Test Case Generation for Data Flow Testing of Smart Contracts Based on Improved Genetic Algorithm

Shunhui Ji, Shaoqing Zhu, Pengcheng Zhang, Hai Dong, and Jianan Yu

***Abstract*—Smart contracts are commonly deployed for safety-critical applications, the quality assurance of which has been a vital factor. Test cases are standard means to ensure the correctness of data flows in smart contracts. To more efficiently generate test cases with high coverage, we propose an improved Genetic Algorithm-based test case generation approach for smart contract data flow testing. Our approach introduces the theory of Particle Swarm Optimization into the Genetic Algorithm, which reduces the influence brought by the randomness of genetic operations and enhances its capability to find global optima. A set of 30 real smart contracts deployed on Ethereum and GitHub is collected to perform the experimental study, on which our approach is compared with three baseline approaches. The experimental results show that, in most cases, the coverage of the test cases generated by our approach is significantly higher than the baseline approaches with relatively lower numbers of iterations and lower execution time.**

***Index Terms*—Smart Contract; Data Flow Testing; Test Case Generation; Genetic Algorithm; Particle Swarm Optimization Algorithm**

I. Introduction

BLOCKCHAIN is widely used in various fields, such as banking, supply chain, and smart city, because of its decentralized, tamper-evident, and transparent traceability [1], [2]. A smart contract deployed on the Blockchain platform is a computer program that can automatically realize the content of the agreement expressed in natural language [3]. Currently, the most popular Blockchain platform is Ethereum, which is a cryptocurrency system that supports smart contracts developed with *Solidity* [4]. The inherent characteristics of the Blockchain make it impossible to modify the smart contract once it is deployed. If a vulnerable smart contract is deployed, it may lead to serious consequences. A representative case is the Dao incident occurred in 2016, in which hackers stole 3.6 million Ethereum tokens (worth 50 million U.S. dollars) by exploiting the reentrancy vulnerability [5]. Therefore, it is necessary for the publisher of a contract to ensure that the deployed contract is free of vulnerabilities and errors. This makes thorough testing of smart contracts essential, especially before the code is deployed [6].

S. Ji, S. Zhu, P. Zhang and J. Yu are with the Key Laboratory of Water Big Data Technology of Ministry of Water Resources and the College of Computer and Information, Hohai University, Nanjing 211100, China (e-mail: shunhuiji@hhu.edu.cn; 1139104058@qq.com; pchzhang@hhu.edu.cn; yu_poppy@qq.com).

H. Dong is with the School of Computing Technologies, RMIT University, Melbourne, Australia (e-mail: hai.dong@rmit.edu.au).

Corresponding author: Pengcheng Zhang.

Most research has been focused on the application of techniques, such as symbolic execution, abstract interpretation, and fuzz testing, for detecting vulnerabilities in smart contracts [7]. Some studies [5], [8], [9] also aim to detect the vulnerabilities through machine learning. An effective means to check the correctness of a program is to identify if the program can generate the expected output based on an input. The output generation is realized by a series of definitions and uses of variables accompanied with control flows. However, even for the strongest control flow testing criterion *all-paths*, it cannot guarantee that all errors will be detected by traversing all paths [10]. In addition, the *all-paths* criterion is not suitable for complex programs which may have an infinite number of paths. Compared with control flow testing, data flow testing is usually more effective, which could lead to more efficient and targeted test suites [11]. It examines not only the definition-use relationship of variables, but also the control flow in a program. In addition, the number of paths that fulfill the data flow testing criterion is always finite. With the prosperity of smart contract applications, it is a valuable and indispensable research direction to verify whether unknown programming errors exist in a smart contract with the data flow testing.

For the data flow testing, one of the major challenges is that the automatic test case generation needs to fulfil the data flow testing requirements. The random technique, which generates test cases at random, does not take the testing requirements into consideration [12]. The symbolic test case generation technique, which establishes and solves predicate equations to drive test case, may not be useful in practice because of the non-deterministic nature of the loop number and the explosion problem [13]. Evolutionary algorithms, such as Genetic Algorithm, which search for test cases to fulfill the testing criterion with the coverage analysis as guidance, have been demonstrated to be effective for test case generation in the data flow testing of traditional programs [14], [15]. A Genetic Algorithm-based test case generation approach, called ADF-GA (All-uses Data Flow criterion based test case generation using Genetic Algorithm), was proposed for data flow testing of smart contracts in our previous work [16]. However, the following problems remain:

1) Difficulties in achieving the overall high coverage of test cases. In the previous research, we used Genetic Algorithm in ADF-GA to perform test case generation. Although the test cases generated by ADF-GA can achieve higher coverage compared with RT [12] and GA-

C# [17], the overall quality of the generated test cases is still unsatisfactory with the average coverage rate of 77.3% [16].

2) Low performance of Genetic Algorithm-based test case generation. In a Genetic Algorithm, test cases with low quality may be generated due to the randomness of genetic operations. With the evolutionary process performed on top of these low-quality test cases, it would increase the number of iterations required to find the optimal solution and result in extra time costs.

To address the above two issues, we propose Iga-Sc (Improved Genetic Algorithm-based test case generation for Smart Contract) to generate test cases for data flow testing of smart contracts, by effectively integrating Genetic Algorithm and Particle Swarm Algorithm. The main workflow of Iga-Sc is described as follows: firstly, based on the structural and interactive features of a smart contract, a CFG (Control Flow Graph) of the contract is built from its *Solidity* source code; secondly, data flow analysis is performed based on the CFG to obtain the information of variables and the definition-use pairs to be tested; finally, we apply the improved Genetic Algorithm to generate test cases for the smart contract. The experimental results show that Iga-Sc generates test cases with high coverage in addition to ensuring its execution efficiency.

Compared with our previous research, the main contributions of this paper are as follows.

1) We introduce the principle of Particle Swarm Optimization to accelerate the Genetic Algorithm in finding the optimal solution to improve the performance of the approach. In addition, recombination of parent populations is adopted in the evolutionary process to improve the quality of test cases in the new population. It can significantly reduce the influence generated by the randomness of genetic operations and make it more effective to find the global optima.
2) We collect a dataset comprising 30 smart contracts with different sizes (i.e. less than 100 lines, 100-200 lines, and more than 200 lines) from GitHub and Ethereum to perform the experiments.
3) We perform an empirical comparison of Iga-Sc with ADF-GA, GA-C# (Genetic Algorithm based on the traditional fitness function [17]), and RT (random testing approach [12]). Iga-Sc can generate test cases with overall high coverage. Its average coverage rate is about 89.2% on the dataset, which exceeds that of the aforementioned methods by 13.81%, 19.26%, and 32.54% respectively. In addition, it is also capable of reducing the number of iterations required for test case generation. Compared with GA-C# and ADF-GA, the average number of iterations is reduced by 25.74% and 17.97% respectively.

The rest of the paper is organized as follows. Section II presents related research on smart contract testing and test case generation for data flow testing. Section III introduces the preliminary knowledge related to this research. Section IV illustrates the proposed approach Iga-Sc in detail. Section V performs the experimental study for evaluating Iga-Sc, while Section VI discusses the threats to validity. Finally, Section VII draws conclusions and identifies possible directions for our future work.

II. Related Work

This section provides an overview of smart contract testing and test case generation for data flow testing.

A. Testing for Smart Contracts

Most of the existing approaches and tools developed to detect common vulnerabilities in smart contracts are based on symbolic execution, abstract interpretation, and fuzz testing. Oyente [18], MAIAN [19], SmartCheck [20], and Slither [21] perform vulnerability detection in smart contracts based on symbolic execution. Chan et al. [22]–[24] designed a fuzz testing architecture for smart contracts and implemented a fuzzer called ContractFuzzer to inspect vulnerabilities and coding errors, in which test oracles are defined for different vulnerabilities. This tool is demonstrated to be effective to detect security vulnerabilities with high precision in a variety of real-world smart contracts. For smart contract vulnerability assessment, Liao et al. [5] used *Solidity* opcode as a machine learning feature to verify 13 kinds of vulnerabilities. A dynamic fuzz testing mechanism was created, which simulates the blockchain environment of online transaction verification, to analyze smart contracts and identify potential vulnerabilities. Unlike previous research work, it does not require pre-defined features, and achieves a vulnerability detection accuracy of 90%.

Wang et al. [25] proposed the notions of whole transaction basis path set and bounded transaction interaction to capture essential control flow behaviors of smart contracts. By means of the notions, they tested the smart contracts with the k-bounded transaction coverage criteria. This approach is compared with random testing and statement coverage testing to demonstrate its effectiveness. Wu et al. [26], [27] proposed a framework for mutation testing of smart contracts with fifteen novel mutation operators. The experimental results show that this approach outperforms the coverage-based approaches on defect detection. A mutation testing tool, called MuSC, is built based on this framework. Wang et al. [28] defined the test generation problem of smart contracts as a Pareto minimization problem. They proposed a multi-objective approach based on random approach and NSGA-II to generate cost-effective test suites while retaining the capability of branch covering.

Kim et al. [29] proposed an automated test suite generation method to address the absence of reliability testing of smart contract analyzers. It can diversify test cases by combining vulnerabilities and changing code complexity. Defects and false positives of existing smart contract analyzers can be discovered with the test suites generated by this method. Smart contracts on enterprise permission Blockchain are usually more complicated. Liu et al. [30] proposed a model-based testing platform, called MODCON, for enterprise smart contracts. It relies on user-specified models to impose model testing and efficiently generate test cases for smart contracts. Driessen et al. [31] developed an automated generator AGSoLT to generate test suites for *Solidity*-based smart contracts. They implemented the random search algorithm and the genetic algorithm DynaMOSA [32] to evaluate the efficiency of the generator. Through the experiments with 36 real-world smart contracts, both the approaches demonstrate their capability in achieving high branch coverage and finding certain errors.

B. Test Case Generation for Data Flow Testing

There has been a long history of research on test case generation for data flow testing. Rapps et al. [10] extended and defined the concept of data flow analysis and proposed a series of data flow testing criteria. These criteria focus on the definition-use association of variables, covering not only the data flow but also the control flow in a program. Frankl et al. [33] extended the previous data flow testing criteria by defining a new family of adequacy criteria, called feasible data flow testing criteria. It only generates test cases for those executable definition-use associations to circumvent the problem of inapplicability of data flow testing criteria.

Many studies have explored the use of optimization algorithms to generate test cases for data flow testing. Ghiduk et al. [11] proposed a test case generation method using Genetic Algorithm to meet the data flow testing criteria. This method bases on the concept of inter-node dominance relationships to define a new fitness function. The empirical studies show that this method can achieve coverage of the test requirements and reduce the size of test suites. Rajkumari et al. [14] proposed a new approach for automatic test case generation. They employed Genetic Algorithm guided by the program dependency graph of the target software. The evaluation demonstrates that the use of program dependency graph and Genetic Algorithm to generate test cases outperforms random testing. Sheoran et al. [34] used an Artificial Bee Colony algorithm to perform local and global searches for the extraction of data flow testing paths. This algorithm tracks all definition-use paths by a global search, and identifies definition-use paths which are not definition-clear paths by a local search. In addition, these paths are prioritized, which benefit optimal test suite generation and time reduction. Nayak et al. [15] generated test cases for data flow testing using Particle Swarm Optimization algorithm that generates a set of test cases and a set of definition-use paths covered by each test case. The experiment demonstrates that this approach outperforms Genetic Algorithm in generating test cases for data flow testing in terms of coverage.

Some work combined Particle Swarm Optimization algorithm and Genetic Algorithm to perform test case generation. Singla et al. [35] proposed GPSCA to automatically generate test data for data flow by integrating Genetic Algorithm and Particle Swarm Optimization Algorithm, and a new multi-objective fitness function. Experiments show that the proposed GPSCA is effective in achieving coverage and reducing the number of test cases compared with Particle Swarm Optimization algorithm and Genetic Algorithm. Kumar et al. [36] proposed a hybrid APSO-GA algorithm for test suite generation for data flow testing, in which a fitness function based on branch weight and branch distance is designed. Compared with Genetic Algorithm, Particle Swarm Optimization algorithm and hybrid GA-PSO, APSO-GA performs better in terms of coverage.

Despite the substantial progress made on test case generation for data flow testing, most of the research only focuses on traditional programming languages, such as *Java* and *C*. Zhang et al. [16] proposed ADF-GA, the first approach for generating test cases based on data flow criteria for *Solidity*-based smart contracts, which bases on Genetic Algorithm to perform test case generation. ADF-GA was compared with two traditional approaches. The results showed that ADF-GA can generate usable test cases more efficiently. However, considering the unsatisfactory coverage of the test cases generated by ADF-GA, further study is still required in this area.

III. Preliminaries

This section introduces the prerequisite knowledge of smart contract, data flow testing, Genetic Algorithm and Particle Swarm Optimization algorithm.

A. Smart Contract

```
uint16 dnaDigits = 16;
uint16 dnaModulus = 10 ** dnaDigits;
struct Zombie {
   uint16 id;
   uint16 dna;
 }
Zombie[] public zombies;
function _createZombie(uint16 _id, uint16 _dna) private {
    zombies.push(Zombie(_id,_dna));
}
function _generateRandomDna(uint16 _id) private view returns (uint16) {
   uint16 rand = uint16(keccak256(abi.encodePacked(_id)));
   return rand % dnaModulus;
}
function _createRandomZombie(uint16 _id) public {
   uint16 randDna = _generateRandomDna(_id);
   require(randDna >= 10 && randDna <= 10000);
   _createZombie(_id, randDna);
}
```

Fig. 1. Code Segment of a Smart Contract

A smart contract is a set of promises defined in digital form, including an agreement on which contract participants can execute these promises. In other words, smart contracts are digital versions of traditional contracts, deployed and executed on the Blockchain platform [29]. With the Blockchain technology, contract enforcement can not only take advantage of the cost-efficiency of smart contracts, but also prevent malicious behavior from the proper execution of the contract. Users trust smart contracts, since they specify requirements of transactions and guarantee users' rights within disputes, instead of third-party institutions and participants. In order to facilitate the construction of smart contracts on the Ethereum platform, a high-level programming language *Solidity* was specially created [37]. Smart contracts on the Ethereum platform, which are written in *Solidity*, are the objects of our study.

From the perspective of test case generation for data flow testing, the features of *Solidity*-based smart contracts can be summarized in the following three aspects.

Structural features. Generally, *Solidity* has three logical structures like traditional programming languages: sequential structure, selection structure, and loop structure [38]. Keywords, such as *if-else*, *while*, *do* and *for*, also exist in *Solidity*, which express the same semantics with that in traditional programming languages. In addition, *Solidity* has its own unique selection structure statement: *require* statement. It is used to control the program execution. The program will continue to execute only when the condition in a *require* statement is satisfied.

Variable features. The most commonly used variables in smart contracts are of numeric type, which can be used to represent Blockchain-oriented information, such as address, timestamp, etc. Numerical variables in *Solidity* are mainly divided into two types: unsigned integer and signed integer, of which the type identifiers are *uint* and *int* respectively. In addition, the length of a variable, which determines the range that the variable can represent, is directly defined with the keyword. The minimum length of a variable is 8 bits and the maximum length is 256 bits. For example, a variable declared with keyword *uint8* can represent a natural number between 0 and (2^8-1); a variable declared with *int16* can represent an integer between (-2^{15}) and (2^{15}-1). The direct use of the keyword *uint* indicates the declaration of a variable with length 256 by default.

Interactive features. Similar to the instantiation and function call of *Java* class, smart contracts can be instantiated and function calls can be performed in the program. Fig. 1 shows an example of smart contract containing function calls, where the function *_createRandomZombie()* calls the functions *_generateRandomDna()* and *_createZombie()*.

B. Data Flow Testing

Data flow testing focuses on the associations between the definition of each variable and its uses in a program [10]. Test cases are designed to trace the data flow from the input variables to the produced output values. It can be concluded that the computations are correctly performed only if the result of each computation has been used. Rapps and Weyuker [10] defined a family of data flow testing criteria, in which the *all-uses* criterion has been demonstrated to be practical and effective for the testing of *C++* programs [11], composite services [39], and so on. Therefore, the *all-uses* criterion, which requires test cases to exercise at least one path connecting from each definition to each use reached by that definition, is selected for guiding the test case generation of smart contracts in our project.

Data flow analysis is a prerequisite task in the data flow testing, which is used to extract the associations between a variable's definition and its uses. It is usually performed based on the CFG of a program, in which a node represents a statement in the program and an edge represents the execution sequence of two statements. Suppose *def(n)* and *use(n)* respectively denote the set of variables defined and used in the statement corresponding to node *n*. For variable *x*, if $x \in$ *def(n)* and $x \in$ *use(n')*, and if there is a path (n, n_1, $n_2 \ldots n_m$, *n'*) from *n* to *n'* containing no definition of *x* in nodes n_1, n_2, $\ldots$, n_m, there is a definition-use relationship of *x* between *n* and *n'*, which can be constructed as a definition-use (def-use) pair (*x*, *n*, *n'*). All the def-use pairs of the program need to be extracted as as the prerequisite in the *all-uses* data flow testing.

C. Genetic Algorithm

Genetic Algorithm [11] (GA) is a search algorithm which simulates the genetic and evolution process of organisms in the natural environment. It was first proposed by Professor Hollan and originated from a study of natural and artificial adaptive systems [40]. Similar to the biological evolution process, GA sets a target during the searching process in which the population evolves to find the optimal solution to satisfy this target. In different application scenarios, different encoding approaches are used to encode the individuals of the population so that they can be manipulated by computer programs. GA iteratively updates the population to obtain the final result. In each iteration of updating population, the individuals are evaluated with the defined fitness function to determine which individual performs better for guiding the evolutionary process [17]. Better individuals are then selected for future generations. Crossover and mutation are performed on the individuals to generate a new population. The evolution process stops only if the optimal solution is found or the maximum number of iterations is achieved.

D. Particle Swarm Optimization Algorithm

Particle Swarm Optimization (PSO) algorithm is motivated by simulating the social behavior of animals and the methods by which they find roosting places or food sources [15]. Based on this principle of biological group behavior, PSO algorithm searches the optimal position of each particle of the swarm using its own experience and the experience of other particles [41].

Firstly a population is created by randomly generating n particles in the d-dimensional problem space with each particle represented as a d-dimensional vector. Every particle moves to search the optimal position by updating the velocity and position. During this process, each particle preserves its achieved best position, which is called personal best *pbest*. Meanwhile, the best position among all the particles' *pbest*s is called global best *gbest*. For the current population, it calculates the fitness of each particle based on the value of the d-dimensional parameter and the fitness function, and compares the current fitness of each particle with its corresponding *pbest*. If the current fitness of the *i*-th particle p_i is better than that of $pbest_i$, it updates the position of $pbest_i$ with the current value of p_i. Then *pbest*s of all the *n* particles are compared with *gbest* to determine whether *gbest* needs to be updated [42].

Specific termination conditions are defined for the algorithm according to the actual problem to be solved [43]. When the predefined conditions are met, *gbest* is returned, which is the optimal solution of the optimization problem.

IV. The Iga-Sc Approach

This section specifically describes the improved test case generation approach Iga-Sc for smart contract data flow testing. Fig. 2 shows the overview of Iga-Sc, in which OFV_{CurPop} and $OFV_{PrevPop}$ respectively represent the optimal fitness value of the current population and the previous population. This process consists of three main phases: CFG construction, data flow analysis, and test case generation. In the CFG construction phase, it is implemented at the source code level based on the structural and interactive features of smart contracts. In the data flow analysis phase, the definition-use relationships of variables are extracted by traversing the generated CFG and the *require* statements are identified. In the test case generation phase, the contract program is instrumented according to the test target, and the improved GA is applied to iteratively search the optimal test cases.

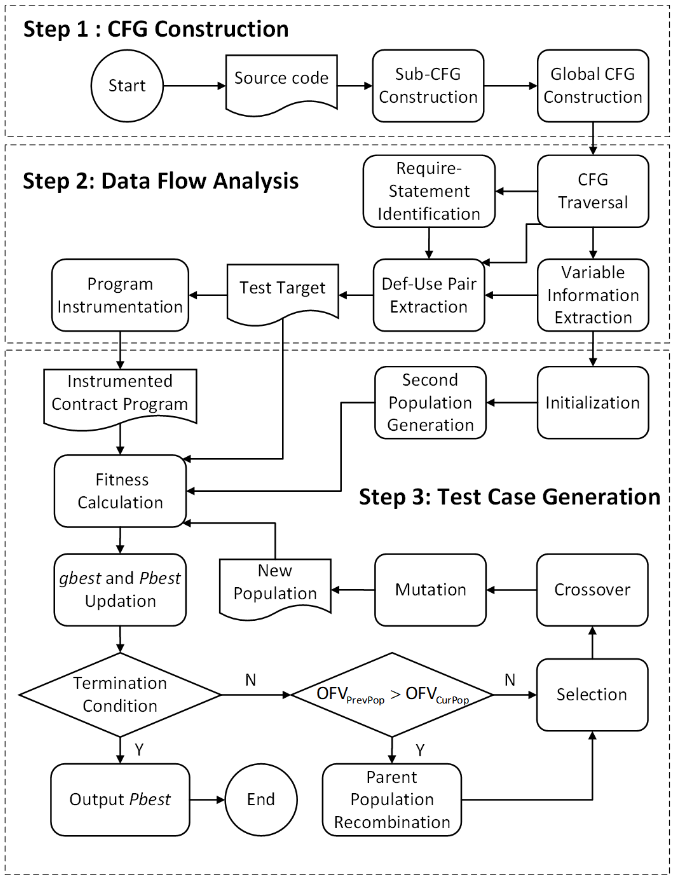


Fig. 2. Overview of Our Approach

A. Control Flow Graph Construction

To automatically implement test case generation for data flow testing, it is necessary to construct CFG [3] for a smart contract to extract the valid information. The control flow graph is more intuitive in describing the execution process of a smart contract. Therefore, it can be more comprehensible in contrast to the program. Although a data flow graph can directly reflect the data flow information, its construction requires control flow analysis. In addition, the data flow testing criterion used in our approach is defined upon the control flow graph. Therefore, we use the control flow graph other than the data flow graph. CFG construction is implemented at the source code level based on the structural and interactive features of a smart contract to retain the necessary semantic information and handle the inter-functional calls in the smart contract. The constructed CFG contains the following features.

1) Similar to the CFG structure of traditional programs, CFG of a smart contract contains sequential structure, selection structure, and loop structure [21].

2) Each node in the graph represents one valid statement or the beginning and end of a function in the smart contract.

3) The execution order of contract statements is represented by a directed edge connecting between two nodes. An arbitrary directed edge in the CFG is represented by $\langle x, y \rangle$, which means the statement corresponding to node *y* must be executed after completion of the statement corresponding to node *x*.

4) Each function is converted to a sub-CFG, and then a global CFG is constructed according to the function call relationship in the smart contract.

Sub-CFG Construction. For each function in the contract, which is declared by the keyword *function*, or a function modifier which is declared by the keyword *modifier*, a separate CFG is firstly constructed as a sub-CFG. During the construction, the *require* statement in the contract is regarded as a selection structure. Two edges are created for it, one of which points to the node of the next statement for indicating the situation that the condition specified in the *require* statement is true, and the other points to the end node of the function for indicating it exits execution when the condition is false.

Global CFG Construction. The inter-function call relationship in the contract is handled by generating two edges representing the call and return relations between two functions. Firstly, the edge of function call is created, which starts at the corresponding call node in the sub-CFG of the function and points to the start node of the sub-CFG of the function being called. Then, the edge of function return is created, which starts at the terminal node of the sub-CFG of the function being called and points to the corresponding call node.

The CFG corresponding to the sample smart contract in Fig. 1 is shown in Fig. 3.

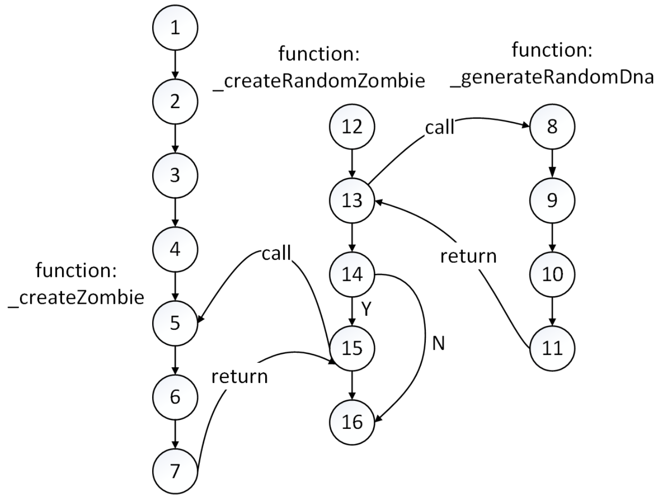


Fig. 3. The CFG corresponding to the sample smart contract

B. Data Flow Analysis

Data flow testing aims at covering all the def-use pairs of a program during the testing. Before performing data flow

TABLE I
The Parametric Variables of the Sample Smart Contract

Variable Name	Variable Type	Variable Length
_id	0	16
_dna	0	16

TABLE II
The General Variables of the Sample Smart Contract

Variable Name	Variable Type	Variable Length
dnaDigits	0	16
dnaModulus	0	16
rand	0	16
randDna	0	16

testing, data flow analysis is required to extract the def-use pairs of a smart contract, which will be located as the coverage target for test case generation. Based on the CFG of the smart contract constructed in Section IV-A, the process of data flow analysis is executed in the following three steps.

1) Extraction of Variable Information: The main purpose is to obtain the information of variables in a smart contract, which includes variable name, variable type, and variable length. The variables are distinguished into two types: parametric variables and general variables. A parametric variable is the one used to execute the smart contract, which mainly appears as the input parameter of the function in the contract. A general variable is declared in the contract through a variable declaration statement.

Extraction of parametric and general variable information is separately performed as follows.

Extraction of Parametric Variable Information. We traverse the CFG of a smart contract and analyze the information of each node in the CFG to determine whether it corresponds the function declaration statement. The variable information, including variable name, variable type, and variable length, of the input parameters of the functions are extracted and stored in the set V_p. For a variable declared with *uint* and *int*, the variable type is denoted with 0 and 1 separately. The variable-length is specified with the variable type identifier. The parametric variable information extracted from the smart contract code segment in Fig. 1 is shown in TABLE I.

Extraction of General Variable Information. We traverse the CFG of a smart contract and analyze each node in the CFG to determine whether it corresponds to the variable declaration statement. Information of general variables is extracted and stored in the set V_g. The general variable information extracted from the contract code segment in Fig. 1 is shown in TABLE II.

With V_p and V_g, the information of all the variables can be obtained and stored in L_v.

2) Identification of Require Statements: A *Solidity*-based smart contract has structural statements to indicate the transfer of control flow, including *for*, *switch*, *while*, and *if-else*, which are similar to traditional languages, such as *Java*. In addition, it contains a unique type of statement declared by the *require* keyword, which is used to determine whether the execution conditions are met. When the contract program reaches a *require* statement, it continues the execution only if the condition in the *require* statement is true. The *require* statement is commonly used to restrict the status of critical variables, such as account status and quantity relationships. Therefore, the def-use pairs associated with the *require* statement are computed separately to emphasize them under the testing coverage. To compute the def-use pairs associated with the *require* statement, the *require* statement requires to be identified firstly. By traversing the CFG of the smart contract, each CFG node is analyzed with the recorded information to determine whether it corresponds to a *require* statement. The *require* statement in the smart contract code segment in Fig. 1 is shown in TABLE III.

TABLE III
The Require Statement of the Sample Smart Contract

Require Statement	Node Number
require(randDna >= 10 && randDna <= 10000)	14

3) Extraction of Def-Use Pairs: The def-use pairs contained in a smart contract are collected in two ways [16]. *N_dup* is used to store all the def-use pairs in the contract program. *R_dup* is used to store the def-use pairs related to the *require* statement. A def-use pair is defined as (*v*, *def*, *use*), where *v* denotes a variable, *def* denotes a definition of variable *v*, and *use* denotes the use of the value of *v* defined in *def*. To obtain the test target *N_dup* and *R_dup*, two lists L_{v-d} and L_{v-u} are respectively used to store the definition nodes and use nodes of each variable in L_v. The extraction of *N_dup* and *R_dup* is performed as follows.

Extraction of *N_dup*. The CFG of the smart contract is traversed from the initial node to process each path in the graph in a depth-first order. During the process of traversing each path in the CFG, def-use pairs in the path are extracted. *N_dup* can be obtained after all paths have been processed.

For the currently traversed path, each visited node is analyzed whether it defines or uses some variables. If the currently visited node defines variable *var1*, the node is added to the list L_{v-d} of *var1*. If the currently visited node *u* uses variable *var1*, node *u* is added to the list L_{v-u} of *var1*, and the list L_{v-d} of *var1* is traversed to find the definition node *d* closest to node *u* in the current path. Then, a def-use pair (*var1*, *d*, *u*) is constructed and added to the test set *N_dups*.

Extraction of *R_dup*. Based on the identification results of the *require* statement, further extraction of *R_dup* is completed by filtering def-use pairs in *N_dup*. *R_dup* includes the following two types of def-use pairs:

- For a def-use pair (*x*, *d*, *u*), if the use node *u* corresponds to a *require* statement, the def-use pair belongs to *R_dup*.
- For a def-use pair (*x*, *d*, *u*), if the definition node *d* or the use node *u* control depends on the *require* statement, the def-use pair belongs to *R_dup*.

With *N_dup* and *R_dup*, the test target of the smart contract that satisfies the data flow testing criterion can be obtained. The def-use pairs extracted in the smart contract code segment in Fig. 1 are shown in TABLE IV.

TABLE IV
The Def-Use Pairs of the Sample Smart Contract

Variable Name	Def_node	Use_node	N_dup	R_dup
dnaDigits	1	2	(dnaDigits,1,2)	—
dnaModulus	2	10	(dnaModulus,2,10)	—
_dna	5	6	(_dna,5,6)	—
_id	5,8,12	6,9,13,15	(_id,5,6) (_id,8,9) (_id,12,13) (_id,12,15)	(_id,12,15)
rand	9	10	(rand,9,10)	—
randDna	13	14,15	(randDna,13,14) (randDna,13,15)	(randDna,13,14) (randDna,13,15)

C. Test Case Generation

For covering the untested def-use pairs of the smart contract, an improved GA is employed to generate the test cases. The fitness function defined in our previous work [16] is shown in Equation (1), which is applied to evaluate the closeness of the generated test case *tc* to the test target and guide the optimal test case search.

$$fit(tc) = \frac{(n-m)+(1+\varepsilon)m}{s} \tag{1}$$

where *s* denotes the number of *N_dup* that need to be tested, *n* denotes the number of *N_dup* covered by the current test case *tc*, *m* denotes the number of *R_dup* covered by *tc*, and ε is the weighting parameter of *R_dup*, whose exact value is determined by experiments. The *require* statement is unique to the *Solidity*-based smart contract, which is widely used to restrict the execution of the main functionality and avoid vulnerabilities. If the execution condition of *require* statement is not met, the coverage of the definition-use pairs is very limited. Therefore, extra weighting is added on *R_dup* to favor the *require* statement related pairs, which will lead to a high coverage of the definition-use pairs of the smart contract. With this fitness function, low coverage of def-use pairs related to the require statements can be addressed in contrast to the traditional fitness function [17].

In our previous work [16], a GA is applied to generate test cases iteratively. It selects the better individuals of the current population to form the parent population and performs genetic operations to generate the new population. Due to the randomness of the genetic operations of crossover and mutation, individuals with low coverage may be generated in the new population. The further optimization based on such a population would end with a set of test cases with low coverage or take excessive time to find the optimal test cases. In addition, ADF-GA only guarantees high coverage of an individual other than that of the whole population, since it uses the maximum fitness of the individual as the termination condition of the GA.

To solve the above problems, Iga-Sc incorporates the principle of the PSO algorithm into the GA to improve the test case generation.

During an algorithm iteration, *pbest* of each individual is stored in the set *PB*. In the PSO algorithm, *pbest* holds the optimal position for each particle in the population. Iga-Sc treats each individual in the GA as a particle in the PSO and considers genetic operators-based evolution of each individual as position updating of a particle. During the iterative process of searching the optimal test cases, *PB* preserves the optimal fitness of each individual by comparing the fitness of PB_i and the *i*-th individual. And *gb* is set to record *gbest* which is the optimal test case during an algorithm iteration.

Suppose P_j represents the population of the *j*-th generation and P_{ji} represents the *i*-th individual in P_j. As shown in Fig. 4, PB_i records the optimal fitness of the *i*-th individual. When updating P_{ij} to P_{ij+1}, the fitness of P_{ij+1} and PB_i is compared to decide whether PB_i needs to be updated.

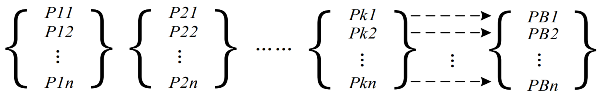


Fig. 4. Population's individuals and PB

Algorithm 1 TcGeneration( )

```
Input: Instrumented smart contract;
       Def-use pairs to be covered;
       Parametric variable information;
       Parameters of the genetic algorithm.
Output: Test cases Pbest.
 1: Initialize population PrevPop;
 2: Calculate the fitness of each individual in PrevPop;
 3: Initialize Pbest and gbest;
 4: Perform genetic operators on PrevPop to generate the second generation population CurPop;
 5: Calculate the fitness of each individual in CurPop;
 6: Update Pbest and gbest;
 7: while the termination condition is not met do
 8:     if $OFV_{PrevPop} > OFV_{CurPop}$ then
 9:         Compare and determine $CurPop_i$ that has the minimum fitness in CurPop;
10:         $CurPop_i \leftarrow gbest$;
11:     end if
12:     Perform selection, crossover and mutation on CurPop to generate new population NewPop;
13:     $PrevPop \leftarrow CurPop$;
14:     $CurPop \leftarrow NewPop$;
15:     Calculate the fitness of each individual in CurPop;
16:     if $fit(CurPop_i) > fit(Pbest_i)$ then
17:         $Pbest_i \leftarrow CurPop_i$;
18:     end if
19:     Compare and determine $Pbest_k$ that has the maximum fitness in Pbest;
20:     if $fit(Pbest_k) > fit(gbest)$ then
21:         $gbest \leftarrow Pbest_k$;
22:     end if
23: end while
24: return Pbest;
```

The process of test case generation is shown in Algorithm 1. **Line 1:** Firstly, an initial population *PrevPop* is generated randomly, in which each individual represents a test case. Supposing the population size is *m*, it is achieved by iterating *m* times and initializing one individual each time. For each individual *PrevPop*$_i$, the initialization is based on the extracted parametric variable information, including variable type and variable length. Since the types of variables in *Solidity*-based smart contracts are mainly numeric, specifically *uint* and *int*, binary encoding is sufficient to satisfy the requirements for variable precision. By randomly generating a valid value within the range for each parametric variable, the value of each parametric variable is encoded into a binary string as a sub-chromosome [16]. For a test case, which is a collection of variable values, it is represented as a series of sub-chromosomes in the test case generation algorithm. The test cases are then decoded into actual values to execute the smart contract.

For each sub-chromosome, the length of the binary string is the length of the variable plus 1, where the first bit of the string indicates the type of the variable (0 for *uint* and 1 for *int*). The remaining bits indicate the binary form of the variable value. Using the encoding method, a variable of type *uint8* with a value of 108 can be encoded as shown in Fig. 5.

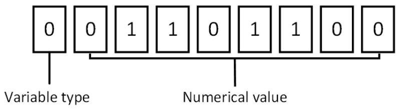


Fig. 5. Example of Sub-Chromosome Encoding

Line 2-3: Meanwhile, *Pbest* and *gbest* are initialized based on the initial population *PrevPop*. Each element *Pbest*$_i$ is initialized with the individual *PrevPop*$_i$. The *gbest* is initialized with the individual that achieves the maximum fitness in *PrevPop*. With the fitness function defined in Equation (1), the fitness of a test case is calculated by executing the instrumented smart contract with decoded individuals as the input. The fitness values of individuals in *PrevPop* are compared to decide *gbest*. **Line 4:** Then the second generation population *CurPop* is generated by performing selection, crossover, and mutation on *PrevPop*. **Line 5-6:** By calculating the fitness of each individual in *CurPop*, the fitness values of the individuals between *CurPop* and *Pbest* are compared to update *Pbest* and *gbest*.

Line 8-11: The iteration starts with determining whether parent population requires recombination before genetic operations being performed. It compares the optimal fitness of the current population OFV_{CurPop} with that of the previous population $OFV_{PrevPop}$. If $OFV_{PrevPop}$ is greater than OFV_{CurPop}, the parent population *CurPop* is recombined by replacing the individual with the lowest fitness in *CurPop* with *gbest*. **Line 12:** Then selection, crossover and mutation operations are performed on *CurPop* to generate a new population *NewPop*, followed by *PrevPop* and *CurPop* being updated.

For the genetic operators, a selection is performed based on the fitness of each individual to form a parent population. The higher the fitness of an individual, the higher the probability of being selected. It not only makes the better individuals have greater probability of being retained, but also ensures the diversity of the population.

Crossover is performed with the sub-chromosome as the basic unit to enable the renewal of individuals. Each gene on paired sub-chromosomes is exchanged with an equal probability. Supposing that two paired sub-chromosomes are denoted as "X = x_1, $x_2 \ldots x_m$" and "Y = y_1, $y_2 \ldots y_m$", a binary string S of length m is randomly generated, denoted as "S = s_1, $s_2 \ldots s_m$". As introduced in the encoding rules, the first bit of the binary string indicates the variable type. Therefore, the crossover is performed between X and Y starting from the second bit to the m-th bit. Whether to exchange information between x_i and y_i depends on the value of S_i. If the value of S_i is 1, the values of x_i and y_i are exchanged. Otherwise, the original values of x_i and y_i are retained.

Mutation is performed on bits of sub-chromosomes to further enable the renewal of individuals. With the preset mutation probability P_m, each bit of the sub-chromosome is mutated from 0 to 1 or 1 to 0. A random r between 0 and 1 is generated to determine whether the bit is mutated or not. If $r < P_m$, the mutation is performed.

Line 13-14: Then, *PrevPop* is updated to the current population *CurPop*. The new population *NewPop* generated with the genetic operators is taken as the current population *CurPop*. **Line 15-22:** The fitness for each individual of *CurPop* is calculated and *Pbest* and *gbest* are updated. If the fitness of *CurPop*$_i$ is greater than that of *Pbest*$_i$, *Pbest*$_i$ is updated with *CurPop*$_i$. After *Pbest* being updated, *gbest* is compared with the individual *Pbest*$_k$ that has the maximum fitness in *Pbest*. If the fitness of *Pbest*$_k$ is greater than that of *gbest*, *gbest* is updated with *Pbest*$_k$. The iteration continues until the termination condition is satisfied, which is based on the status of *Pbest* and *gbest*. If *gbest* reaches a predefined threshold and *gbest* is not updated compared to the previous iteration, each individual of *Pbest* has the same fitness as *gbest*; otherwise if the maximum number of iterations is reached, *Pbest* is produced as the test cases and the algorithm terminates.

In general, compared to ADF-GA, the improvement of Iga-Sc specifically includes the following aspects.

1) We apply *Pbest* to preserve the set of optimal test cases generated during the GA execution, which ensures that each test case in the output set has high coverage of the def-use pairs to be tested.

2) We apply *gbest* to assist the recombination of a parent population prior to the selection operation, which ensures high fitness of the parent population.

3) *Pbest* and *gbest* are updated constantly during the iteration, statuses of which are used to determine whether the algorithm can be terminated. It helps to reduce useless iterations and additional time consumption.

V. Experimental Evaluation

This section describes the experimental study we conducted to show the effectiveness and efficiency of our approach Iga-Sc by comparing it with the other three test case generation approaches, including ADF-GA, RT, and GA-C#.

A. Experimental Setup

The experimental setup includes the following three aspects: experimental environment, experimental dataset, and evaluation indicators.

TABLE V
Experimental Hardware and Software Environment

Name	Standard
Operating system	Windows10
CPU	Intel (R) Core (TM) i5-8300H
Memory	8GB
Development tool	Eclipse
Programming Language	Matlab, Java
Smart Contract Execution Environment	Remix

TABLE VI
Information about the Dataset of Smart Contracts

SC Name	LOC	Number of Function	Number of N_dup	Number of R_dup
safeadd	22	1	10	4
basictoken	24	3	6	4
fund	39	3	16	14
getcenter	47	1	28	0
election	50	3	7	1
getsum	55	1	23	0
zuniswap	56	4	13	2
aztcharairdr	59	3	5	2
smdigicion	59	5	14	1
eip20token	59	5	14	12
lotterywin	62	6	8	4
monmail	65	5	5	1
greeter	65	4	8	0
mathop	84	7	102	0
safebuy	88	5	29	12
lottery10	110	4	42	4
idemana	114	15	50	16
fundraise	123	4	33	14
mathopreq	126	7	108	22
erc20	126	8	18	0
safemath	150	8	36	25
multiwallet	151	13	23	22
operofarr	155	11	62	25
rubixi	155	19	62	0
infomanasys	163	5	74	30
therun	177	17	58	0
erctoken	185	10	38	28
trade	202	10	64	30
geometry	246	9	93	50
timelibrary	336	39	694	143

1) Experimental Environment: The hardware and software environment of the experimental study is shown in TABLE V. The extraction of variable information and def-use pairs in a given smart contract is implemented with *Java* programming language on the platform of Eclipse. The improved GA based test case generation is implemented based on MATLAB. Smart contracts are deployed and run on the online integrated development environment of Remix[1].

2) Experimental Dataset: The following three main factors are considered when constructing the experimental dataset:

(i) The diversity and universality of smart contracts,

(ii) The independence of smart contracts so that each can be executed without calling other contracts, and

(iii) Whether the selected dataset covers the common statements and structures of *Solidity*.

Based on these factors, we choose 30 typical smart contract programs as the experimental dataset. These contract programs contain commonly used statements and structures of *Solidity*, and cover business logic in different scenarios. The number of code lines, functions and def-use pairs of the chosen smart contracts are shown in TABLE VI. The first column refers to the names of the smart contracts. The second column refers to the number of valid code lines of the smart contracts. Since the smart contracts developed and run on the Ethereum platform are mostly small-scale, the number of smart contracts with a high volume of code lines is small. Smart contracts of different scales, including less than 100 lines of code, 100-200 lines of code, and more than 200 lines of code, are selected as the experimental objects. Iga-Sc is applicable to smart contracts that contain only a single function or multiple functions and function calls. The smart contracts used in the experimental study cover both the cases. The third column refers to the number of functions contained in the smart contracts, where 1 indicates that there is no function call in a contract. The fourth and fifth columns respectively refer to the numbers of *N_dup* and *R_dup* in the smart contracts. If *require* statement is not used in a smart contract, the number of *R_dup* is 0.

[1]http://remix.hubwiz.com/

TABLE VII
Experimental Parameters

Parameter Name	Parameter Value
N	4
P_m	0.1
$Max\_iteration$	100

The above dataset is collected from the typical smart contract test datasets publicly available on GitHub and real smart contracts deployed on the Ethereum platform. To ensure the executability of the smart contracts, they are edited and improved, followed by being deployed, compiled and run on Remix before the experimental evaluation.

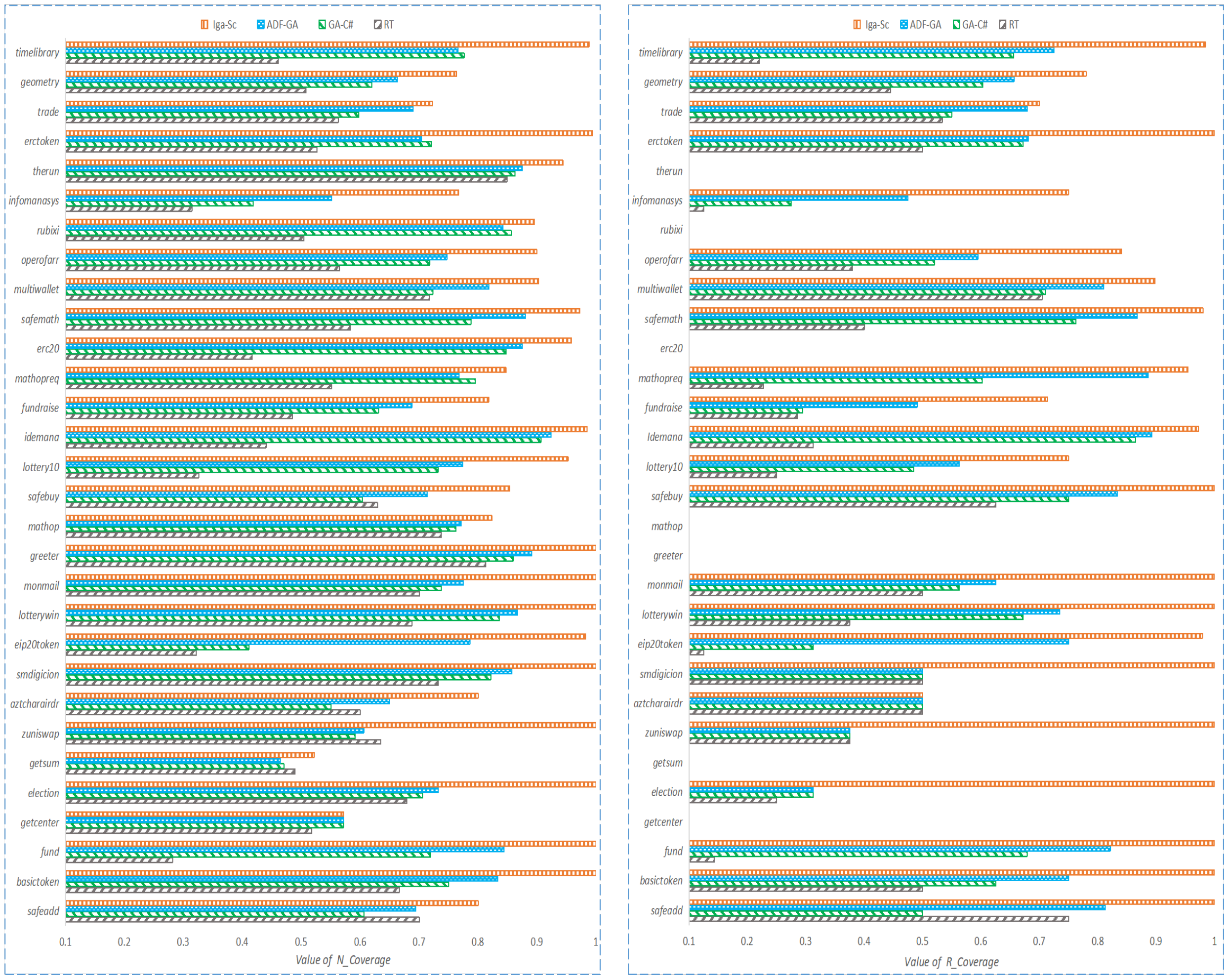
3) Experimental parameters: The parameters of the GA in the experimental study, including the number of population N (i.e. the number of test cases), the mutation probability P_m, and the maximum number of iterations $Max\_iteration$, are shown in TABLE VII. In addition, the parameter ε in the fitness function is set as 0.45, which is determined by the experimental study in [16].

4) Evaluation Indicators: To evaluate the effectiveness of an approach, the generated test cases will be measured in terms of coverage and fitness. To evaluate the efficiency of an approach, the number of iterations of the algorithm and the execution time are used as indicators. The evaluation indicators are described in detail as follows.

Coverage. *Coverage* is used to evaluate the extent to which the generated test cases cover the test targets. It can be calculated as follows:

$$Coverage = \frac{n_{cov}}{n_{sum}} \tag{2}$$

where n_{cov} indicates the number of def-use pairs actually covered by the test cases and n_{sum} indicates the total number of test targets. The higher the coverage rate, the more effective an approach is. Since the test targets are divided into *N_dup* and *R_dup*, the indicator *coverage* is divided into *N_Coverage* and *R_Coverage*, whose corresponding n_{sum} is the numbers of *N_dup* and *R_dup* contained in a smart contract.



(a) Statistical Results of N_Coverage (b) Statistical Results of R_Coverage

Fig. 6. N_Coverage and R_Coverage of Different Approaches

Fitness. *Fitness* is used to measure how well the generated test cases fit a test case generation algorithm, which is based on the specific fitness function used in an approach. When performing comparison with other approaches, it is necessary to use the same fitness function in different approaches to make the experimental results comparable.

Iteration. *Iteration* represents the number of iterations of an optimization algorithm required for generating test cases, which is a commonly used indicator for evaluating the performance of an approach. With the same targets , the lower *Iteration* is, the better performance the approach has.

Execution Time. *Execution Time* is an another straightforward indicator for evaluating the performance of an approach. Although the average time consumption per iteration of GA is tiny, it cumulatively makes significant time consumption when the algorithm runs a large number of iterations. Therefore, *Execution Time* and *Iteration* are combined to evaluate the efficiency of an approach.

5) Comparative Approaches: The proposed approach Iga-Sc is compared with the following existing approaches for test case generation:

- **ADF-GA.** ADF-GA is an approach using GA to generate test cases of smart contracts based on the improved fitness function [16].
- **RT.** RT is an approach that randomly generates test cases [12].
- **GA-C#.** GA-C# is originally designed for *C#* [17]. It uses GA for the automatic test case generation based on the traditional fitness function, as shown in Equation (3):

$$fit(tc) = \frac{m}{n} \tag{3}$$

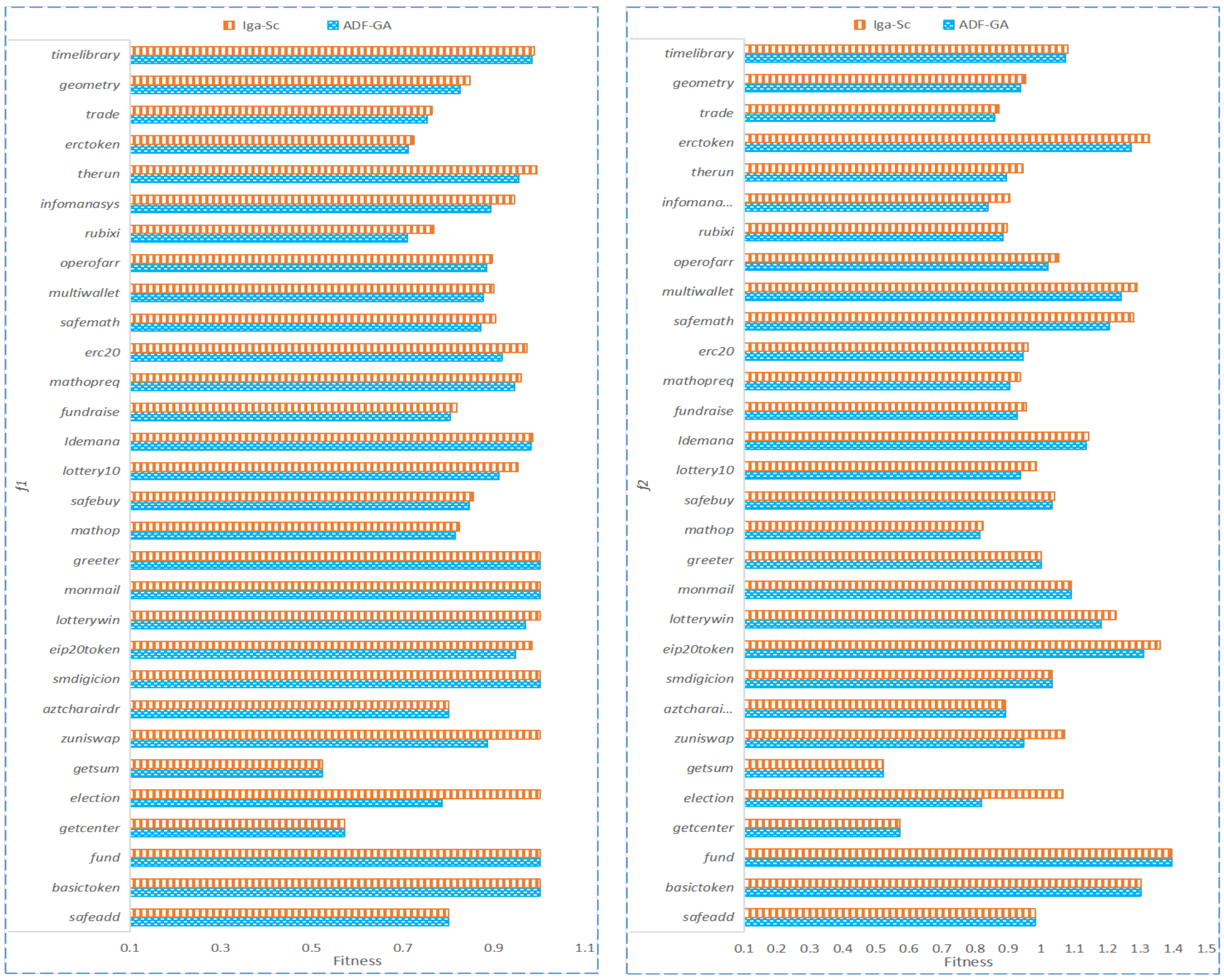
where n denotes the number of def-use pairs that need to be tested, and m denotes the number of def-use pairs covered by the current test case tc.

B. Experimental Result

We ran Iga-Sc, ADF-GA, GA-C#, and RT on top of the 30 smart contracts to generate test cases for covering their embedded def-use pairs. The in-depth experimental analysis against each aforementioned indicator can be found below.

1) Coverage Analysis: The test cases generated with different approaches for each smart contract are input into the corresponding instrumented contract to analyze the coverage rates of *N_dup* and *R_dup*. Fig. 6(a) and Fig. 6(b) respectively show the results of *N_Coverage* and *R_Coverage* on the 30 smart contracts with the approaches.

The average *N_Coverage* of Iga-Sc for the 30 smart contracts is 89.2% and the average *R_Coverage* for the 23 smart



(a) Statistical Results of f1 (b) Statistical Results of f2

Fig. 7. Fitness of Population for Different Approaches

contracts is 90.44% since the other 7 cases have no *R_dups*. It demonstrates that Iga-Sc can effectively generate test cases with high coverage. We can also find that both *N_Coverage* and *R_Coverage* of the test cases generated by Iga-Sc are significantly higher than those of ADF-GA, GA-C#, and RT. Iga-Sc increases the coverage of the test cases for def-use pairs through the incorporation of *Pbest* and *gbest* and the improved fitness function.

The fitness values of the test cases generated by the GA in ADF-GA and by the improved GA in Iga-Sc are calculated based on two different fitness functions, where *f1* is the traditional fitness function (shown in Equation (3)) and *f2* is the improved fitness function (shown in Equation (1)). The fitness values for the 30 smart contracts based on *f1* and *f2* are respectively shown in Fig. 7 (a) and (b). It shows that the fitness value of the test cases generated by Iga-Sc is equal to or greater than that of ADF-GA. The average fitness of Iga-Sc and ADF-GA for the 30 smart contracts based on *f1* are 0.892 and 0.865, and the average fitness based on *f2* are 1.031 and 0.998, respectively. The improved GA can generate better test cases than the GA since it achieves a higher fitness compared to the GA. It can be concluded that the improved GA performs more effectively than the GA.

2) Efficiency Analysis: To evaluate the efficency of Iga-Sc, the number of iterations of GA and the execution time of Iga-Sc, ADF-GA, and GA-C#, are calculated and compared.

The number of iterations required in the algorithm execution for generating test cases are given in TABLE VIII. Iteration A represents *Iteration* of the three approaches. Iteration B1 and Iteration B2 represent the numbers of iterations required by Iga-Sc to achieve the same optimal coverage acquired by GA-C# and ADF-GA, respectively. The results of *iterations* of the three approaches are shown in TABLE VIII. It can be observed that the number of iterations of Iga-Sc is smaller than the other two approaches for most of the contracts. The average number of iterations for Iga-Sc is 15.61, which is significantly lower than 21.02 for GA-C# and 19.03 for ADF-GA. The average number of iterations of Iga-Sc to achieve the same optimal coverage of GA-C# and ADF-GA are 6.93 and 8.28, respectively, which are notably lower than the number of iterations required by the two approaches. This is because the improved GA can accelerate the optimization process and achieve a higher fitness value. Therefore, it requires smaller number of iterations to achieve the same optimal fitness acquired by the other approaches.

The results of *Execution Time* given in TABLE IX show that the average execution time of Iga-Sc in 30 smart contracts is

TABLE VIII
Iteration of Different Approaches

SC Name	Iteration A			Iteration B1	Iteration B2
	GA-C#	ADF-GA	Iga-Sc		
safeadd	7.50	7.00	7.00	1.00	1.00
basictoken	8.25	7.25	7.00	2.00	2.00
fund	15.00	16.75	12.00	2.75	3.25
gencenter	9.25	8.00	6.75	1.25	1.25
election	8.00	7.75	7.25	1.00	1.00
getsum	21.75	21.00	17.00	4.75	5.25
zuniswap	16.00	16.25	9.50	4.25	5.00
aztcharairdr	10.00	7.25	4.50	2.25	2.25
smdigicion	8.75	9.25	7.00	2.5	2.5
eip20token	15.25	13.75	11.75	4.00	4.00
lotterywin	12.25	10.75	7.75	3.00	3.50
monmail	7.00	7.75	4.00	1.25	1.25
greeter	14.50	13.25	13.75	5.25	5.75
mathop	30.00	30.00	25.25	7.00	7.25
safebuy	22.00	25.25	20.75	5.25	12.75
lottery10	30.50	26.00	21.25	10.75	14.25
idemana	26.75	23.00	20.50	14.50	15.25
fundraise	14.50	21.00	8.25	2.50	3.50
mathopreq	32.00	21.50	20.50	7.75	10.75
erc20	21.25	21.00	21.00	11.25	13.50
safemath	33.75	25.75	21.00	12.00	15.75
multiwallet	25.50	22.25	16.75	11.00	11.50
operofarr	27.75	18.25	12.00	5.50	7.00
rubixi	26.00	24.00	22.25	15.50	16.00
infomanasys	26.75	26.25	22.25	11.25	11.50
therun	22.00	21.50	20.75	13.75	15.00
erctoken	28.50	22.75	19.75	10.25	13.00
trade	25.75	31.25	26.25	9.25	11.25
geometry	31.75	18.25	16.25	8.50	9.25
timelibrary	52.25	46.75	38.25	16.50	22.75
Average	**21.02**	**19.03**	**15.61**	**6.93**	**8.28**

TABLE IX
Execution Time of Different Approaches

SC Name	RT(s)	GA-C#(s)	ADF-GA(s)	Iga-Sc(s)
safeadd	32.9612	3.0000	2.8378	2.9932
basictoken	32.3127	3.1317	2.8174	2.8154
fund	33.8622	6.0330	6.7771	4.9656
getcenter	33.4846	3.8869	3.2464	2.7581
election	33.4840	3.0208	2.9171	2.8159
getsum	32.8731	8.5434	8.3202	7.0006
zuniswap	33.7371	6.1856	6.3083	3.7696
aztcharairdr	33.3590	4.1840	2.9421	1.8621
smdigicion	33.1121	3.6190	3.7796	2.9218
eip20token	33.5844	6.0756	5.3983	4.8246
lotterywin	33.3973	4.8510	4.1560	3.2457
monmail	33.2942	2.8686	3.1512	1.6424
greeter	33.5336	5.7594	5.1092	5.6183
mathop	34.4372	12.1080	12.0180	10.8828
safebuy	33.4648	8.9188	10.3475	8.7026
lottery10	33.2926	12.4928	10.5508	8.9123
idemana	33.3230	10.9889	9.4852	8.7412
fundraise	33.6634	5.7304	8.5260	3.4320
mathopreq	34.7843	13.0048	8.7419	8.6838
erc20	33.4951	8.5255	8.4588	9.0048
safemath	34.7832	13.6958	10.3412	8.6226
multiwallet	33.4456	10.2816	8.9178	6.9312
operofarr	34.2238	11.3664	7.6614	5.2536
rubixi	33.1341	10.4988	9.6480	9.0869
infomanasys	33.7525	10.7375	10.4843	9.3094
therun	33.3264	8.9848	8.6688	8.7731
erctoken	33.3915	11.7078	9.1774	8.3266
trade	33.3445	10.4442	12.5063	11.9018
geometry	34.3203	13.5319	7.5519	6.8250
timelibrary	36.5817	24.9024	21.5424	18.5360
Average	**33.6587**	**8.6360**	**7.7463**	**6.6386**

6.6386 seconds, which is the lowest compared to 33.6587s, 8.6360s and 7.7463s for RT, GA-C# and ADF-GA. The reason why the execution time of RT is so high is that it is executed iteratively to generate test cases of the same number. The iteration of RT terminates if it achieves the same coverage acquired by Iga-Sc; otherwise it terminates if the maximum number of iterations is reached. In the experiments, RT requires the maximum number (i.e. 100) of iterations for each smart contract, whereas its coverage is still lower than Iga-Sc. In general, it can be concluded that Iga-Sc significantly improves the performance of test case generation for smart contracts from the perspectives of coverage quality and efficiency.

VI. Threats to Validity

In this section, we discuss the possible validity threats of the proposed approach.

A. *Internal Validity*

There are three main threats against the internal validity. First, the parameters adopted for the improved GA, such as the population size, the methods of selection and crossover, and the possibility of mutation, may affect the coverage of the generated test cases and the efficiency of the algorithm. Second, the parameter ε specified in the fitness function may also affect the experimental results. Third, the measured time cost for each iteration relies on the actual experimental environment, which might be different in a different environment.

To address the randomness of the experimental study, the same initial population is used by the three approaches for each smart contract during the test case generation.

B. *External Validity*

The dataset used in our experiment is based on open-source contracts collected from GitHub and real smart contracts deployed on the Ethereum platform. Since the size of the experimented smart contracts is not large and these smart contracts only contain numerical inputs, we cannot claim that this dataset represents all the available smart contracts. Although the size of the smart contracts is small, they have diverse sizes and the same constructs as large-scale smart contracts. Therefore, it is believed that our approach would have the feasibility to handle smart contracts with larger sizes.

In addition, *Solidity* is a quickly evolving language, whose version was updated from version 0.4.0 to 0.8.0 in just a few years. With the continuous development of *Solidity*, new syntax and semantics may bring novel challenges to test case generation of *Solidity*-based smart contracts.

VII. Conclusion

This paper presents a test case generation technique for *Solidity*-based smart contracts. On the one hand, there is a lack of research on data flow testing for smart contracts. On the other hand, it is difficult to balance the coverage of generated test cases and the efficiency of test case generation. We propose an improved Genetic Algorithm-based test case generation approach for smart contract data flow testing, called Iga-Sc, in which we introduce the theory of Particle Swarm Optimization to improve Genetic Algorithm. Within the improved approach, *pbest* and *gbest* are introduced to save generated optimal test cases. The recombination of parent populations is adopted to reduce the randomness of the Genetic Algorithm. The improvements make it more effective and efficient to find the optimal solution.

A dataset comprising 30 open-source smart contracts is collected from Ethereum and GitHub to perform the experimental study. The comparative experiments among Iga-Sc and three baseline models, including ADF-GA, GA-C#, and RT are performed to evaluate the coverage of the generated test cases on def-use pairs and the efficiency of the algorithms. The experimental results show that Iga-Sc can notably reduce the execution time of the algorithm, while generating a set of test cases with overall high coverage.

In future work, we will expand the dataset with more large-size and complex smart contracts for large-scale experiments. In addition, we will design more advanced fitness functions to consider whether the same def-use pairs are covered by test cases and further guide the searching of optimal test cases. We also will detect vulnerabilities in smart contracts with the generated test cases.

Acknowledgment

The work is supported by the National Natural Science Foundation of China (U21B2016, 61702159), the Fundamental Research Funds for the Central Universities of China (B220202072, B210202075) and the Natural Science Foundation of Jiangsu Province (BK20191297) .

References

[1] W. Zou, D. Lo, P. S. Kochhar, X. B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[2] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[3] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, p. 101227, 2020.

[4] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

[5] J. W. Liao, T. T. Tsai, C. K. He, and C. W. Tien, "Soliaudit: smart contract vulnerability assessment based on machine learning and fuzz testing," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 458–465.

[6] Z. Zheng, S. Xie, H. N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[7] L. Ante, "Smart contracts on the blockchain–a bibliometric analysis and review," *Telematics and Informatics*, vol. 57, p. 101519, 2021.

[8] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[9] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[10] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE transactions on software engineering*, no. 4, pp. 367–375, 1985.

[11] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using genetic algorithms to aid test-data generation for data-flow coverage," in *14th Asia-Pacific Software Engineering Conference (APSEC'07)*. IEEE, 2007, pp. 41–48.

[12] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[13] M. Pei, E. D. Goodman, Z. Gao, and K. Zhong, "Automated software test data generation using a genetic algorithm," *Michigan State University, Tech. Rep*, no. 1, pp. 1–15, 1994.

[14] M. R. Rajkumari and B. Geetha, "Automatic test data generation using genetic algorithm and program dependence graph," *Journal of Computer Applications*, vol. 3, no. 4, p. 1, 2010.

[15] N. Nayak and D. P. Mohapatra, "Automatic test data generation for data flow testing using particle swarm optimization," in *International conference on contemporary computing*. Springer, 2010, pp. 1–12.

[16] P. Zhang, J. Yu, and S. Ji, "Adf-ga: data flow criterion based test case generation for ethereum smart contracts," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 754–761.

[17] M. R. Girgis, A. S. Ghiduk, and E. H. Abd-Elkawy, "Automatic generation of data flow test paths using a genetic algorithm," *International Journal of Computer Applications*, vol. 89, no. 12, pp. 29–36, 2014.

[18] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[19] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[21] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[22] W. K. Chan and B. Jiang, "Fuse: An architecture for smart contract fuzz testing service," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 707–708.

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

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

[25] X. Wang, Z. Xie, J. He, G. Zhao, and R. Nie, "Basis path coverage criteria for smart contract application testing," in *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, 2019, pp. 34–41.

[26] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for ethereum smart contract," *CoRR*, vol. abs/1908.03707, 2019, [Online]. Available: https://arxiv.org/abs/1908.03707.

[27] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1198–1201.

[28] X. Wang, H. Wu, W. Sun, and Y. Zhao, "Towards generating cost-effective test-suite for ethereum smart contract," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 549–553.

[29] K. B. Kim and J. Lee, "Automated generation of test cases for smart contract security analyzers," *IEEE Access*, vol. 8, pp. 209 377–209 392, 2020.

[30] Y. Liu, Y. Li, S. W. Lin, and Q. Yan, "Modcon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1601–1605.

[31] 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," *CoRR*, vol. abs/2102.08864, 2021, [Online]. Available: https://arxiv.org/abs/2102.08864.

[32] 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, 2017.

[33] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[34] S. Sheoran, N. Mittal, and A. Gelbukh, "Artificial bee colony algorithm in data flow testing for optimal test suite generation," *International Journal of System Assurance Engineering and Management*, vol. 11, no. 2, pp. 340–349, 2020.

[35] S. Singla, D. Kumar, H. Rai, and P. Singla, "A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts," *International journal of advanced science and technology*, vol. 37, pp. 15–26, 2011.

[36] S. Kumar, D. K. Yadav, and D. A. Khan, "A novel approach to automate test data generation for data flow testing based on hybrid adaptive pso-ga algorithm," *International Journal of Advanced Intelligence Paradigms*, vol. 9, no. 2-3, pp. 278–312, 2017.

[37] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck: Quickly detecting smart contract problems through regular expressions," *CoRR*, vol. abs/1911.09425, 2019, [Online]. Available: https://arxiv.org/abs/1911.09425.

[38] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[39] L. Mei, W. Chan, T. Tse, and F. C. Kuo, "An empirical study of the use of frankl-weyuker data flow testing criteria to test bpel web services," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1. IEEE, 2009, pp. 81–88.

[40] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," *Queen's University Belfast*, 2010.

[41] S. Kumar, D. K. Yadav, and D. A. Khan, "An accelerating pso algorithm based test data generator for data-flow dependencies using dominance concepts," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 2, pp. 1534–1552, 2017.

[42] Y. Shi *et al.*, "Particle swarm optimization: developments, applications and resources," in *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, vol. 1. IEEE, 2001, pp. 81–86.

[43] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS'95. Proceedings of the sixth international symposium on micro machine and human science*. IEEE, 1995, pp. 39–43.

Shunhui Ji received the B.S. degree in computer science and technology and the Ph.D. degree in computer software and theory from Southeast University, in 2008 and 2015, respectively. She is currently an Associate Professor with the College of Computer and Information, Hohai University, Nanjing, China. Her research interests include service computing, cloud computing, software modeling, analysis, testing, and verification. She is a Reviewer of some international conferences and journals.

Shaoqing Zhu is an M.S. candidate in the College of Computer and Information, Hohai University, Nanjing, China. He received his bachelor's degree in Aliyun School of Big Data from Changzhou University, Changzhou, China in 2020. His current research interests include software analysis and testing.

Pengcheng Zhang received the Ph.D. degree in computer science from Southeast University in 2010. He is currently a Professor with the College of Computer and Information, Hohai University, Nanjing, China, and was a visiting scholar at San Jose State University, USA. His research interests include software engineering, services computing and data science. He has published in premiere or famous computer science journals. He was the co-chair of IEEE AI Testing 2019 conference. He served as technical program committee member on various international conferences. He is a member of the IEEE and has published in premiere or famous computer science journals, such as *IEEE Transactions on Services Computing*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Big Data*, *IEEE Transactions on Emerging Topics in Computing*, and *IEEE Transactions on Software Engineering*.

Hai Dong is a Lecturer at School of Computing Technologies in RMIT University, Melbourne, Australia. He was previously a Vice-Chancellor's Research Fellow in RMIT University and a Curtin Research Fellow in Curtin University, Perth, Australia. He received a PhD from Curtin University. His primary research interests include: Service-Oriented Computing, Distributed Computing, Cyber Security, Machine Learning and Data Analytics. He is a senior member of the IEEE and has published a monograph and more than 100 research articles in international journals and conferences, such as *ACM Computing Surveys*, *Communications of the ACM*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Industrial Informatics*, *IEEE Transactions on Industrial Electronics*, *Journal of Computer and System Sciences*, *World Wide Web*, *ICSOC*, *ICWS*, etc. He received the Best Paper Award in *ICSOC* 2016.

Jianan Yu received her master's degree from the College of Computer and Information, Hohai University, Nanjing, China in 2021, and her bachelor's degree in the College of Internet of Things Engineering from Hohai University, Nanjing, China in 2018. Her current research interests include software analysis and testing.