PROPOSED APPROACHES AND THE BASELINES

| SC Name | Uncovered Branches (%) | | | | | Time Cost (s) | | | | | Gas Cost (million) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CRB | RBS | NBS | RBM | NBM | CRB | RBS | NBS | RBM | NBM | CRB | RBS | NBS | RBM | NBM |
| NEVERDIE | 36.73 | 16.67 | 16.67 | 16.67 | 16.67 | 8.29 | 10.60 | 11.59 | 8.92 | 4.36 | 2.25 | 2.55 | 3.53 | 1.90 | 0.67 |
| Bancor | 60.41 | 60.00 | 60.00 | 60.00 | 60.00 | 11.35 | 14.09 | 16.90 | 11.24 | 6.83 | 2.22 | 2.05 | 2.88 | 1.14 | 0.51 |
| UranBank | 37.37 | 21.87 | 15.62 | 21.87 | 15.62 | 10.37 | 12.77 | 14.15 | 10.86 | 7.88 | 2.67 | 2.63 | 4.84 | 2.52 | 1.83 |
| OpenSea | 57.56 | 55.00 | 55.00 | 55.00 | 55.00 | 15.85 | 20.52 | 21.66 | 12.79 | 9.21 | 4.74 | 5.01 | 5.64 | 4.17 | 2.01 |
| 0xBitcoin | 74.57 | 73.91 | 73.91 | 73.91 | 73.91 | 9.85 | 10.55 | 10.74 | 9.33 | 5.99 | 1.91 | 1.82 | 1.77 | 1.66 | 0.99 |
| FuseaNetwork | 56.73 | 51.92 | 48.08 | 53.85 | 50.00 | 7.55 | 7.84 | 10.09 | 6.77 | 4.46 | 1.56 | 1.77 | 3.64 | 1.41 | 1.12 |
| LEE | 68.42 | 65.78 | 60.53 | 66.11 | 62.04 | 7.05 | 9.33 | 9.83 | 6.91 | 3.91 | 1.90 | 3.80 | 4.02 | 2.08 | 0.75 |
| Etherchicks | 62.75 | 52.50 | 33.75 | 57.50 | 37.50 | 35.16 | 39.43 | 53.23 | 34.56 | 19.38 | 9.28 | 10.47 | 15.29 | 6.10 | 3.92 |
| AVG | 56.82 | 49.71 | 45.45 | 50.61 | 46.34 | 13.18 | 15.64 | 18.52 | 12.67 | 7.75 | 3.32 | 3.76 | 5.20 | 2.62 | 1.47 |

TABLE III
STATISTICAL ANALYSIS RESULTS

| Approaches | RBM (p-value) | | | NBM (p-value) | | |
|---|---|---|---|---|---|---|
| | $f_1$ | $f_2$ | $f_3$ | $f_1$ | $f_2$ | $f_3$ |
| CRB | 0.049 | 0.247 | 0.108 | 0.023 | 0.010 | 0.013 |
| RBS | 0.193 | 0.008 | 0.055 | 0.112 | 0.005 | 0.013 |
| NBS | 0.112 | 0.022 | 0.033 | 0.111 | 0.016 | 0.013 |
| RBM | - | - | - | 0.119 | 0.013 | 0.003 |

To further evaluate the performance of RBM and NBM, we applied the paired Wilcoxon tests to the baselines. In order to show that RBM and NBM are more cost-effective, we carried out the two-tailed alternative hypothesis to verify that RBM (or NBM) costs the least executing time (or gasses) that the compared baselines, respectively. From table III, we can obtain the following observations: (1) the p-values of all tests between RBM and single-objective based approaches range from 0.008 to 0.055, and the p-values of all tests between RBM and CRB range from 0.108 to 0.247. Therefore, we can accept the hypothesis with confidence level 0.945 of the test between RBM and single-objective based approaches, but reject the hypothesis of the test between RBM and CRB. (2) the p-values of all tests between NBM and other approaches range from 0.003 to 0.016. Therefore, we can accept the hypothesis of the test between NBM and all the other approaches. In summary, both approaches can provide more cost-effective test-suites even though RBM is not very significant when compared to the completely random approach.

Moreover, we applied the paired Wilcoxon tests to verified the effectiveness of covering branch. The p-values of all tests between our approaches and CRB range from 0.023 to 0.049, and the p-values of all tests between our approaches and single-objective based approaches range from 0.111 to 0.193. Results indicate that both proposed approaches significantly outperform the completely random approach, and can maintain the branch covered ability of taking only minimizing $uncov_{branch}$ as the test generation objective.

Based on these above results, for this set of smart contracts, in our empirical study, we can answer the two proposed research questions as follows: our multi-objective based approaches can (1) significant reduce the time cost and gas cost, and (2) retain the ability of branch covering.

## V. CONCLUSION

Due to the fact that ensuring the quality of smart contract is important, researchers have proposed a set of testing and verifying techniques [13]–[15]. However, in these studies, testing cost had not been dedicated discussed. Our work provides a well complementation to them. In this paper, we proposed and studied RBM and NBM, two multi-objective test generation approaches for Ethereum smart contract. Both approaches aim to generate cost-effective test-suites while do not sacrifice the ability of branch covering. Our empirical study on a set of open-source, widely used smart contracts verify their effectiveness. Conducting more experiments as well as employing more optimization objectives such as maximizing mutation killed ability during the test-suite generation are our next work.

## REFERENCES


[1] Z. Zheng et al., "Blockchain challenges and opportunities: a survey," *IJWGS*, vol. 14, no. 4, pp. 352–375, 2018.
[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, pp. 1–32, 2014.
[3] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.
[4] V. Buterin, "A next-generation smart contract and decentralized application platform," *Ethereum White Paper*, 2014.
[5] WIRED, "A $50 million hack just showed that the dao was all too human," 2016.
[6] A. Orso et al., "Software testing: a research travelogue (2000-2014)," in *Proc. FOSE*, pp. 117–132, 2014.
[7] T. T. Chekam et al., "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. ICSE*, pp. 597–608, 2017.
[8] N. Srinivas et al., "Multiobjective function optimization using nondominated sorting genetic algorithms," *IEEE TEC*, vol. 2, no. 3, pp. 221–248, 1995.
[9] K. Deb et al., "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE TEC*, vol. 6, no. 2, pp. 182–197, 2002.
[10] R. T. Marler et al., "Survey of multi-objective optimization methods for engineering," *SMO*, vol. 26, no. 6, pp. 369–395, 2004.
[11] J. Ferrer et al., "Evolutionary algorithms for the multi-objective test data generation problem," *SPE*, vol. 42, no. 11, pp. 1331–1362, 2012.
[12] G. Fraser et al., "Whole test suite generation," *IEEE TSE*, vol. 39, no. 2, pp. 276–291, 2013.
[13] K. Bhargavan et al., "Formal verification of smart contracts: Short paper," in *Proc. PLAS*, pp. 91–96, 2016.
[14] B. Jiang et al., "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. ASE*, pp. 259–269, 2018.
[15] R. M. Parizi et al., "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proc. CASCON*, pp. 103–113, 2018.