Automatic Generation of Solidity Test for Blockchain Smart Contract using Many Objective Search and Dimensionality Reduction

Dongcheng Li[1], W. Eric. Wong[1,*], Sean Pan[2], Liang-Seng Koh[2], and Nicholas Chau[1]
[1]Department of Computer Science, University of Texas at Dallas, Richardson, Texas, USA
[2]RFCyber Corporation, Frisco, Texas, USA
*corresponding author

***Abstract*—**Blockchain and smart contract technologies have emerged as innovative approaches to trustworthy and reliable service computing. With the growing popularity of decentralized applications, the quantity of smart contracts has seen an exponential increase. Research on testing smart contracts has primarily focused on identifying specific vulnerabilities in smart contracts and blockchains. However, generating a robust test suite for smart contracts remains a daunting challenge. The state-of-the-art DynaMOSA algorithm uses many-objective optimization for test case generation, introducing preference sorting and dynamic target selection strategies. Yet, under hyper many-objective conditions, the algorithm faces difficulties including insufficient selection pressure and reduced efficiency. This paper proposes a solution to these issues by introducing a many-objective optimization algorithm with a dimensionality reduction strategy. The goal is to condense an extensive dataset into a smaller, more manageable and effective dataset using specific reduction criteria. Additionally, this paper utilizes 28 open-source Solidity projects from GitHub for testing. The experimental results show that compared to DynaMOSA, the proposed approach achieves higher testing coverage on most projects under test and shows a noticeable improvement in algorithm efficiency for the automatic generation of Solidity Tests.

Keywords-automatic test case generation; many-objective optimization; DynaMOSA; principal component analysis; laplacian eigenmap

1. INTRODUCTION

Bitcoin whitepaper [1] signaled the inception of blockchain, a decentralized distributed ledger enabling peer-to-peer transactions without mutual trust. Ethereum, a leading public blockchain platform, supports smart contracts [2] - specialized programs that use the blockchain's consensus mechanism to autonomously reach agreements. These contracts, equivalent to written contracts with terms coded, are reliably executed by the Ethereum Virtual Machine (EVM) and their immutability ensures determinism.

Ethereum smart contracts, primarily written in Solidity, are sets of code and data hosted on the Ethereum blockchain. They can be triggered by either other smart contracts or users. Each account has persistent storage and an Ether balance adjusted by transactions [3]. Although Solidity smart contracts seem similar to JavaScript or C programs in syntax, their unique semantics can lead to unconventional reliability and security issues. Developers often need to employ creative strategies to express desired functions, which can introduce vulnerabilities and bugs arising from discrepancies between Solidity's semantics and the programmer's intentions [4].

Due to the popularity of Decentralized Applications (DApps), the number of smart contracts has dramatically increased. These contracts, involving significant digital assets and immutable post-deployment, encounter more stringent reliability and security demands than conventional software [5]. Since 2016, various security incidents have uncovered diverse vulnerabilities in smart contracts, resulting in considerable financial losses. Notable incidents include the Decentralized Autonomous Organization (DAO)'s hacking in 2016, Ethereum's Parity wallet bug in 2017, and the BeautyChain (BEC) attack in 2018 [6]. Given these severe repercussions, ensuring the reliability and dependability of smart contracts has become a pressing concern.

Luu et al. [7] and Atzei et al. [8] have identified design flaws in published smart contracts. Zou et al. [9] further validated that many developers desire tools to ensure smart contract code correctness. This highlights the urgency of preventing smart contract bugs and vulnerabilities through robust testing before deployment. To date, research on testing smart contracts is primarily focused on identifying known vulnerabilities [10] and test case creation is often done manually. However, this approach is labor-intensive, time-consuming, and heavily relies on the skills and experience of smart contract developers, considering the growing quantity and complexity of smart contracts. Therefore, the automatic generation of efficient and reliable test cases to detect bugs and vulnerabilities in smart contracts is vital [11]. The lack of effective automatic test suites generator is a significant obstacle to transitioning technologies from academia to industry, making the automatic generation of test suites a critical research area at present [12].

This paper reframes the problem of automatic test case generation for smart contracts as a many-objective optimization problem. We decompose a smart contract into multiple branches, treat each as an optimization goal, and by optimizing program branch coverage, we search for optimal test cases. To overcome the shortcomings of existing frameworks, we propose a many-objective test case generation algorithm using dimensionality reduction strategies, namely, principal component analysis (PCA) and Laplacian Eigenmap (LE). This approach reduces numerous test case generation objectives to a few essential ones, thereby generating test cases with high coverage and

2767-6684/23/$31.00 ©2023 IEEE
DOI 10.1109/DSA59317.2023.00057

improving test case generation efficiency. The approach's effectiveness is measured by the branch coverage and execution time of the generated tests, and its superiority and feasibility are validated with 28 open-source Solidity smart contract projects collected from GitHub.

The remainder of the paper is organized as follows. Section 2 presents related studies; Section 3 introduces the smart contract test case generation problem as a many-objective optimization problem; Section 4 describes the proposed many-objective search algorithm with dimensionality reduction for the automatic generation of Solidity tests of Smart Contracts; and Section 5 describes the experimental setup and analyzes the results. Finally, Section 6 presents the conclusions.

2. RELATED STUDIES

From the previous century onwards, a multitude of research has been conducted on automatic test case generation by scholars around the world. Numerous methodologies for the automated generation of test cases have been proposed [13]. Notably, the automatic test case generation technique rooted in heuristic search algorithms has achieved substantial progress. In essence, the challenge of test case generation presents itself as a multi-objective problem. The process of program data flow execution can be visualized through a control flow graph, with each statement block representing an objective. The aim is to generate test cases that cover these objectives, with the evaluation of fitness calculations being determined based on various testing coverage criteria, for example, line, branch, and mutation coverage. Inspired by biological evolution, genetic algorithms operate with a set of (candidate) solutions or chromosomes. They employ iterative applications of evaluation, selection, crossover, and mutation to yield the subsequent generation of optimal solutions [14].

DynaMOSA (Many-objective Sorting Algorithm with Dynamic target selection) [15] is a state-of-the-art algorithm purpose-built for automated test case generation. Its methodology employs the principles of NSGA-II-inspired multi-objective optimization, facilitating the production of compact, efficient, and high-coverage test suites [16]. Empirical evidence illustrates DynaMOSA's significant superiority over alternative test case generation algorithms, especially concerning branch and mutation coverage across a diverse range of Java class projects [17], [18]. In EvoSuite [19], basic static analysis is initially used to extract information regarding classes and their constructors, methods, and fields. Bytecode is then inserted when the Java class loader loads the classes. Ultimately, the meta-heuristic search algorithm (DynaMOSA) [20] is employed to automatically generate JUnit test cases aimed at maximizing code coverage.

Fraser et al. [17] introduced the concept of Whole Suite (WS) optimization within the EvoSuite framework, marking the first application of many-objective optimization to test case generation. This approach strives to cover all objectives simultaneously with the entire test suite while minimizing its total size. Fraser's work demonstrated superior results compared to those targeting single objectives. Sahin et al. [21] developed an archive-based, multi-criterion Artificial Bee Colony (ABC) algorithm for test suite generation. This algorithm maximizes the fitness functions of various objectives for object-oriented software. The Archive-Based Artificial Bee Colony algorithm (ABC) retains covered objectives in an archive to effectively utilize available search resources. The feasibility and effectiveness of the algorithm were verified through experiments. The introduction of an archive into the ABC algorithm leads to faster convergence speed than the basic ABC algorithm and achieves a higher coverage rate.

Panichella et al. [20] introduced an enhanced multi-objective genetic algorithm, known as Many-objective Sorting Algorithm (MOSA), building upon the test suite concept. The fundamental premise involved an initial preference sorting to select optimal individuals, followed by a fast non-dominated sort for the remaining candidates. These individuals then underwent genetic operations like selection, crossover, and mutation. Nevertheless, in automatic test case generation, the principal challenge was the decrease in efficiency brought about by the overwhelming number of objectives. To address this, Panichella et al. proposed the dynamic target selection strategy known as DynaMOSA [15] to bolster the algorithm's search performance. The key idea centered on examining coverage targets that held a dominant position via a control dependency graph. The approach entailed dynamic target selection to reduce the number of objectives per algorithm search, consequently improving test case search performance.

To achieve higher test case coverage, Panichella et al. [15] proposed a multi-criterion coverage strategy based on the DynaMOSA algorithm. This primarily involved handling objectives such as branch, line, and mutation coverage simultaneously. Following this, to integrate non-functional metrics into test case generation, Panichella and his team presented an adaptive approach, termed ADynaMOSA (Adaptive Many-Objective Sorting Algorithm with Dynamic Target Selection) [22]. This approach factored the test case execution time and memory usage into the algorithm's optimization objectives. The result was a maintained performance while generating test cases with high coverage. However, for many-objective multi-criteria test case generation problems, there exist dozens or even hundreds of covering objectives in each class under test, which poses a significant challenge to the algorithm’s search performance and scalability.

Aiming at alleviating the above-mentioned issue, Li et al. [23] proposed PCA-DynaMOSA algorithm for Java program, addressing its inefficiencies in scenarios with a large number of objectives. It is utilizing the dimensionality reduction technique for a set of optimization objectives or fitness matrixes, transforming original objective space data into a new smaller space, where dimensions represent objectives. It can generate high-coverage test cases within a constrained timeframe. It was tested on 49 projects from the SF110 benchmarking dataset, demonstrating superiority over DynaMOSA in line, branch, mutation, multi-criteria coverage, and search efficiency.

Although search-based test case generation methods can effectively produce relatively comprehensive test suites, they

still fall short when it comes to generating test cases for inputs with complex structures. Dynamic Symbolic Execution (DSE) [24], [25] can achieve high code coverage by executing programs concretely, collecting symbolic conditions, and solving constraint systems. To mitigate the c challenges, it is worth considering a fusion of symbolic execution and heuristic search [26]. Galeotti et al. [27] expanded the genetic algorithm (GA) in the EvoSuite unit test generator by integrating DSE into an adaptive approach. In this system, feedback from the search process determines when DSE was the appropriate solution to use for a search. Experimental findings indicated that this hybrid approach provided improvements over its individual constituent techniques alone, GA and DSE.

Li et al. [28] proposed an innovative local search algorithm that unites adaptive simulated annealing and symbolic path constraints to optimize the neighborhood search of ideal solutions. This approach was designed to address limitations associated with both global search methods and local search strategies using the alternating variable method. The adaptive simulated annealing algorithm was introduced to effectively explore parameters of each statement and avoid local optimal solutions. The symbolic path constraints were utilized to navigate constraints encountered during test case execution, ensuring high code coverage. The research team demonstrated an effective balance between resource consumption for global and local search. They compared their algorithm with leading test case generation algorithm, using the SF110 open-source benchmarking datasets to demonstrate its efficacy.

Furthermore, some researchers have combined machine learning and search-based methods to enhance the efficiency of test case generation. Modonato [29] proposed Tardis, a unit test generator for Java programs, which integrates dynamic symbolic execution, search-based testing, and machine learning to efficiently generate comprehensive class-level test suite. Tardis employs the following approaches: (1) It explores the path space of the target program using dynamic symbolic execution; (2) It instantiates complete test cases using genetic search algorithms; and (3) It applies a priority ranking to symbolic formulas that are more likely to correspond to feasible program paths, utilizing a primitive classification algorithm.

Currently, there are two main frameworks for generating test cases for smart contracts. The first is AGSolt [30], designed specifically for generating test cases for smart contracts with the intention of achieving high branch coverage in Solidity smart contract unit testing. Utilizing both random testing and the DynaMOSA algorithm, AGSolt has managed to attain substantial branch coverage. Empirical evidence indicates that, in terms of branch coverage, DynaMOSA outperforms the fuzzer. Despite lacking additional selection, crossover, and mutation operations, the fuzzer does not exhibit greater speed compared to search-based test case generation. This might be attributed to the fact that DynaMOSA's preference sorting and dynamic selection of coverage objectives techniques that minimizes the time spent on searching for optimal test suite.

The second framework, SynTest-Solidity [31], was put forward by Mitchell et al. It features a command-line interface (CLI), which simplifies testing during the development process and enables developers to modify various parameters related to the test case generation process. Additionally, SynTest-Solidity offers an online web service that allows developers to utilize the tool without needing local installation or configuration. It serves as an automatic test case generation and fuzzing framework for Solidity, encompassing various meta-heuristic search algorithms, such as random search (traditional fuzzing) and genetic algorithms (i.e., NSGA-II, MOSA, and DynaMOSA). In generating test cases for 20 real-world Solidity smart contracts, SynTest-Solidity has demonstrated its effectiveness by achieving an average branch coverage of 61%, thereby validating its utility in testing Solidity smart contracts.

3. AUTOMATIC TEST-CASE GENERATION PROBLEM FOR SOLIDITY SMART CONTRACTS

The central issue addressed in this paper is the automatic generation of test cases, specifically for smart contracts. The goal is to define a test suite that includes multiple test cases for all methods in the contracts under test, where each test case could differ in terms of the quantity and type of test statements included. Two critical considerations underscore this process. Firstly, it is essential to minimize the number of test case statements as lengthy test cases not only increase manual costs but often lead to redundancy in the statement function. This can be mitigated by simplifying interdependent conditional relationships, which consequently shortens the testing cycle, making it preferable to have fewer statements in a test case. Secondly, the target coverage rate needs to be maximized for automatic white-box test case generation. In this context, the concept of coverage criteria is used to evaluate the sufficiency of the current test case for testing the contract under test. Common coverage criteria for traditional programs include line, branch, and mutation coverage, each offering a unique perspective for verifying test adequacy. Studies suggest that a higher testing coverage results in more comprehensive testing of the program under test with the generated test cases [32].

In this section, we analyze and describe the problem of the automatic generation of test cases for smart contracts, reframing it as a many-objective optimization problem. We then describe the search-based test case generation framework, including its comprehensive design, coding architecture, genetic operations, and fundamental algorithmic process, laying the groundwork for subsequent algorithm improvements.

3.1. Control Flow and Control Dependency Graph

A Control Flow Graph (CFG) depicts the execution process of a program [33]. It can be represented as $Graph = (Node, Edge, start, end)$. Where, *Node* denotes the set of block statement nodes, corresponding to bytecode instructions after the smart contract is compiled. Based on different standards, a node could represent a single line of code, multiple lines of statements, or branches. *Edge* symbolizes the intermediary paths between nodes, which could comprise a single line or

multiple lines of statements. However, unlike *Node*, an *Edge* does not physically exist within the code but is an abstract concept that illustrates the flow of data between bytecodes. *start* refers to the entry node of the Control Flow Graph (i.e., the program under test), and *end* corresponds to the exit node of the Control Flow Graph.

The Control Flow Graph (CFG) of a smart contract, obtained from its source code and serving as an intermediate representation, has the following characteristics: (1) It maintains the three structures found in traditional language CFGs: sequence, selection, and loop. (2) Each node in the CFG represents a single valid statement rather than a block of statements. Based on these attributes, the following rules to construct the CFG is defined: each functional structure in the smart contract has its unique CFG (i.e., a sub-CFG). The 'require' statement is treated as a selection structure, and for 'switch' structures, each 'case' is regarded as a selection condition.

While a CFG can provide a comprehensive overview of a program's execution process, certain edges in the CFG can introduce unnecessary nodes, which could lead to inefficient use of resources, as search operations may be wasted on them [34]. To address this inefficiency, the COMPACIFYCFG algorithm, proposed in [30], utilizes the COMPACIFY process to effectively eliminate these redundant nodes. After establishing the control dependency relationships among nodes, a control dependency graph can be extracted from the control flow graph.

3.2. Test Cases and Test Suite

A test case is a sequence of calls targeted at one or more methods within a smart contract, whereas a test suite refers to a collection of test cases formed for the entire smart contract. A test suite can be represented as $T = \{t_1,\ t_2,\ t_3 \dots\ t_n\}$, where each test case can consist of multiple method call sequences, represented as $t = \{statement_1,\ statement_2,\ statement_3 \dots\ statement_k\}$. The critical aspect of the test case optimization problem is the generation of these test cases, particularly the creation of sequences for method calls. As such, it's essential to extract the methods from the smart contract and establish the input and output types, values, and call sequences for each one.

In the test case generation problem, the process begins by identifying the target function of the program under test. Subsequently, the input and output values of various methods are continuously optimized until a particular condition is met—namely, finding the optimal solution for the conditions of the target function. For generating test cases for smart contracts, all branches of the program under test are first extracted to form a target set. This set is then treated as a constraint system, which is ultimately converted into a target function until an optimal solution set is found. In single-objective search, a test case is generated for each target. However, many-objective test case generation, unlike single-objective search, can optimize multiple targets simultaneously. That is, the final generated test case may cover multiple targets. The generated test cases are then combined to form the final test suite. This approach enables the goal of generating many test case coverage targets while minimizing the number of test statements.

3.3. Coverage Criteria

Coverage criteria serve as a crucial benchmark for assessing the adequacy of software testing. Branch coverage, which involves executing all possible outcomes of each branch, ensures that all code branches of the program under test are covered. Compared to other test case coverage criteria (e.g., Statement, Mutation coverahe), branch coverage can detect the correctness of branch conditions in the code under test in a relatively shorter time, thereby improving testing efficiency. Therefore, this paper adopts branch coverage as the evaluation criterion for test cases.

The branch coverage criterion requires that, in the generated test suite, every "TRUE" and "FALSE" branch result of each decision statement in the program under test should be executed at least once. The coverage rate of a specific branch in a test case can be represented by the following function:

$$f_{Branch}(t) = bDist(b_i, t) + accDist(b_i, t) \tag{1}$$

The definition of branch distance is as follows:

$$bDist\ (t, b) = \begin{cases} 0, & \text{if branch has been covered} \\ \frac{f_b(t,b)}{f_b(t,b)+1}, & \text{if current branch has been excuted twice} \\ 1, & \text{other} \end{cases} \tag{2}$$

accDist: This stands for 'approach distance', which is the distance from the actual branch executed in the test case t to the target branch, that is, the number of control dependencies between two branch nodes in the control flow graph.

3.4. Many-objective Test Case Generation

In practical applications, many-objective optimization is a universal challenge spanning across all disciplines. These objectives are intrinsically conflicting and subjected to a variety of constraints [35]. Notably, it is implausible to achieve optimal outcomes for all objectives at once; each objective requires individual weighting. A many-objective optimization problem consists of three or more objective functions and a set of related equality and inequality constraints [31]. This concept can be mathematically described as follows:

$$\begin{gathered} \min\ \ fitness_1(x_1, x_2, x_3, \dots, x_n) \\ \min\ \ fitness_2(x_1, x_2, x_3, \dots, x_n) \\ \dots \\ \min\ \ fitness_k(x_1, x_2, x_3, \dots, x_n) \\ \max\ \ fitness_{k+1}(x_1, x_2, x_3, \dots, x_n) \\ \dots \\ \max\ \ fitness_m(x_1, x_2, x_3, \dots, x_n) \\ constraint.\ p_i(x) \geq 0,\ \ i = 1,2,3 \dots, r \\ onstraint.\ q_j(x) = 0,\ \ j = 1,2,3 \dots, s \end{gathered} \tag{3}$$

In the above equation, $fitness_i(x)${i=1,2,3,…,m} represents the objective function, $p_i(x)$ and $q_i(x)$ represent the constraint functions, $x = \{x_1, x_2, x_3, \dots, x_n\}^T$ is the n-dimensional decision variable. $X = \{x | x \in R^n, g_i(x) \geq 0, h_j(x) = 0, i = 1,2,3, \dots, r,\ j = 1,2,3, \dots, s\}$ represents the feasible domain for the above equation.

In this many-objective optimization problem, there are $m(m \geq 3)$ objective functions, k minimization objective functions, $m - k$ maximization objective function, and $r +$

$s(r, s \geq 0)$ constraint functions, including r inequality constraints and s equality constraints.

In practical engineering design, the decision variables $x_1, x_2, x_3, \ldots, x_n$ are designated and controlled by the user, the objective function is used to evaluate system performance, the constraints represent the limits that the decision variables must meet, and the goal is to optimize these performance indicators. A feasible solution is a set of design variables that satisfy all constraint conditions, and the feasible domain can be defined as the collection of all feasible solutions.

$$\begin{cases} f_1(t) = bDist(b_1, t,) + accDist(b_1, t) \\ f_2(t) = bDist(b_2, t) + accDist(b_2, t) \\ \ldots \\ f_k(t) = bDist(b_k, t) + accDist(b_k, t) \end{cases} \quad (4)$$

This can also be expressed as:

$$min\, f_{Branch,i}(t)\ (i = 1,2, \ldots, s_{Branch}) \quad (5)$$

3.5. Encoding Design

In the context of test case generation, a test case is used as a chromosome. The encoding represents individuals as input sequences, each comprising an environment block and a transaction block, both of which are encoded as key-value mappings. The environment includes block information, such as current timestamp and block number, return values of calls, data size, and external code size, etc. The transaction comprises the address of the sending account (from), transaction amount (value), maximum gas amount for contract execution (gas limit), and the input data for contract execution (data). The input data is represented as a value array, where the first element is always the function selector, and the remaining elements represent function parameters. The function selector is calculated using the Application Binary Interface (ABI) and extracts the first four bytes of the Keccak (SHA-3) hash of the function signature. The individual encoding method in the test case generation problem is shown in Figure 1.

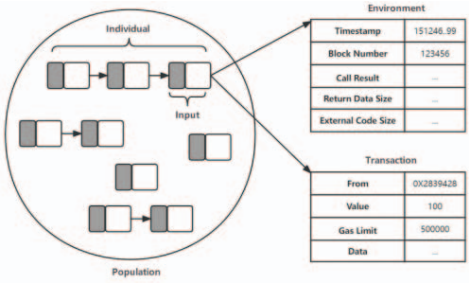

Figure 1. Encoding design

3.6. Population initialization

After extracting the required information, the test case population is initialized randomly with a fuzzing approach. Each test case compose of various statement, and can be represented as $t = \{statement_1, statement_2, statement_3 \ldots statement_k\}$. Those statements can be categorized into two types. The first type is the construction statement, which is used to deploy smart contracts on the blockchain. Such a statement serves as the first statement $statement_1$ for every test case t, ensuring each test case that can call function statements instantiates a new instance of a smart contract. This type of statement contains the information necessary for deploying relevant smart contract instances on the blockchain, including the input variables required by the smart contract constructor and transaction metadata, such as the amount of Ether sent with the transaction and the account sending the transaction. The second type is the function statement, which is used to create transactions that call functions within the deployed smart contracts. In fact, the only way to interact with smart contracts in Ethereum is to send transactions to their addresses. In the test cases, all statements apart from the first one (i.e., the construction function statement) are function statements responsible for traversing the branches of the smart contracts. This type of statement contains references to the function to be covered, its input variables, and transaction metadata. A group of test cases is initialized by creating N random test cases, where N is the population size, i.e., the number of test cases in any given generation.

3.7. Search Operations

The search operations of the genetic algorithm primarily include selection, crossover, and mutation [36]. This replicates the biological evolution process of "survival of the fittest," aiming to guide the population's evolution towards superior genes, thereby achieving an optimal solution.

3.7.1.Selection Operation

The core idea of the selection operator is to select two individuals from the current population as parents, who then generate the next generation of the population following certain rules. During the evolutionary process, it is generally believed that individuals with higher fitness values are more suitable as parents. However, this poses a risk of losing other genes, leading to population convergence around a solution and falling into local optima. Therefore, preserving excellent genes, maintaining population diversity, and improving genetic efficiency are all equally important.

Based on the idea of a tournament, during each evolution, a certain number of individuals are randomly selected from the parents, and the one with the highest fitness is chosen for the genetic operation. This process is repeated until the size of the child population is the same as the parent population. This method greatly maintains population diversity. Therefore, the selection operator based on the tournament to ensure population diversity and convergence is used for the smart contract test case generation problem.

3.7.2.Crossover Operation

The crossover operator involves performing a crossover operation on two individuals selected from the parent population. First, a crossover point is determined, and then the gene strings are swapped at the crossover point while maintaining the rest of the positions unchanged, thus generating two new individuals. Single-point crossover entails pairing individuals in the population, selecting an arbitrary position in the individual's encoding string as the crossover point, and swapping the subsequent gene strings based on this point, with other positions remaining unchanged. This can produce two new individuals. This is shown in Figure 2.

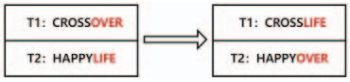

Figure 2. Single-point crossover

Single-point crossover requires less computational effort compared to other crossover operations. For large-scale problems, it can search the solution space faster, and single-point crossover can preserve more parent information in two individuals. It effectively inherits superior genetic information and accelerates convergence. Therefore, the single-point crossover is used for the smart contract test case generation problem.

3.7.3.Mutation Operation

Mutation involves modifying some encodings on the gene positions of the current chromosome (individual) to form a new individual. It introduces randomness to discover unexplored regions in the search space, thereby enhancing the algorithm's global search capability. To ensure the uniformity of individual mutation, three mutation operations, including insertion, modification, and deletion of the statements contained in the test case, each with a 1/3 probability, is used for the smart contract test case generation problem.

- Deletion: Each statement of a test case of length S is deleted with a probability of 1/S, ensuring that the test case can still run normally after deletion.
- Modification: Each statement of a test case of length S is modified with a probability of 1/S. Different variable types have different value ranges and modification methods. Value generation includes two methods: random and mutation pool value extraction.
- Insertion: Unlike deletion and modification operations, this step requires generating method calls according to the environment and method, followed by insertion.

3.8. Test Case Generation Framework

The test case generation framework used in this paper, as shown in Figure 3, is based on [30], and is primarily divided into two main parts: initialization module and test loop module. During the initialization phase, the characteristics of the smart contract (program under test) are extracted to create the first generation of test cases, setting the stage for further improvement during the test loop.

Within this framework, the input consists of a smart contract written in Solidity language, which is a combination of code and data residing at a specific address on the Ethereum blockchain. This is followed by the ABI (Application Binary Interface) Analysis wherein the Solidity code is compiled into bytecode, and an ABI file is created that contains vital information such as function names and input types necessary for test case generation, capturing also the hard-coded values of the contract. Next, to track the branches to be traversed and those already covered, CFG Extraction is performed using the Python EVM-CFG-BUILDER library to extract the control dependencies from the bytecode, with each branch node's opcode identified for branch distance calculations. Subsequently, COMPACIFY [30] algorithm is executed on redundant nodes to eliminate them, aiming at budget reduction in the CDG Generation phase.

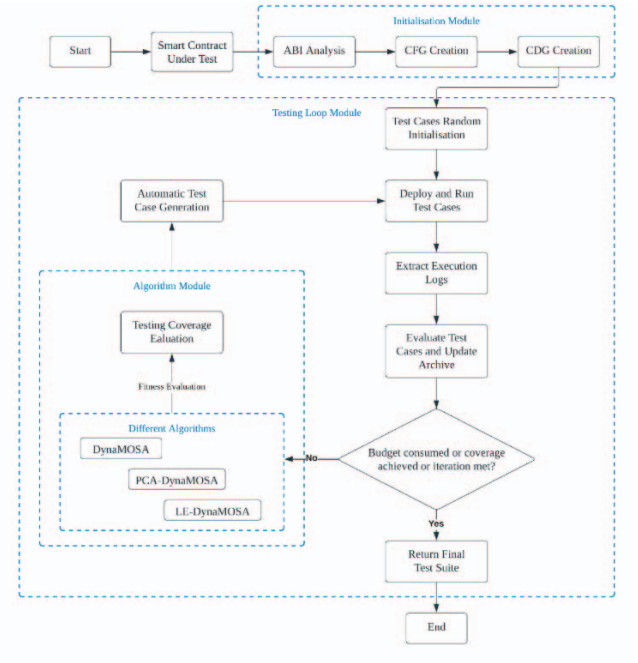

Figure 3. The overall framework

During the test loop, an exhaustive search for the best test case is conducted until the budget is expended. This loop can be broken down into two parts, starting with the algorithm module that utilizes integrated algorithms such as DynaMOSA, PCA-DynaMOSA, and LE-DynaMOSA. These algorithms, all based on the genetic principle and inspired by biological evolution, work together to obtain comprehensive test cases through iterative application of evaluation, selection, crossover, and mutation. The result archive module then follows, wherein each test case is run on the Ethereum blockchain, beginning with a constructor statement. After executing and evaluating all test cases to generate a distance vector for fitness description, the normalized branch distances are updated, and all traversed edges and uncovered branches are calculated. Finally, if a test case excels over the best one found so far for a particular branch, it is stored in an archive that systematically tracks the best test case for each branch, culminating in a return at the end of the process, thereby reflecting the framework's methodical approach to test case generation.

4. Utilizing Many-Objective Search and Dimensionality Reduction in the Auto-Generation of Solidity Tests for Blockchain Smart Contracts

DynaMOSA [15], proposed by Panichella, is one of the most popular many-objective test case generation algorithms, it has been used to generate efficient test cases for variety of systems and programing languages [18], [20], [23]. It incorporates strategies such as archiving, dynamic selection of objectives and preference sorting based on the foundation of NSGA-II. On one hand, the preference sorting function determines the test case with the lowest objective score (e.g., branch distance + approach level for branch coverage) for each uncovered target. And assign all these test cases to rank 0, in other words, give them a higher chance of surviving into

the next generation. On another hand, the dynamic selection of objectives uses the control dependency graph to derive which targets are independent of any others (that is the targets that are free of control dependencies) and which ones can be covered only after satisfying previous targets in the graph, and selects only those targets that are free of control dependencies as the initial set of objectives.

Despite the substantial progress that existing multi- and many-objective optimization algorithms have made in solving automatic test case generation problems, they still face considerable challenges. Most notably, these algorithms often experience a significant decrease in performance when applied to large-scale many-objective optimization problems. This issue necessitates the exploration of strategies to reduce the dimensionality of the objectives, as such a reduction could potentially enhance the performance of the test case generators.

Dimensionality reduction refers to the process of transforming high-dimensional data into lower-dimensional data based on certain criteria. It can decrease computational requirements and enhance the search efficiency of the algorithm by eliminating redundant objectives. Therefore, it can be utilized to address hyper-many-objective optimization problems. In the context of test case generation, strategies like preference sorting and dynamic selection of objectives, deployed in the DynaMOSA algorithm can help alleviate issues associated with a high number of objectives to some extent. However, this solution remains insufficient when generating effective test cases for complex modern software programs, with tens, and sometimes even hundreds, of coverage objectives (branches) in a single class.

Hence, in this paper, we propose a solution to the performance degradation of the DynaMOSA algorithm in large-scale many-objective optimization problems by incorporating dimensionality reduction algorithms in smart contract test case generation. Our primary objective is to reduce the dimensionality of the objective space in the test case generation problems, filtering out redundant and invalid feature information. Then, carry out many-objective sorting on population individuals using the reduced objective set. Moreover, the application of dimensionality reduction for test case generation problem is primarily justified by three considerations: first, the inherently vast quantity of objectives within the problem; second, the potential for a single test case to encompass multiple objectives; and third, the presence of correlations amongst different coverage objectives.

4.1. PCA for Smart Contract Test Case Generation

Principal Component Analysis (PCA) [37] is a statistical technique that uses orthogonal transformation to convert a set of observations of potentially correlated variables into a set of values of linearly uncorrelated variables. Moreover, PCA serves as a tool to reduce multi-dimensional data to a lower dimension while retaining most of the information. It incorporates standard deviation, covariance, and eigenvectors. The main steps in PCA are as follows:

- Column vectors and row vectors of size N^2 represent a collection of M objectives $(B_1, B_2, B_3, \dots, B_m)$ of size $N * N$.
- The average value of the objective μ is described as:

$$\mu = \frac{1}{m}\sum_{n=1}^{m} B_n \quad (6)$$

- Each objective has a different average value:

$$W_i = B_i - \mu \quad (7)$$

- The covariance matrix is calculated as follows:

$$C = \sum_{n=1}^{m} w_n {w_n}^T = AAT \quad (8)$$

Where $A = [W_1 W_2 W_3 \dots W_n]$.

- Compute the covariance matrix C's eigenvectors U_l and eigenvalues λ_l.
- Form a new matrix from the first k eigenvectors corresponding to the largest eigenvalues.
- Project the sample points onto the chosen eigenvectors to obtain the result.

The principal advantage of Principal Component Analysis (PCA) is its low sensitivity to noise, along with its reduced requirements for storage capacity and memory. Furthermore, it enhances efficiency due to the dimensionality reduction of the data during the processing. The benefits of PCA encompass three main points: firstly, there is no redundancy in data represented by orthogonal components; secondly, it lessens the complexity of the objectives; and thirdly, PCA effectively minimizes noise by selecting the basis associated with the largest variances, while smaller changes are automatically disregarded [37].

While PCA brings certain benefits, it's not without its limitations. One notable drawback is that the process of dimensionality reduction can be computationally demanding, consuming a significant portion of the system's resources. Additionally, accurately calculating the covariance matrix, a crucial step in PCA, can pose a challenge [37]. Finally, PCA often struggles to capture invariant features in the data, which could limit its effectiveness in some applications. Therefore, to minimize the computational time, this paper also employs the Laplacian Eigenmap (LE) method for dimensionality reduction, in addition to PCA.

4.2. LE for Smart Contract Test Case Generation

Laplacian Eigenmap (LE) [38] is an unsupervised nonlinear finite element analysis technique that seeks low-dimensional data while preserving the "local properties" of the manifold. Initially, LE generates a neighborhood graph G', where each datum x_a is linked to its nearest neighborhood. All x_a and x_b in G' are connected by an edge, and the Gaussian kernel function is employed, where the weight of the edge is assessed as:

$$Weight_{ab} = e^{\frac{-|x_a - x_b|^2}{2\sigma^2}} \quad (9)$$

where σ specifies the variance of the Gaussian function leading to the adjacency matrix (W). To reduce the dimensions from X to Y, the cost function defined by LE is minimized as:

$$\Phi(Y) = \sum_{ab} |y_a - y_b|^2 . Weight_{ab} \quad (10)$$

Optimizing the cost function implies that the distance between x_a and x_b in the high-dimensional space is smaller, and consequently, the distance between y_a and y_b in the low-dimensional space is also smaller. Furthermore, by calculating the diagonal matrix D and the Laplacian matrix L of G', the cost function in the above equation can be rephrased as an eigenvalue decomposition problem, as follows:

$$\sum_{ab}|y_a - y_b|^2 . Weight_{ab} = 2.\mathrm{Y.L.}Y^T \quad (11)$$

where L can be defined as L = W - D; D is a diagonal matrix with elements being the row sum of W, like $D_{aa}=\sum_b W_{ab}$, and the elements of W represent whether a pair of vertices are adjacent; the elements of W are either 1 or 0, where diagonal elements are 0. Hence, the aim is to make $\Phi(Y)$ subject to minimizing $\mathrm{Y.L.}Y^T$.

The main steps of LE are as follows:

- Construct a neighborhood graph using the adjacency matrix.
- Calculate the weight of each edge of the adjacency graph.
- Obtain the new space by eigen-decomposition through optimizing the cost function.

In the context of test case generation, the LE algorithm is introduced not only to reduce the dimensionality of the objective set-in test case generation problems but also for comparison with the PCA algorithm. We consider the relative strengths and weaknesses of the algorithms in terms of branch coverage and time efficiency. This enables the appropriate algorithm to be selected according to the actual situation in real-world scenarios.

Algorithm 1: Improved Multi-objective Test Case Generation Algorithm Based on Dynamic Grouping and Dimensionality Reduction Strategy

Input:
U: Set of objectives
Output:
T : Test suite
Algorithmic Process:

```
Procedure
    U' = Select targets in U with not control dependencies
    t = 0
    Create initial population : P_t
    archive = UPDATE_ARCHIVE(P_t. ∅)
    U' = UPDATE_TARGETS(U', Graph) //Update objectives based on the control dependency graph and objective correspondence
    while TERMINATION_CONDITION is not met do
        U'[i] = group(U') //Group objectives by a certain number
        U'_PCA [i]= PCA(U') || U'_LE [i]= LE(U') //Perform dimensionality reduction on the grouped objectives separately
        U_PCA = SORT(U'_PCA [i]) //Combine the reduced objectives
        PREFERENCE_SORT(U_PCA) //Perform preference sorting
        CROWED_SORT(F_s, U_PCA) //Perform crowding distance sorting
    end while
    T = archive //Return the final archive as the optimal test case set
end Procedure
```

4.3. Algorithmic Process for Test Case Generation based on Dynamic Grouping and Dimensionality Reduction Strategy

As inferred from the analysis of dimensionality reduction algorithms, the process of reducing the objectives will also entail additional computational time. If the number of objectives in the test case generation problem is large, the computational process of the dimensionality reduction algorithm will accordingly be more complex. This could lead to issues such as lengthy computation time and difficulties in convergence. Therefore, it can be considered to reduce the dimension of the objectives in a dynamically grouped manner: group the objectives by a certain number, carry out dimensionality reduction separately, and then integrate the reduced objectives to obtain the final objective set. The pseudocode for the improved many-objective test case generation algorithm based on dynamic grouping and dimensionality reduction strategy is shown in Algorithm 1.

Naturally, within the grouping strategy, the group size is hard to determine in advance, given its impact on the algorithm's performance. The appropriate group size needs to be chosen empirically, based on the number of objectives and through experimentation in real-world situations. This paper intends to conduct experiments with varying group sizes to determine an empirical value. The flowchart for the test case generation algorithm of DynaMOSA, based on dynamic grouping and dimensionality reduction strategies, is shown in Figure 4.

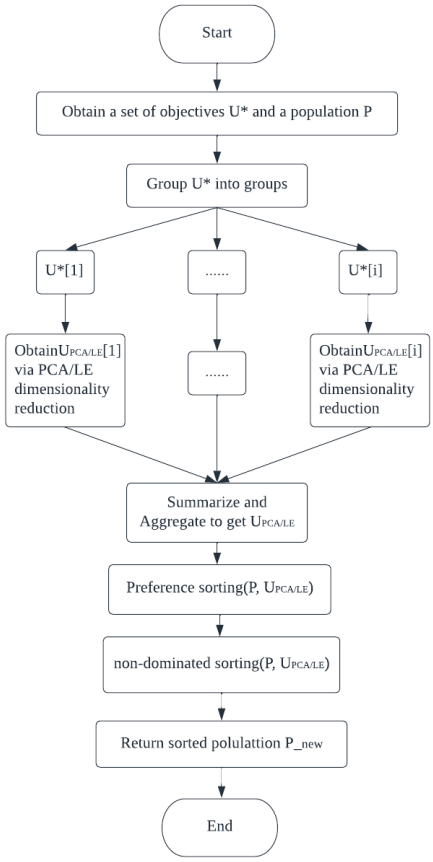

Figure 4. Flowchart of test case generation algorithm based on dynamic grouping and dimensionality reduction strategy

5. Experimental Design and Result Analysis

In this section, we will conduct experiments using our proposed algorithms, and compare their performance with that of DynaMOSA, the state-of-the-art many-objective test case generation algorithm implemented in AGSolt [30]. We have named our novel many-objective test case generation algorithms using PCA and LE as PCA-DynaMOSA and LE-DynaMOSA, respectively.

The experiments designed and conducted in this study were primarily aimed at answering the following research questions (RQs) for many-objective test case generation of smart contracts:

- RQ 1: When considering branch coverage during the generation of test cases for smart contracts, how does the performance of the PCA-DynaMOSA and LE-DynaMOSA algorithms compare to both the DynaMOSA algorithm and each other?
- RQ 2: Regarding search efficiency during the test case generation process, how do the PCA-DynaMOSA and LE-DynaMOSA algorithms stack up against the DynaMOSA algorithm, as well as against each other?
- RQ 3: Under the dimensionality reduction strategy, to what extent does the grouping size affect the test coverage rate and search efficiency? Moreover, how can insights gleaned from the experimental results guide the choice of grouping size in practical scenarios?

5.1. Experimental Design

5.1.1.Evaluation Metrics

To benchmark the efficacy of our proposed algorithms in generating smart contract test cases, we leverage a set of evaluation metrics. These metrics are applied during experimental testing and are crucial in facilitating a comprehensive comparative analysis of the performance of different algorithms.

Testing coverage: The coverage rate evaluates the sufficiency of test cases generated by the algorithm for smart contract under test. We choose branch coverage as the evaluation criterion, and it can be calculated using the formula below:

$$\text{Branch Coverage} = \frac{\text{Number of covered branches}}{\text{Total number of branches to be covered in Smart Contract}} * 100\% \quad (12)$$

Execution Time: Execution time mainly includes the total time of the algorithm and the average total time, represented as follows:

$$\text{Average total time} = \frac{\sum_{i=0}^{N} \text{Time required to generate ith test cases}}{\text{N}} \quad (13)$$

5.1.2.Experimental Setup

The primary focus of this study is to improve and optimize the state-of-the-art algorithm, DynaMOSA [15] in AGSolt [30], for smart contract test case generation. The current framework, written in Python, operates within the Ubuntu system. The detailed specifications of the experimental test platform are presented in Table 1.

Table 1. Experimental environment

Hardware	Intel(R) Core(TM) i9-10900F CPU @ 2.80GHz
Software	Ubuntu 22.04
	Python 3.9
	Solidity 0.4.9

Table 2. List of smart contracts under test

Smarta Contract	# of Branches	Smarta Contract	# of Branches
BasicToken	8	LotteryFor10	12
Casino	11	LotteryMultipleWinners	27
DosAuction	7	MyAdvancedToken	3
EasyPayAndWithDraw	8	OpenAddressLottery	19
EZCoin	11	PullOverPush	7
EtherBank	17	ProveIT	5
FixedSupplyToken	22	PermissionGroups	66
FundRaising	21	Rubixi	66
Greeter	65	RICO	13
Greeter3	57	Reentrance	14
GuardCheck	14	SecureAuction	6
Gift_1_ETH	18	tutorial-25	58
GuessTheNumberChallenge	8	tutorial-26	71
IdentityManager	7	VulnerableTwoStep	10

5.1.3.Datasets

The experimental data primarily comprises some of the most popular Solidity smart contract projects available on GitHub. As illustrated in Table 2, among these twenty-eight projects, the smallest project has as few as three branches, while the largest project encompasses as many as seventy branches. Moreover, the smart contracts selected in this paper adhered to the following criteria: (1) They do not have any user-defined inputs; (2) They are standalone programs that do not call other smart contracts during their execution.

5.1.4.Experiment Setup

Considering a plethora of experimental validation outcomes conducted by a multitude of researchers [15], [16], [20], [23], the AGSolt framework has been configured using the algorithm parameters that were initially set by DynaMOSA [15]. To avoid unintended bias or distortion in our results, we consistently applied the same algorithm parameters across the experiment. Their exact configuration is outlined in Table 3.

Table 3. Parameter settings for the proposed algorithm

Parameter name	Value
Population Size	50
Coverage Criterion	Branch Coverage
Number of Accounts	10
Number of Individuals in Binary Tournament Selection	10
Crossover Probability	0.75
Mutation (Replace/Delete/Insert) Probability	0. 3333333

Table 4. Group settings for LE-DynaMOSA

Group No.	Group size
LED3	3
LED6	6
LED9	9
LED12	12

In the LE-DynaMOSA algorithm, selecting an appropriate group size is crucial to achieving high algorithmic efficiency. The precise configuration for group size is shown in Table 4.

5.2. Experimental Results and Analyses for Smart Contract Test Case Generation

5.2.1.Different Coverage Achieved for Each Smart Contract (RQ1)

Table 5 shows the branch coverage performance of the DynaMOSA, PCA-DynaMOSA, and LE-DynaMOSA algorithms. In Table 5, the "-" indicates that there is no significant difference in the performance of the algorithm for a particular project.

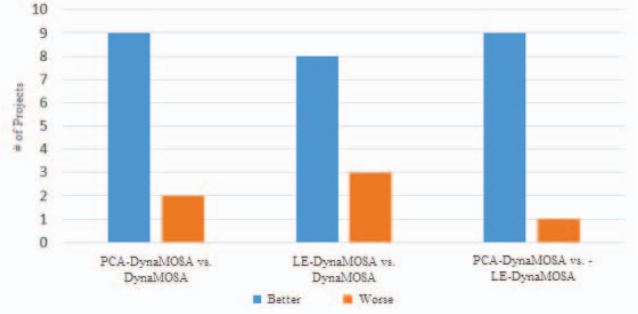

Figure 5. Number of projects with superior and inferior results in branch coverage

Table 5. Mean branch coverage achieved for each smart contract

Smart Contract	Branch Coverage (%)			PCA VS DynaMOSA (%)	LE VS DynaMOSA (%)	LE VS PCA (%)
	DYNAMOSA	PCA-DYNAMOSA	LE-DYNAMOSA			
BasicToken	100	100	100	-	-	-
Casino	57.5	63.75	60	6.25	2.5	-3.75
DosAuction	100	100	100	-	-	-
EasyPayAndWithDraw	100	100	100	-	-	-
EZCoin	100	100	100	-	-	-
EtherBank	69.12	73.53	72.35	4.41	3.23	-1.18
FixedSupplyToken	98.18	100	95.45	1.82	-2.73	-4.55
FundRaising	100	100	100	-	-	-
Greeter	99.84	98.46	98.46	-1.38	-1.38	-
Greeter3	100	100	100	-	-	-
GuardCheck	92.85	92.85	92.85	-	-	-
Gift_1_ETH	74.44	84.44	77.78	10	3.34	-6.67
GuessTheNumberChallenge	100	100	100	-	-	-
IdentityManager	100	100	100	-	-	-
LotteryFor10	100	100	100	-	-	-
LotteryMultipleWinners	84.44	78.15	77.78	-6.29	-6.66	-0.37
MyAdvancedToken	100	100	100	-	-	-
OpenAddressLottery	100	100	100	-	-	-
PullOverPush	100	100	100	-	-	-
ProveIT	100	100	100	-	-	-
PermissionGroups	94.84	95.45	96.97	0.61	2.13	1.52
Rubixi	10.6	19.69	18.18	9.1	7.58	-1.52
RICO	100	100	100	-	-	-
Reentrance	86.42	94.29	92.86	7.86	6.43	-1.43
SecureAuction	100	100	100	-	-	-
tutorial-25	8.62	21.72	20.69	13.1	12.07	-1.03
tutorial-26	4.22	32.54	22.54	28.32	18.32	-10
VulnerableTwoStep	100	100	100	-	-	-
Mean over all SC	85.04	87.67	86.64	2.64	1.60	-1.03

Table 5 presents the branch coverage rates achieved by the DynaMOSA, PCA-DynaMOSA, and LE-DynaMOSA algorithms. It is discernible that both our proposed algorithms, PCA-DynaMOSA and LE-DynaMOSA, outperform the baseline DynaMOSA algorithm. On most of the smart contracts, these two improved algorithms yield similar or superior branch coverage results, albeit there are a few cases where the coverage rates decreased. On average, the PCA-DynaMOSA algorithm enhances branch coverage by 2.64% compared to DynaMOSA, while the LE-DynaMOSA algorithm registers an average improvement of 1.60% on the smart contracts under test.

Table 6. Number of smart contracts with Better and Worse results in branch coverage criteria

Algorithm	# of "Better"	# of "Better" (%)	# of "Worse"	# of "Worse" (%)
PCA vs. DynaMOSA	9	32.14	2	7.14
LE vs. DynaMOSA	8	28.57	3	10.71
PCA vs. LE	9	32.14	1	3.57

The superior performance of the proposed algorithms can be attributed to the challenges inherent to many-objective optimization, where discerning the relative merit of individuals becomes exceedingly difficult, even without the consideration of computational resources. Traditional algorithms such as non-dominated sorting can pose difficulties when applied to search in hyper-objective context. The dimensionality reduction algorithms address this issue by reducing the dimensionality of the test case generation problem space and constraining the number of objectives within a manageable limit. This, in turn, allows for a degree of optimization in the solution process. The effectiveness of the algorithms is statistically analyzed as shown in Table 6 and Figure 5.

As evidenced by Table 6 and Figure 5, both the PCA-DynaMOSA and LE-DynaMOSA algorithms exhibit improvements in branch coverage over the DynaMOSA algorithm. Specifically, The PCA-DynaMOSA algorithm shows superior performance in 32.14% of the cases and inferior performance in 7.14% of the cases. For the LE-DynaMOSA algorithm, superior performance is seen in 28.57% of the cases, and inferior performance in 10.71% of the cases. Moreover, PCA-DynaMOSA demonstrates superior performance compared to the LE-DynaMOSA, with superior performance in 32.14% of cases and inferior performance in 3.57% of cases. This is due to the dimensionality reduction of the LE algorithm, which primarily deals with the correlation between different coverage targets, combining different branches into inclusive or dependent relationships. This could potentially result in some objectives being difficult to cover, resulting in lower branch coverage rates. From the above analysis, it can be concluded that in the smart contracts test case generation problem, both dimensionality reduction strategies can show improved results in terms of branch coverage, with the PCA-DynaMOSA algorithm delivering the better performance among the two without considering the time consumption.

Table 7. Comparison of time consumption

Smart Contract	Time Consumption (s)			PCA VS DynaMOSA (s)	LE VS DynaMOSA (s)	LE VS PCA (s)
	DYNAMOSA	PCA-DYNAMOSA	LE-DYNAMOSA			
BasicToken	93.67	68.33	60.86	25.34	32.81	7.47
Casino	5307.82	4729.1	4509.77	578.72	798.06	219.33
DosAuction	70.05	51.86	52.89	18.19	17.16	-1.03
EasyPayAndWithDraw	138.86	56	68.9	82.86	69.96	-12.9
EZCoin	156.25	33.54	52.9	122.71	103.35	-19.36
EtherBank	5968.44	5296.47	5209.78	671.97	758.65	86.69
FixedSupplyToken	2179.91	1693.77	1590.54	486.14	589.38	103.23
FundRaising	82.1	69.92	62.94	12.18	19.16	6.99
Greeter	168.33	152.67	132.41	15.66	35.92	20.26
Greeter3	138.54	127.2	118.95	11.33	19.59	8.25
GuardCheck	1390.39	2867.57	2705.8	-1477.19	-1315.42	161.77
Gift_1_ETH	659.95	637.33	611.31	22.62	48.64	26.02
GuessTheNumberChallenge	57.87	52.86	57.11	5.02	0.77	-4.25
IdentityManager	75.14	63.23	62.7	11.91	12.44	0.53
LotteryFor10	75.47	61.47	61.05	14	14.43	0.43
LotteryMultipleWinners	1005.42	925.33	895.8	80.09	109.62	29.52
MyAdvancedToken	130.25	127.35	114.71	2.9	15.54	12.64
OpenAddressLottery	69.6	61.32	57.42	8.28	12.18	3.9
PullOverPush	160.52	150.4	134.8	10.12	25.72	15.6
ProveIT	210.74	199.43	174.72	11.31	36.02	24.71
PermissionGroups	6237.36	5992.43	5920.74	244.93	316.63	71.69
Rubixi	602.72	593.46	567.61	9.26	35.11	25.85
RICO	72.29	61.95	62.71	10.34	9.57	-0.76
Reentrance	5647.86	5603.28	5438.9	44.57	208.96	164.38
SecureAuction	58.79	58.21	54.7	0.58	4.09	3.51
tutorial-25	630.75	824.17	795.82	-193.42	-165.07	28.35
tutorial-26	783.86	926.06	854.21	-142.2	-70.35	71.85
VulnerableTwoStep	76.03	63.97	59.71	12.06	16.31	4.26
Mean over all SC	1151.75	1126.74	1088.92	25.01	62.83	37.82

5.2.2.Search Performance and Time Consumption Achieved by the Algorithm (RQ2)

The time consumption of the DynaMOSA, PCA-DynaMOSA, and LE-DynaMOSA algorithms is presented in Table 7. Table 7 presents the time consumption of the DynaMOSA, PCA-DynaMOSA, and LE-DynaMOSA algorithms for smart contract test case generation. As can be seen, both PCA-DynaMOSA and LE-DynaMOSA, outperforms the DynaMOSA algorithm. These algorithms show similar or better time efficiency on most smart contracts, with a minor increase in time consumption in a few cases. LE-DynaMOSA consistently yielded the best results, whereas DynaMOSA was the least efficient. The PCA-DynaMOSA and LE-DynaMOSA algorithms, on average, reduced time consumption by 25.01 and 62.83 seconds, respectively, compared to DynaMOSA across all tested smart contracts. This is because, after dimensionality reduction was performed on smart contracts with numerous objectives, the optimization process covered multiple objectives with a single test case. This resulted in a corresponding reduction in the final number of objectives, thereby increasing the time efficiency of test case generation. However, this also introduces certain drawbacks, such as the extra computational time required for the dimensionality reduction. Therefore, striking a balance between the pros and cons of dimensionality reduction algorithms is crucial in tackling this issue. A statistical analysis of the superior and inferior performance in time efficiency among different algorithms is illustrated in Table 8 and Figure 6.

Table 8. Number of smart contracts with Better and Worse results in time consumption

Algorithm	# of "Better"	# of "Better" (%)	# of "Worse"	# of "Worse" (%)
PCA vs. DynaMOSA	25	89.29	3	10.71
LE vs. DynaMOSA	25	89.29	3	10.71
PCA vs. LE	5	17.86	23	82.14

As inferred from Table 8 and Figure 6, both the PCA-DynaMOSA and LE-DynaMOSA algorithms exhibit significant time efficiency improvements. The PCA-DynaMOSA algorithm shows superior performance in 89.29% of the cases, and inferior in 10.71% of the cases, compare to DynaMOSA. The most prominent growth can be seen in the '*Casino*' and '*EtherBank*' smart contracts. For the LE-DynaMOSA algorithm, superior performance is seen in 89.29% of the cases, and inferior performance in 10.71% of the cases, compare to DynaMOSA. Moreover, the LE-DynaMOSA algorithm outperforms the PCA-DynaMOSA algorithm, with superior time efficiency in 82.14% of cases and inferior performance in 17.86% of cases. This could be attributed to the LE's approach to dimensionality reduction, which aims to consolidate similar objectives. In contrast, the number of objectives post-PCA reduction is determined by how much information can be preserved following reduction. The impact of reduction is less noticeable for smart contracts with fewer objectives, and the PCA reduction process is inherently more complex.

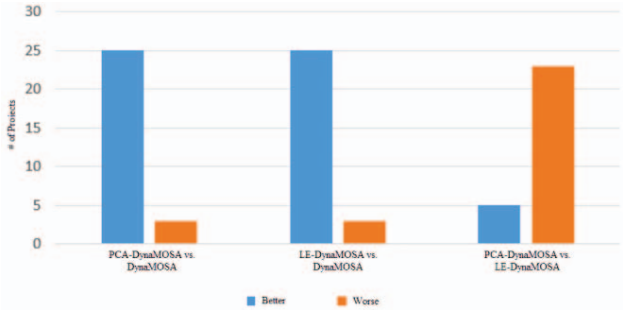

Figure 6. Number of projects with superior and inferior results in time consumption

Based on our analysis, we find that both dimensionality reduction strategies, when applied to the test case generation problem for smart contracts, show enhanced time efficiency. Notably, LE-DynaMOSA yields superior results. Depending on needs, an appropriate dimensionality reduction algorithm can be selected. If higher test case coverage is desired, PCA-DynaMOSA is recommended. Conversely, when balancing coverage and time efficiency, LE-DynaMOSA serves as the optimal choice, offering solid coverage while maintaining time efficiency.

5.2.3. Performance of Proposed Algorithm Under Different Group Settings (RQ 3)

The method adopted in this section of the experiment involves dividing the objectives of smart contracts into groups. The group sizes are set to 3, 6, 9, and 12. The analysis is then carried out based on the average coverage rate. The experiment results are as follows.

Table 9. Number of smart contracts with Better and Worse results in branch coverage and time consumption

Group No.	Group size	Mean branch coverage	Mean time consumption
LED3	3	86.04	1165.91
LED6	6	**87.84**	**1074.87**
LED9	9	87.23	1097.82
LED12	12	86.32	1121.57
LE-DynaMOSA	-	86.64	1088.92
DynaMOSA	-	85.04	1151.75

As can be seen from Table 9, the grouping of the LE algorithm indeed influences the performance of the test case generation algorithm for smart contracts. If the grouping is too large or too small, it may not only fail to improve the results but may even decrease the efficiency of the algorithm. Thus, the choice of group size is a key issue. For instance, the search results of the LED3 and LED12 group algorithms show no significant improvement compared to the algorithm with no grouping. In some cases, their coverage rate is even lower, and the time consumption is relatively increased. This is because when the grouping is too small, the number of objectives before and after reduction changes little or not at all. The process of reduction increases time consumption and wastes computational resources. On the other hand, when the grouping is too large, there may be difficulties in convergence of the reduction algorithm for smart contracts with a vast number of objectives, which leads to increased time consumption.

However, for most cases, when a reasonable group size is set, the LE algorithm can effectively enhance the coverage rate and time efficiency of test case generation for smart contracts. As shown in Table 9, both LED6 and LED9 show certain improvements compared to the DynaMOSA algorithm. But when comparing LED6 and LED9, the former increases the coverage rate by 0.61% and reduces the average time consumption by 22.95 seconds. Therefore, LED6 could be chosen as the final group number for the LE-DynaMOSA algorithm when considering these factors. Despite LED6 showing optimal performance in this experiment, we must acknowledge, depending on the complex of the smart contract and the testing criterion chosen, the number of objectives in a smart contract test case generation problem can be varied. Therefore, realistic analysis and comparison are vital when selecting group size to effectively improve efficiency.

6. CONCLUSION

This paper presented new approaches to the challenge of automatically generating test cases for smart contracts in the context of excessive number of coverage objectives. More specifically, two dimensionality reduction strategies were utilized to reduce the objective space of the problem and to enhance the performance of the many-objective optimization algorithm for smart contract test case generation, namely, PCA and LE. The effectiveness of these algorithms was evaluated using 28 open-source Solidity smart contract projects from GitHub. Our results demonstrated noticeable improvements in both test case coverage and time efficiency when compared to the state-of-the-art DynaMOSA algorithm adapted in AGSolt framework. Specifically, PCA-DynaMOSA and LE-DynaMOSA algorithms are on average notably higher in 32.14% and 28.57% of the classes under test for branch coverage, and boosted average branch coverage by 2.64% and 1.60%, respectively, over the DynaMOSA algorithm. Moreover, time consumption was reduced by an average of 25.01 seconds and 62.83 seconds for PCA-DynaMOSA and LE-DynaMOSA algorithms, respectively. When considering test case coverage, the PCA-DynaMOSA algorithm was shown to be more effective. However, in terms of a balance between coverage and time efficiency, the LE-DynaMOSA algorithm was found to be the optimal choice. The grouping of objectives was found to influence the performance of the test case generation, with group size six providing the best results among the groups tested, demonstrating an increase in coverage rate by 0.61% and a decrease in average time consumption by 22.95 seconds. Despite the promising results, some limitations should be noted. The impact of dimensionality reduction was less noticeable for smart contracts with fewer objectives, and PCA reduction is inherently more complex. Selecting the appropriate group size is crucial and highly dependent on the complexity of the smart contract and the testing criteria chosen. The future work of the study may include expanding the framework's support for the latest smart contract versions and developing a user-friendly GUI for improved usability, integrating dynamic symbolic execution for fine-grained test case searches and analyzing historical code fault to improve the framework's bug detection capabilities and combining test case selection and prioritization methods to enhance fault detection efficiency.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Decentralized business review, 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, pp.1-32, 2014.

[3] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," 2021 Australasian Computer Science Week Multiconference, pp. 1–10, 2021.

[4] X. Mei, I. Ashraf, B. Jiang, and W. K. Chan, "A Fuzz Testing Service for assuring smart contracts," 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 544–545, 2019.

[5] H. Watanabe et al., "Blockchain contract: Securing a blockchain applied to smart contracts," 2016 IEEE International Conference on Consumer Electronics (ICCE), pp. 467–468, 2016.

[6] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," IEEE Access, vol. 7, pp. 77894–77904, 2019.

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269, 2016.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum Smart Contracts (SOK)," In Principles of Security and Trust: 6th International Conference, Springer, pp. 164–186, 2017.

[9] W. Zou et al., "Smart contract development: Challenges and opportunities," IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 2084–2106, 2021.

[10] M. Di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum Smart Contracts," 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), pp. 69–78, 2019.

[11] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, Consensus, and future trends," 2017 IEEE International Congress on Big Data (BigData Congress), pp. 557–564, 2017.

[12] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for ethereum smart contract," arXiv preprint arXiv:1908.03707, 2019.

[13] R. A. Khanum et al., "On the hybridization of global and local search methods," Journal of Intelligent & Fuzzy Systems, vol. 35, no. 3, pp. 3451–3464, 2018.

[14] Y. Dong and J. Peng, "Automatic generation of software test cases based on improved genetic algorithm," 2011 International Conference on Multimedia Technology, pp. 227–230, 2011.

[15] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," IEEE Transactions on Software Engineering, vol. 44, no. 2, pp. 122–158, 2018.

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, 2002.

[17] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 276–291, 2013.

[18] A. Panichella, F. M. Kifetew, and P. Tonella, "Lips vs Mosa: A replicated empirical study on Automated Test Case Generation," Search Based Software Engineering: 9th International Symposium, Springer, pp. 83–98, 2017.

[19] S. Vogl et al., "Evosuite at the SBST 2021 Tool Competition," 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), pp. 28–29, 2021.

[20] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10, 2015.

[21] O. Sahin, B. Akay, and D. Karaboga, "Archive-based multi-criteria artificial bee colony algorithm for whole test suite generation," Engineering Science and Technology, an International Journal, vol. 24, no. 3, pp. 806–817, 2021.

[22] T. Y. Chen, G. Eddy, R. Merkel, and P. K. Wong, "Adaptive random testing through dynamic partitioning," Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings., pp. 79–86.

[23] D. Li et al., "Automatic test case generation using many-objective search and principal component analysis," IEEE Access, vol. 10, pp. 85518–85529, 2022.

[24] J. C. King, "Symbolic execution and program testing," Communications of the ACM, vol. 19, no. 7, pp. 385–394, 1976.

[25] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen, "On the automated generation of Program Test Data," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 293–300, 1976.

[26] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 297–306, 2008.

[27] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 360–369, 2013.

[28] D. Li, W. Eric. Wong, S. Li, and M. Chau, "Improving search-based test case generation with local search using adaptive simulated annealing and dynamic symbolic execution," 2022 9th International Conference on Dependable Systems and Their Applications (DSA), pp. 290–301, 2022.

[29] M. Modonato, "Combining Dynamic Symbolic Execution, Machine Learning and Search-Based Testing to Automatically Generate Test Cases for Classes," arXiv preprint arXiv:2005.09317, 2020.

[30] S. Driessen, D. Di Nucci, G. Monsieur, and W. J. Van Den Heuvel, "AGSolt: A tool for automated test-case generation for solidity smart contracts," arXiv preprint arXiv:2102.08864, 2021.

[31] M. Olsthoorn, D. Stallenberg, A. Van Deursen, and A. Panichella, "Syntest-solidity: Automated test case generation and fuzzing for smart contracts," 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 202–206, 2022.

[32] S. Shamshiri et al., "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T)," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 201–211, 2015.

[33] P. Zhang, J. Yu, and S. Ji, "ADF-GA: Data flow criterion based test case generation for Ethereum smart contracts," Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 754–761, 2020.

[34] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of Automated Unit Test Generation for Python," Empirical Software Engineering, vol. 28, no. 2, p. 36, 2023.

[35] A. Ramírez, J. R. Romero, and S. Ventura, "A survey of many-objective optimisation in search-based software engineering," Journal of Systems and Software, vol. 149, pp. 382–395, 2019.

[36] S. Ji, S. Zhu, P. Zhang, H. Dong, and J. Yu, "Test-case generation for data flow testing of smart contracts based on improved genetic algorithm," IEEE Transactions on Reliability, vol. 72, no. 1, pp. 358–371, 2023.

[37] B. M. S. Hasan and A. M. Abdulazeez, "A review of principal component analysis algorithm for dimensionality reduction," Journal of Soft Computing and Data Mining, vol. 2, no. 1, pp.20-30, 2021.

[38] C. Chen, L. Zhang, J. Bu, C. Wang, and W. Chen, "Constrained laplacian eigenmap for dimensionality reduction," Neurocomputing, vol. 73, no. 4–6, pp. 951–958, 2010.