Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts

Olsthoorn, Mitchell; van Deursen, A.; Panichella, A.

Publication date
2022

Document Version
Accepted author manuscript

Published in
The 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022)

Citation (APA)
Olsthoorn, M., van Deursen, A., & Panichella, A. (Accepted/In press). Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts. In *The 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022)* IEEE .

Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts

1st Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

2nd Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

3rd Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

***Abstract*—Transaction-reverting statements are key constructs within *Solidity* that are extensively used for authority and validity checks. Current state-of-the-art search-based testing and fuzzing approaches do not explicitly handle these statements and therefore can not effectively detect security vulnerabilities. In this paper, we argue that it is critical to directly handle and test these statements to assess that they correctly protect the contracts against invalid requests. To this aim, we propose a new approach that improves the search guidance for these transaction-reverting statements based on interprocedural control dependency analysis, in addition to the traditional coverage criteria. We assess the benefits of our approach by performing an empirical study on 100 smart contracts *w.r.t.* transaction-reverting statement coverage and vulnerability detection capability. Our results show that the proposed approach can improve the performance of *DynaMOSA*, the state-of-the-art algorithm for test case generation. On average, we improve transaction-reverting statement coverage by $14\,\%$ (up to $35\,\%$), line coverage by $8\,\%$ (up to $32\,\%$), and vulnerability-detection capability by $17\,\%$ (up to $50\,\%$).**

***Index Terms*—test case generation, search-based software engineering, smart contracts, interprocedural analysis**

I. Introduction

Ever since its launch in 2015, *Ethereum* has been the largest and most prominent smart contract platform [1]. One key property of these smart contracts is that once a contract has been deployed, it cannot be updated [2]. This property makes sure that contracts that are in use on the platform cannot be altered by the creators of the contract for their benefit. However, this creates certain challenges, *e.g.,* what happens if a bug is discovered. This greatly increaces the importance of quality assurance in the smart contract development lifecycle. In the last few years, various search-based methods have been developed to assist developers with this problem, like *fuzzing* [3], [4], [5], [6], [7], [8] and *test case generation* [9].

Smart contracts on the *Ethereum* platform are written in *Solidity*, a high-level smart contract language. *Solidity* is a transactional language, meaning transactions either succeed or fail. Hence, it is impossible for the contract to be in a broken state. To accomplish this, the language makes use of *transaction-reverting statements*, which allow developers to check the validity of requests. When the conditions of such statements are not met, an exception is raised and all modifications made by a given request to the current state are reverted [10]. For the purpose of checking the (external) inputs or the validity of the state to receive such inputs, *Solidity* provides the `require` routine – this transaction-reverting statement makes the contract robust against improper usage. Typical examples of `require` statements are: (i) checking that certain requests can only be done by the owner of the contract —*i.e.,* `require(msg.sender==owner)` — or (ii) that a transaction amount is positive —*i.e.,* `require(amount>0)`.

A recent study by Liu *et al.* [11] shows that transaction-reverting statements are extensively used within *Solidity* smart contracts for authority and validity checks. They found that removing or modifying these statements may compromise the security of the smart contract. Additionally, the study showed that existing *Solidity* testing tools cannot effectively detect security vulnerabilities caused by these statements. Internally, the `require` statements are just normal function calls that are handled in a special way by the interpreter. Existing search-based approaches [3], [9], however, treat these statements as any other function call without taking this critical construct of *Solidity* into account. In particular, there is no gradient in the fitness landscape that the search algorithm could use as guidance to satisfy the condition of these statements, the search algorithm has to resort back to random testing.

In this paper, we argue that these transaction-reverting statements should be treated as first class citizens during testing, since any error in them likely corresponds to a security vulnerability. To this aim, we propose a new approach to improve the search guidance for transaction-reverting statements, without changing the semantics of the contract under test. First, we statically analyze the contract under test and identify the transaction-reverting statements and modifiers. *Modifiers* are interprocedural constructs that group transaction-reverting statements that are executed by the Ethereum Virtual Machine (EVM) as a dependency for certain methods. Then, we perform interprocedural dependency analysis to link the *Control Flow Graph* (CFG) of the method under test with the associated modifiers. Lastly, we calculate an interprocedural-level fitness value (*i.e.,* to guide the search process) based on the runtime data collected by a context-sensitive instrumentation of the transaction-reverting statements. The new fitness function allows to measure how far a test case is from satisfying the condition within these statements.

To evaluate the effectiveness of our approach, we implemented it within *SynTest-Solidity* [12] — a test case generation framework for *Solidity* that implements *DynaMOSA*. *DynaMOSA* is the state-of-the-art algorithm for unit test case generation originally designed for Java programs [13], [14] and recently applied to *Solidity* [9]. It is guided by state-of-the-art unit-level fitness functions that consider structural coverage [15]. We performed an empirical study on 100 real-world smart contracts gathered from *etherscan.io*. We compare the results achieved by *DynaMOSA* with and without the improved guidance with regard to vulnerability detection capability and structural coverage.

Our results show that *DynaMOSA* covers significantly more transaction-reverting statements with the improved guidance in 37 % of the smart contracts, with an average increase in transaction-reverting statement coverage of 12 %. This also leads to an average increase in line coverage by 8 %. Finally, our approach helps *DynaMOSA* detect more vulnerabilities, with 17 % (on average) more captured vulnerabilities for those contracts on which we observe an increase in transaction-reverting statement coverage.

In summary, this work makes the following contributions:

1) A lightweight approach based on interprocedural analysis to improve the search guidance for transaction-reverting statements (Section III).
2) An implementation of our approach[1] within a state-of-the-art *Solidity* smart contract testing tool, named *SynTest-Solidity* [12].
3) A *Solidity* smart contract benchmark consisting of a diverse set of 100 real-world smart contracts (Section IV-B).
4) An empirical study demonstrating the benefit of the proposed approach (Section V).
5) A full replication package including the code, results, and the scripts to analyze the results [16].

While in this paper we focus on *Solidity* smart contracts, this approach can be benefecial for any programming language with explicit contracts or declarative input validation rules.

II. Background and Related Work

This section provides an overview of basic concepts and related work on smart contracts, fuzzing, and test case generation.

A. Smart Contracts and Ethereum

In the last decades, there has been an increased focus on creating decentralized services to cut out intermediaries from the interaction between people. One example of this trend is smart contracts —digital agreements between multiple parties on how certain tasks need to be executed— and in particular *Ethereum*, the most popular smart contract platform [1]. The main benefits that smart contracts can provide are trustless interactions, automated task handling, and hosting of decentralized applications (dApps). Smart contracts are built on top of a blockchain, a tamperproof ordered ledger. When a smart contract gets deployed, it creates a transaction containing code (a collection of functions) and data (state) that resides at a specific address on this ledger. Users can make requests to this address, using the functions to modify the state of the contract. Each state modification creates a new transaction on the blockchain. This chain of blocks will grow over time with the addition of new contracts and requests. Since the logic that can be applied to the state is fixed and the state is publically available, users in the network can verify if a transaction was properly executed.

Ethereum runs on a decentralized network of nodes. These nodes process the requests made to the contracts and create the blocks needed to modify state and deploy new contracts. To secure the platform against attacks, there should be consensus between the nodes. To get consensus within the network, *Ethereum* makes use of the mechanism called *Proof of Work* (PoW). PoW relies on a computationally-expensive mathematical problem that is difficult to calculate, but easy to verify. The random node that solves the problem first gets to decide which transactions are accepted.

B. Transaction-Reverting Statements

Since smart contracts cannot be modified once deployed, it is crucial that they are thoroughly tested to detect and remove potential vulnerabilities. In addition, transaction-reverting statements are used by developers to further assess the validity of requests and verify that the contract remains in a valid state. Hence, it is critical that these statements are correctly added to assess the important properties of the contract under analysis.

To better show how these reverting statements work, let us consider the simplified example of a *Solidity* smart contract

```
pragma solidity ^0.5.0;
contract Account {
  address public owner;
  mapping (address => uint) private balances;

  constructor() public {
        owner = msg.sender;
  }

  modifier isOwner() {
    require(msg.sender == owner, "You are not the owner"
        );
    _;
  }

  function withdraw(int amount) public isOwner {
    require(amount > 0, "Amount too low");
    if (amount <= balances[msg.sender]) {
      balances[msg.sender] -= amount;
      msg.sender.transfer(amount);
    }
    return balances[msg.sender];
  }
  ...
}
```

Listing 1: Example *Solidity* smart contract

[1]https://github.com/syntest-framework/syntest-solidity

shown in Listing 1 that represents a bank account. On lines 6-8, the owner of the account is set to the creator of the contract. Lines 10-13 define a `modifier`, consisting of a transaction-reverting statement that is executed by the Ethereum Virtual Machine (EVM) as a dependency for the `withdraw` method. Lastly, the method on lines 15-22 allows users to withdraw money from the account. The `withdraw` method makes use of the `isOwner` modifier to guarantee that only the owner of the account can withdraw money. In addition, the method uses a local reverting statement (line 16) to check if the amount to withdraw is positive. When the `require` check on line 16 fails, state-of-the-art coverage heuristics (like used by existing search-based approaches [3]) would assume that line 17 and 21 are also covered. In reality, however, only line 16 is covered and the execution is halted.

C. Testing Solidity Smart Contracts

Various techniques have been used in literature to test *Solidity* smart contracts. An overview of the different techniques is available in the recent survey by Ren *et al.* [17].

Static Analysis [18], [19], [20]: Static analysis tools analyze a contract for vulnerabilities without running it. This can be done at both a source code and a byte-code level. The benefit of analyzing a contract statically is that the entire contract can be scanned at once. However, static analysis tools often have a high false-positive rate requiring manual verification [17].

Symbolic Execution [21], [22]: Symbolic execution tools also statically analyze a contract. What differentiates symbolic execution tools is that they keep track of all constraints they encounter on every path through the code. This allows these tools to perform constraint solving to determine which range of input values will lead to certain branches. Symbolic Execution, however, unavoidably suffers from problems like path explosion [23].

Formal Verification [24]: Formal verification methods transpose the source code of the contract to a mathematical proof language. Within this proof language, this method mathematically checks the source code against a manually constructed model of the code's behaviour. This method provides the most security, however, it requires developers to construct a complex model in a different language than the contract.

Fuzzing [3], [4], [5], [6], [7], [8]: Fuzzing automatically generates test data. This data is fed to the contract under test to see how the contract responds to it. This technique is very effective at finding inputs that make the contract crash. However, it cannot be used for verifying the behavior of the contract. Besides, it only focuses on test data without generating complete test cases (*e.g.,* without assertions).

Test Case Generation [9]: Test case generation generates test data, method sequences, and assertions. One study used this technique. However, this study [9] uses existing algorithms without adapting them to Solidity.

D. Search-based Testing and Fuzzing

Search-based software testing (SBST) is a well studied research area that focuses on automating the generation of test data and test cases. Automatic test case generation significantly reduces the time needed for testing applications [23] and has been successfully used in industry [25], [26]. Various studies have been performed that use meta-heuristics to test programs at different levels *e.g.,* unit [27], integration [28], and system-level [29]. These studies have shown that these techniques are effective at achieving high coverage [30] and detecting faults [31], [32], [33].

One of the most commonly used classes of meta-heuristics is Evolutionary Algorithms (EAs) [30], [34], [35]. EAs are inspired by the process of natural selection. They evolve an *initial population* of randomly generated individuals (test data or test cases). These individuals are then *evaluated* based on a predefined fitness function. After the evaluation, the individuals with the best fitness values are selected for *reproduction*. Reproduction creates new offspring by applying mutation (small delta changes to an individual) and crossover (exchanging information between two individuals). Lastly, the new population is created by *selecting* the best individuals across the parents (current population) and the offspring (newly created test data or test cases). These three steps *evaluation*, *reproduction*, and *selection* happen in a loop until a stopping condition has been met. After the search process ends, an archive is created with the best individuals from the population [13], [14].

EAs are often used in fuzzing for generating input data. For example, Nguyen *et al.* [3] used an efficient genetic algorithm for fuzzing *Solidity* smart contracts. The main difference between fuzzing and test case generation is that the former focuses on generating test inputs while the latter aims to generate full test cases, including input data, method sequence, and assertions.

E. Unit-level Fitness function

The purpose of a fitness function is to measure and indicate how far off the individual (test) is from satisfying a test objective, *e.g.,* branches. In SBST, the *de facto* fitness function is made up of two *heuristics*: *approach-level* and *branch distance* [36], [27], [3], [13]. The *approach-level* relies heavily on the Control Flow Graph (CFG). A CFG represents the flow of the logic within a function of a program — all paths that might be traversed during the execution of the program. CFGs are created from the Abstract Syntax Tree (AST) provided by the parser of the language, in our case the *Solidity* compiler. A node in the CFG is called a *basic block* and corresponds to a sequence of statements that are always executed altogether [37], *i.e.,* with no branches inside the block. The *approach-level* uses the CFG and the data that is gathered from the instrumentation during runtime to measure how far, in terms of graph distance, the execution flow is removed from the targeted branch point. More precisely, the instrumentation data is used to determine which branches of the CFG have been covered by the test case. Afterwards, the fitness function calculates the shortest difference along the CFG between the targeted branch node and the closets covered node. Once the execution path reaches the targeted branch node, the fitness function uses the *branch distance* to calculate how far the input variable is from satisfying the condition of the target *true* or *false* branch.

F. Testability Transformations

The *flag* problem is a common issue in SBST [38], [39] that manifests when the conditions in the if-statements are not explicit (*e.g.,* an inline method call like `if (isNull(y)))` or it reads boolean variables (*e.g.,* `if(y==true)`). To address this problem, researchers have proposed *testability transformations* [39], which transform the program under test into an equivalent one (*i.e.,* by preserving the semantics) where the conditions are replaced with predicates reading non-boolean variables. Prior studies have shown that testability transformations dramatically improve code coverage without the need for adapting the underline search algorithms [40], [41], [38].

Compared to these prior studies, we do not apply testability transformation for two reasons. First, creating testability transformations that fully preserve the semantics of program is challenging, limiting its practical applicability [39]. Second, state-reverting conditions are internal subroutines executed by the EVM at run-time and not part of the branch conditions of the source program under test and, therefore, they cannot be transformed.

III. Approach

This section outlines our approach to improve the search guidance (*i.e.,* restoring the gradient) for transaction-reverting statements using the contract shown in Listing 1 as a running example.

A. Problem Definition

A primary challenge in SBST is defining an effective fitness function that guides the search algorithm toward covering an uncovered branch. As an example of test objectives, let us consider the false branch of the `if` condition in line 17 for the method `withdraw` in Listing 1 and its CFG depicted in Fig. 2a. If we apply the state-of-the-art unit-level fitness function, we obtain the *fitness landscape* depicted in Fig. 1. This fitness landscape shows the fitness values for the false branch with varying inputs for the `amount` parameter. The inputs where the fitness function is zero lead to covering the target branch. Ideally, the fitness function should have a gradient to effectively guide the search algorithms. However, in Fig. 1, we can observe that the landscape is *flat* for all negative values of `amount`. This is due to the program execution ending when the condition within the `require` in line 16 of Listing 1 is not met, without providing any information on how close the execution is to satisfying that condition.

The problem of the flat landscape does not apply only to our example but it generalizes to all contracts that have transaction-reverting statements. As shown by Liu *et al.* [11], these statements are extensively used in smart contracts for authority and validity checks. Therefore, explicitly considering

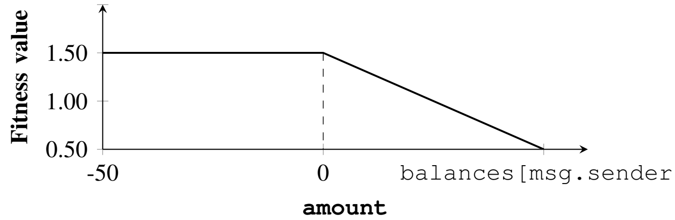


Fig. 1: Fitness landscape for the false branch of the `if` in line 17 of the method `withdraw` in Listing 1

these constructs when computing the fitness function is critical to restore the gradient and make the search more effective. Otherwise, the search algorithm has to resort to random testing when encountering such transaction-reverting statements. This approach is not ideal as random testing (*i.e.,* without guidance) is slow and might not lead to a solution within the allocated search budget. In practice, this means the search algorithm either randomly guesses the input values needed to satisfy the condition or gets stuck.

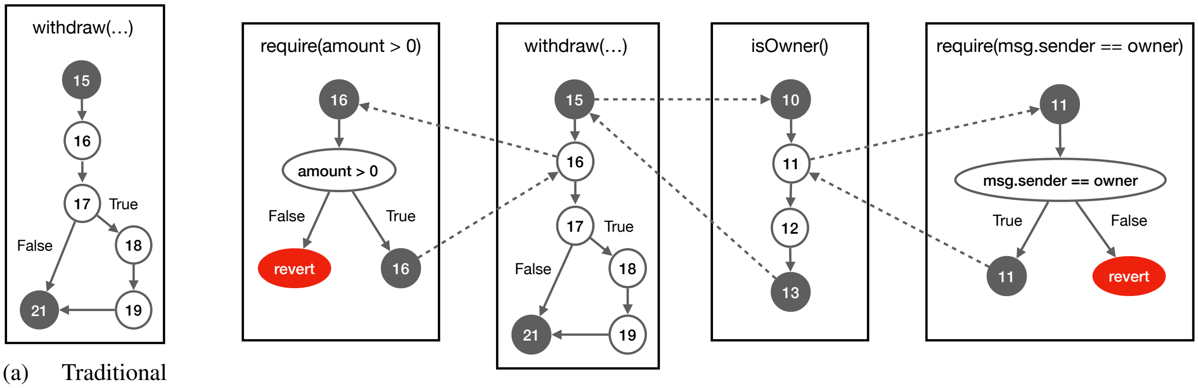
Additionally, in Listing 1, we can see that the `withdraw` method defines a dependency on the `isOwner` modifier. In this example contract, the `require` statement within the `isOwner` modifier (line 11) has to be satisfied before the main branch of the `withdraw` function can be executed. As a consequence, the search algorithm has to overcome two independent obstacles without guidance through random testing before it can reach the branch in line 17.

B. Overview

The goal of our approach is to restore the gradient for *Solidity* smart contracts containing transaction-reverting statements, by providing a quantitative measurement on how far a test case is from satisfying these statements. To this aim, we first statically analyze the Abstract Syntax Tree (AST) of the contract under test and identify the transaction-reverting statements and modifiers (**Step 1**). Then, we perform interprocedural control dependency analysis to determine the control flow across the different methods and sub-routines (**Step 2**). Lastly, we define a new interprocedural fitness function based on the runtime data collected by the context-sensitive instrumentation of transaction-reverting statements (**Step 3**). The last two steps will be further explained in the next subsections.

C. Interprocedural Dependency Analysis

The idea behind the interprocedural dependency analysis is to determine how the transaction-reverting statements and modifiers impact the execution of the method under test at runtime. To explain how this analysis works, we will use the example in Fig. 2. Figure 2a depicts the traditional CFG for the `withdraw` method in Listing 1 while Figure 2b shows the results of enriching it with our interprocedural dependency analysis. In these two figures, the gray nodes represent the flow entry and exit blocks of the CFG. The numbers within the nodes indicate the line number of the statement that the block



(a) Traditional CFG

(b) CFG with interprocedural dependency analysis

Fig. 2: Control Flow Graphs (CFGs) of the *withdraw* function in Listing 1

represents. Lastly, the solid edges indicate how the execution flows through the nodes.

1) Linking Transaction-Reverting Statements: Transaction-reverting statements are special sub-routines within the EVM and, therefore, they do not have a corresponding CFG nor branching nodes. We apply context-sensitive instrumentation around the transaction-reverting statements to capture their impact on the dependent methods. The instrumentation allows to capture these interprocedural dependencies and build an artificial control flow representation of the sub-routines. In the example of Fig. 2b, we build the control flow of the statement in line 16 (*i.e.,* the box with the header `require(amount > 0)`, which is linked (dashed edges) to the CFG of the `withdraw` method. The red nodes in the sub-routine represent the *Solidity* revert mechanism.

The context-sensitive instrumentation injects two additional instrumentation statements, namely *pre-trs* and *post-trs* (where *trs* stands for transaction-reverting statements). The *pre-trs* and *post-trs* are injected before and after each of these statements, respectively. The *pre-trs* indicates if the execution of the contract reached the statement, meaning the search process is at the revert point. If the *post-trs* is reached, it indicates that the condition of the transaction-reverting statement has been met. If the *pre-trs* has been reached and the *post-trs* has not, the condition has not been met and the execution is halted and reverted.

However, the *pre-* and *post-trs* do not provide information on how to satisfy the particular condition but only if the condition has been met or not. To collect information on how far a test case is from satisfying the conditions, we add additional instrumentation statements (the context) to record the type of operator and the values of the operands from the memory stack at runtime. For example, for the statement `require(amount > 0)`, our instrumentation records the operator > and the runtime value of the `amount` operand and the constant value `0`. This data can be integrated into the fitness function as discussed in Section III-D to restore its gradient.

2) Linking Modifiers: In step 1 of the approach, we analyze the Abstract Syntax Tree (AST) of the contract to compile a list of all modifiers that each method is dependent on. As an example, the method `withdraw` in Fig. 2b depends on a single modifier, called `isOwner`. Note that a modifier cannot be directly invoked but can be tested only through the methods that define it as a dependency. In general, a modifier acts like a template (or around advice in terms of aspect-oriented programming), wrapping its logic around the method that depends on it. Modifiers use a special identifier (`_;`), as can be seen on line 12 of Listing 1, to indicate where the function's logic should be executed. In the example, all statements within the method `withdraw` are post-dominated by the conditions of the `isOwner` modifier. Hence, the statements in `withdraw` are not covered by simply invoking the function if the conditions of `isOwner` are not met.

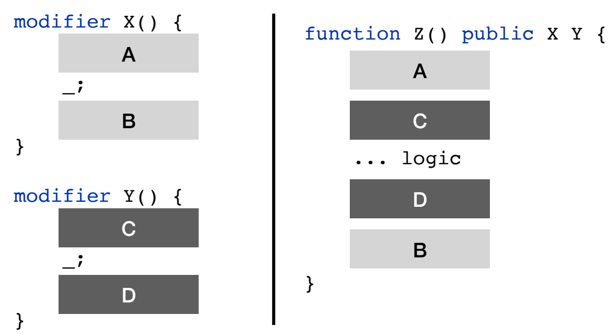


Fig. 3: Modifier structure and execution order

To capture the interprocedural dependencies we build the control flow graph of the modifier and link it to the entry or exit point within the method depending on where the template identifier is located. If a method depends on multiple modifiers, the CFG of each modifier is linked to the dependent method Z in the order they appear in signature of Z in a layered approach.

As an example, consider Fig. 3, which defines two modifiers, named `X` and `Y`, together with their extracted parts (`A`, `B`) and (`C`, `D`), respectively. Method `Z` uses both modifiers in the order they are listed: `X`, `Y`. The overall dependency graph links part `A`, part `C` to the entry point of the body of the method `Z` while its exit point is linked to part `D`, and lastly part `B`.

If the modifier contains transaction-reverting statements, we apply the same procedure described in the previous subsection to the CFG of the modifier. An example of such a case is depicted in Fig. 2b where the `isOwner` modifier contains a `require` statement with the condition `msg.sender == owner`. This condition checks that the request is only made by the owner of the contract. Finally, if a modifier is declared by multiple methods of a contract, the linking procedure (from the modifier CFG to the method CFG) is applied separately for each of these as the context differs among the different methods.

D. Interprocedural Fitness Function

For each branch in the code, we do not simply apply the unit-level fitness function discussed in Section II-E but enrich it with context data collected by the interprocedural dependency analysis.

We define the *interprocedural approach level* as an extension to its unit-level variant. Let t be a test case and b_i be a branch to cover. The interprocedural approach level $IAL(b_i, t)$ is the number of interprocedural control dependencies between the closest executed branch and b_i. The interprocedural control dependencies includes the classic unit-level control nodes (in the CFG) and the interprocedural dependencies related to modifiers and transaction reverting statements. For example in Fig. 2b, the branch 17→21 of the `withdraw` method is control dependent on nodes 15-16 (unit-level dependencies) but also on nodes 10-13 of the *isOwner* modifier and the conditions of the two require statements (nodes 11 and 16).

When the execution of a test t is halted because of a transaction-reverting statement TRS_i, we introduce the *trs-distance*. This distance measures how far t is from satisfying the condition in TRS_i by using Korel's rules [15] for conditions. For example, the *trs-distance* for the statement `require(x == 0)` is computed as $|x - 0|$ [15], which is equal to zero only when the condition $x == 0$ is satisfied.

Therefore, the interprocedural fitness function (f) for a test t *w.r.t.* an uncovered branch b_i is computed as follows:

$$f = \begin{cases} IAL(b_i, t) + \dfrac{d(TRS_i, t)}{d(TRS_i, t) + 1} & \text{if halted at } TRS_i \\ IAL(b_i, t) + \dfrac{d(b_i, t)}{d(b_i, t) + 1} + 1) & otherwise \end{cases} \quad (1)$$

where IAL denotes the interprocedural approach level, $d(TRS_i, t)$ is the *trs-distance* for the transaction-reverting statement TRS_i and $d(b_i, t)$ is the traditional branch distance.

IV. Empirical Study

We carried out an empirical study to assess the effectiveness of the proposed interprocedural fitness function compared to its state-of-the-art unit-level variant. To this aim, we use these functions to guide the state-of-the-art testing algorithm, *DynaMOSA*. We evaluate the impact of the proposed fitness function *w.r.t.* to the following testing criteria: (i) structural (branch, transaction-reverting statement, and line) coverage and (ii) *vulnerability detection capability*.

A. Research Questions

Our empirical evaluation aims to answer the following two research questions:

RQ1 *To what extent does the proposed approach improve the structural coverage achieved by DynaMOSA?*

RQ2 *To what extent does the proposed approach improve the vulnerability detection of DynaMOSA?*

These two research questions aim to evaluate if the proposed approach improves the effectiveness of the state-of-the-art test case generation algorithm *DynaMOSA*. RQ2 reflects the main goal, which is to determine if the proposed approach allows the two algorithms to detect more vulnerabilities in the *Solidity* smart contract under test. We additionally report the structural coverage as test data and test cases cannot detect or capture vulnerabilities in code regions that are uncovered.

B. Benchmark

To evaluate the proposed approach, we created a benchmark consisting of 100 *Solidity* smart contracts. We collected all contracts submitted between January and April of 2021 with *Solidity* versions 5 and 6 from *etherscan.io*. We then selected smart contracts with a cyclomatic complexity of $cc >= 2$, *i.e.,* contracts with at least one conditional statement, *i.e., branch*, *loop*.

A recent study by Ren *et al.* [17] empirically and theoretically criticizes the benchmarks used in prior studies, even those that include the entirety of *etherscan.io*. Moreover, previous studies did not explicitly report the source of the contracts [9] or did not check the cyclomatic complexity [3] as suggested in the literature [42], [43]. This study proposes a benchmark that is more transparant by removing trivial smart contracts ($cc < 2$) and specifying the date and time on which the contracts were submitted to *etherscan.io*.

We ensured that the benchmark contains (i) contracts from different application domains (*e.g.,* wallets, auctions, tokens, financial staking, DAO, voting, insurances); (ii) contracts with and without transaction-reverting statements (70 % use modifiers, 18 % use a single require statement, 62 % use multiple require statements, 5 % use no reverting statements) to validate that the proposed approach does not negatively impact contracts without these constructs; (iii) contracts with a diverse size and complexity. Table I reports the statistics of the 100 *Solidity* smart contracts in our benchmark. In particular, the table reports the minimum, maximum, median, and quartiles (Q_i) of the functions, branches, lines, and transaction-reverting statements in the contracts. The benchmark is available within the replication package.

C. Benchmark Tool & Baseline

To answer the research questions, we implemented our approach within *SynTest-Solidity* [12]. We have used this tool because it generates complete test cases with assertions, which

TABLE I: Statistics (min, max, median and quartiles) of the 100 smart contracts in our benchmark

Code Elements	Min.	Q_1	Median	Q_3	Max
# Functions	2	10	21.5	39	111
# Branches	0	6	12	22	62
# Reverting statements	0	12	27	40.5	102
# Lines	6	53	109.5	154	431

are necessary for capturing vulnerabilities automatically. Instead, other *Solidity* testing tools were either solely built to work as a fuzzer [3] or were not sufficiently extensible to integrate the proposed approach [9]. We briefly describe the state-of-the-art unit-level test case generation algorithm used in *SynTest-Solidity*.

1) DynaMOSA: Dynamic Many-Objective Sorting Algorithm (DynaMOSA) is the state-of-the-art evolutionary search algorithm for test case generation [13]. It models test case generation as a many-objective problem by targeting each test target (*e.g.,* branch, line) simultaneously using a many-objective genetic algorithm. As any evolutionary algorithm, *DynaMOSA* evolves a set of randomly generated test cases (see Section II-D). The fitness of each test case (or individual) is determined based on the approach level, and the branch distance for the remaining uncovered targets. *DynaMOSA* makes use of a dynamic selection of the targets, where test targets are dynamically added based on the control dependency hierarchy when the current target is covered. This dynamic selection improves the efficiency of the search process for smaller search budgets [13]. After evaluating and creating new test cases (offspring), environmental selection is used to select the fittest individuals in the population to survive using the *preference criterion*, *non-dominated sorting*, and *crowding distance*. The *preference criterion* first selects the best test case (the one with the best fitness) for each just-missed branch (front zero). Then, the *non-dominated sorting* selects the remaining test cases based on the concept of Pareto optimality, which is the standard criterion in SBST. Finally, *crowding distance* is in place to promote the diversity among the test cases that are equally good according to the Pareto optimality.

D. Parameter Setting

Previous studies empirically showed [44] that although parameter tuning has an impact on the effectiveness of a search algorithm, the default values, which are commonly used in literature, provide reasonable and acceptable results. For this study, we have chosen to use the following default parameter settings recommended in the literature [45], [27], [44], [13], [46], [47].

Population size. We use a population size of 10 individuals (test cases); We performed a preliminary experiment to determine the size of the population. A population that is too small will not allow for enough exploration and will quickly converge. A population size that is too big will consume more of the search budget per iteration of the search process. Since *Solidity* smart contract tests are performed through an API (in comparison to testing frameworks at unit-level), running tests is drastically slower. In addition, before each test case can be run, the contract has to be deployed to the smart contract network. Therefore, we established that a population of 10 individuals provides sweet spot in the trade off between efficiency and coverage. Our choice of using a relatively small population size is also in line with the recommended population for expensive fitness functions [45], [48].

Mutation Operator. We use the *uniform mutation*, which changes each test case by adding, deleting, or replacing method calls. We use a mutation probability p_m=1/n, where n is the number of statements in the test case as recommended in the literature [27], [44], [13]. For primitive statements (*e.g., int*), the values are mutated using the *polynomial mutation* [46] that is applied with a probability of 80%. For the remaining 20%, the operator applies random sampling.

Crossover Operator. We use the single-point tree crossover with a crossover probability of p_c=0.8, which is within the recommended range $0.50 \leq p_c \leq 0.90$ [47], [49].

Selection. We use the binary tournament selection to sample individuals from the population for reproduction [50].

Search Budget. As a stopping criterion for the search process, we use a search budget based on time instead of the number of executed tests. This was done as a time-based stopping criterion provides the fairest comparison of the different approaches, given that the proposed heuristics add a small computational overhead to the search process. Additionally, practitioners will often only allocate a specific amount of time for the algorithm to run as the time it takes to run a certain number of iteration differs across contracts and across tests for the same contract.

The search budget for the algorithm was set to 30 minutes as this provides a balance between giving the algorithm enough time to explore the search space (considering the slower execution time of a single test case) and making the study infeasible to execute. The algorithm will end prematurely if all its test objectives have been covered. Note that time-based search budgets are considered a less biased stopping criterion than a budget based on the number of executed tests (or fitness evaluation) as not all tests have the same running time [27], [51], [33], [52].

E. Vulnerability Detection

To evaluate how the proposed approach influences the effectiveness of *DynaMOSA* at detecting/capturing vulnerabilities, we considered multiple vulnerable versions of the contracts in our benchmark. We synthesize vulnerable versions that differ from the secure ones by either (i) missing transaction-reverting statements or (ii) transaction-reverting statements with incorrect conditions. As an example, for the contract in Listing 1, one vulnerable version could be obtained by removing the `require` function in line 11. In that case, anyone can withdraw the money from the bank account, not only the owner. Another example of a vulnerable contract version would be if we inverted the condition of the `require` function in line 16. This would allow an attacker to increase the balance of an account by withdrawing a negative amount. Studies have shown that the transaction-reverting statements play a crucial role in the behavior of the contract when testing for faults that cause vulnerabilities [53], [54], [11]. Therefore, we analyze the ability to detect the vulnerability associated with these missing or incorrect statements.

For each contract (with transaction-reverting statements) in the benchmark, we generated 10 vulnerable versions. To assess the vulnerability detection capability, we run the test cases that were generated for the non-vulnerable version of the contract on these vulnerable versions to determine if the test cases fail, and thereby, capturing the vulnerability. Finally, We assess the performance of the testing algorithm with and without our approach measuring the number of vulnerabilities detected by the generated test cases.

F. Experimental Protocol

For each contract in the benchmark, we run *DynaMOSA* with and without the improved guidance. The resulting coverage information for the different evaluation metrics (*i.e.,* branch, reverting statements, line) is collected and stored along with the generated test cases.

Since *DynaMOSA* is a randomized algorithm, we can expect a fair amount of variation in the results of the empirical study. To prevent potential biases in the results, we repeated every experiment 20 times, with a different random seed, and computed the average (median) results. In total, we performed 4000 executions: two configurations of *DynaMOSA* on 100 *Solidity* smart contracts with 20 repetitions each. With each execution taking 30 minutes, the total execution time is 83.5 days of consecutive running time. We ran the experiment on a system with two AMD EPYC™ 7452 using 120 cores running at $2.35\,\mathrm{GHz}$.

To answer *RQ1*, we compare the structural coverage results of the two configuration with each other. To evaluate the vulnerability detection capability of the different approaches (*RQ2*), we compare the same configurations as for *RQ1* but now using the procedure described in Section IV-E.

We use the unpaired Wilcoxon rank-sum test [55] with a threshold of 0.05 to determine if the results of the proposed approach are statistically significant. The Wilcoxon rank-sum is a non-parametric statistical test that determines if two data distributions are significantly different. This is the standard test for evaluating randomized algorithms such as *DynaMOSA* [56]. In addition, we use the Vargha-Delaney statistic [57] to measure the effect size of the result, which indicates how large the difference between the two configurations is.

V. Results

This section discusses the results of our empirical study with the aim of answering the research questions formulated in Section IV-A.

TABLE II: Statistical results for *DynaMOSA* with and without the improved guidance. We report the number of times the proposed approach statistically improve (#Win) or decrease (#Lose) the effectiveness of *DynaMOSA*. Negligible (N), Small (S), Medium (M), and Large (L) denote the $\hat{A}_{12}$ effect size.

Metric	#Win				#Lose				#No diff.
	N	S	M	L	N	S	M	L	
Branch	-	-	2	2	-	-	-	-	96
Rev. statement	-	-	2	35	-	-	-	-	63
Line	-	2	4	29	-	-	-	-	65

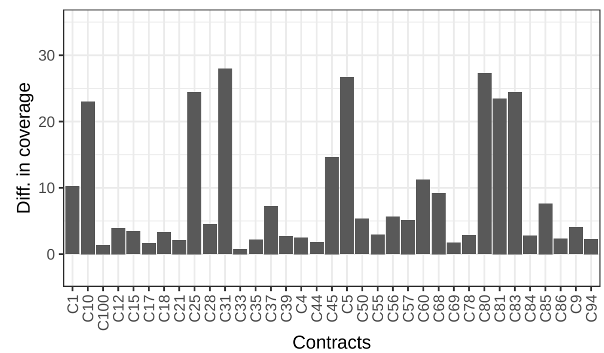


Fig. 4: Absolute difference in line coverage for *DynaMOSA* with and without the improved guidance

A. Result for RQ1: Structural coverage

Table II shows the statistical results for the structural coverage achieved by *DynaMOSA* with the proposed approach, compared to *DynaMOSA* without it, on the *Solidity* smart contracts in the benchmark. *#Win* indicates the number of contracts for which the search algorithms with the improved guidance have a statistically significant improvement (p-value ≤ 0.05) over the algorithms without this guidance. *#Lose* indicates the number of contracts for which the proposed approach did not provide a statistically improvement (p-value > 0.05), and lastly, *#No diff.* indicates the number of contracts for which there is no statistical difference in the results between the search algorithms with and without the improved guidance. In addition, the *#Win* and *#Lose* columns also include the magnitude of the difference through the $\hat{A}_{12}$ effect size, classified in *Negligible* (N), *Small* (S), *Medium* (M), and *Large* (L).

From Table II, we can see that the proposed approach only provides a statistically significant improvement for branch coverage in very few cases (4). This result is as expected as without the additional information that the guidance provides, the search process falsely assumes that the branches containing the transaction-reverting statements are fully covered. Consequently, with the improved guidance, we can observe a statistically significant improvement in 37 and 35 contracts for transaction-reverting statement and line coverage, respectively. This indicates that without this guidance *DynaMOSA* cannot reach the code regions after these statements. For the transaction-reverting statement coverage, *DynaMOSA* im-

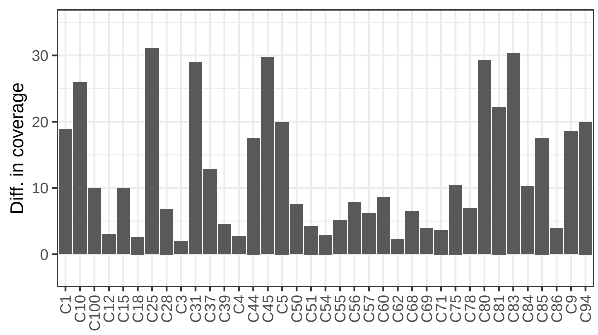


Fig. 5: Absolute difference in reverting statement coverage for *DynaMOSA* with and without the improved guidance

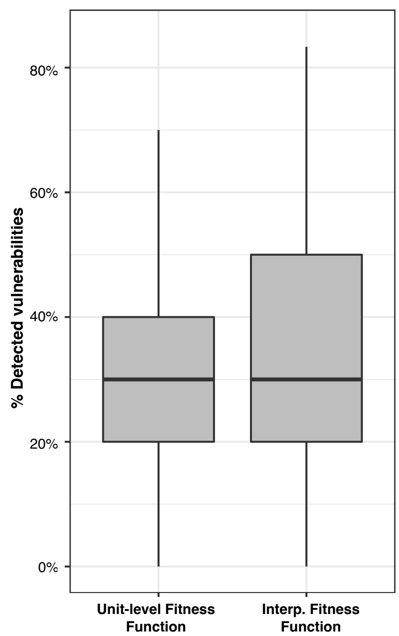


Fig. 6: Vulnerability detection results for *DynaMOSA*

proves with a large magnitude for 35 contracts and medium for 2 contracts. For line coverage, *DynaMOSA* improves with a large magnitude for 29 contracts, medium for 4 contracts, and small for 2 contracts.

Figs. 4 and 5 show the absolute difference in the average (mean) line and transaction-reverting statement coverage achieved by *DynaMOSA* with the improved guidance, compared to *DynaMOSA* without this guidance, for the significant cases. The proposed approach on average improves the line coverage by +8.66 %, with a maximum improvement of +27.97 % for `GreenMarkTrust` (id = C31), and the transaction-reverting statement coverage by +12.29 %, with a maximum improvement of +31.07 % for `MARVELCOIN` (id = C25).

B. Result for RQ2: Vulnerability Detection

Fig. 6 shows the percentage of vulnerabilities that were detected by *DynaMOSA* when comparing the unit-level fitness

```
function burnFrom(address _from, uint256 _value) public ... {
  // Check if the targeted balance is enough
  require(balanceOf[_from] >= _value);

  // Check allowance
  //require(_value<=allowance[_from][msg.sender])); <- SECURE
  require(_value>allowance[_from][msg.sender]);  // <- VULNER.

  // Subtract from the targeted balance
  balanceOf[_from] -= _value;

  // Subtract from the sender's allowance
  allowance[_from][msg.sender] -= _value;
  totalSupply -= _value;

  // Update totalSupply
  emit Burn(_from, _value);
  return true;
}
```

Listing 2: Vulnerable variant for the contract `INS.sol`

```
it('test for INS', async () => {
  const INS0 = await INS.new(BigInt("139"), "fKQs..",
          "lihM...", {from: accounts[2]});

  const bool0 = await INS0.burn.call(BigInt("1361"),
          {from: accounts[2]});

  assert.equal(bool0, true)

  await expect(
          INS0.burnFrom.call(accounts[1], BigInt("1212"))
          ,
          {from: accounts[2]})
        ).to.be.rejectedWith(Error);
});
```

Listing 3: Generated test case that detects the vulnerability (Listing 2) for the contract `INS.sol`

function to the proposed interprocedural one. As we can observe, there is no or small differences in the minimum and first quartile in the box-plots. That means that for 25 % of the contract there is no difference in the vulnerability detection capability. This is also in line with the results we observe in RQ1, considering that covering the line and transaction-reverting statement is a prerequisite to reach the vulnerability. However, we observe larger differences in the second and third quartiles, as well as in the maximum value.

In particular, we observe that the percentage of captured vulnerabilities achieved by *DynaMOSA* increases by 2 % in the 2^{nd} quartile and 8 % in the 3^{rd} quartile, as depicted in Fig. 6. For the contracts with a difference in the number of captured vulnerabilities, our approach improves on average by 17 %. The largest improvement is obtained for `HTDD_contract` with an increase in the number of vulnerabilities captured of 38 %. We also report a *moderate* positive Pearson's r correlation between the increases in the vulnerability detection capability and the increases in line coverage (r=0.48, p-value=<0.01) and transaction-reverting statement coverage (r=0.40, p-value=<0.01) achieved when using the improved guidance with *DynaMOSA*. We applied the Pearson's r correlation coefficient since the difference in these metrics are normally distributed.

To provide a practical example, let us consider the vulnerability reported in line 7 of Listing 2 for the contract `INS`. This vulnerability is caused by changing the condition (from <= to >) in the second `require` statement. The vulnerability is captured by *DynaMOSA* when using the improved guidance but remains undetected when our approach is not applied. The test case that captures the vulnerability is reported in Listing 3. This test case covers both require statements in the function (line 3 and 7) and asserts the reverting operation of the EVM in line 7, *i.e.,* if the transaction-reverting statement is not satisfied, all performed transactions are reverted. The test correctly captures the transaction-reverting statement and fails (via the expected `to.be.rejectedWith(Error)` code) when such a condition is modified. Instead, *DynaMOSA* without the improved guidance could not even reach the `require` in line 7 as it did not manage to satistfy the condition of the `require` statement in line 3.

VI. Discussion

Our experiment empirically shows that applying state-of-the-art test case generation approaches cannot effectively detect vulnerabilities (or produce structural coverage) without treating all constructs of the language to be tested as first class citizins. The success of search-based software testing is, in practice, dependent on many components, including the ability of the search algorithm to get insight on all aspects of the program execution through the fitness function. Our empirical study shows the importance of modelling these language-level constructs in the fitness function.

The benefits of this approach are not only applicable for test case generation, but also to fuzzing approaches and have the potential to improve the testing landscape for *Solidity* smart contracts. Based on a preliminary study, our approach can improve line coverage for *sFuzz* [3], a state-of-the-art fuzzer, by on average +8.42 %, with a maximum improvement of +31.76 %, and the transaction-reverting statement coverage by +13.08 %, with a maximum improvement of +33.09

Additionally, this approach does not only apply to *Solidity* but can be generalized to any programming language with explicit contracts or declarative input validation rules. For example, Java makes use of annotations (*e.g.,* @NotNull) that help control contracts throughout method hierarchies. In general, interprocedural analysis can benefit testing programs that use design by contract constructs.

This paper focusses on *Solidity* as contracts can not be updated once they are deployed, increasing the importance of detecting vulnerabilities related to the transaction-reverting statements as early as possible [11].

VII. Threats to Validity

Construct Validity: The study makes use of well-established metrics in software testing to compare the different approaches: structural coverage (*i.e.,* branch, line) and vulnerability detection capability (how well do the generated tests detect vulnerabilities). A time budget is used as the stopping condition for the search algorithm instead of the number of evaluations. Given that the approaches compared in the study use different genetic operators, with a different execution overhead, search time is a fairer metric for budget allocation [30].

External Validity: To make sure that the study's results can be generalized, the benchmark used to evaluate it has to contain a diverse set of smart contracts of a wide range of complexities. We created a benchmark with 100 real-world smart contracts gathered from *etherscan.io*. This benchmark contains contracts with different sizes and cyclomatic complexities.

Conclusion Validity: Evolutionary algorithms make use of randomness to search the problem space. To minimize the risk that the results were caused by favourable randomness, we have performed the experiment 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [58] and analyzed the possible impact of different random seeds on our results. We used two non-parametric tests: the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

VIII. Conclusions and Future Work

Previous studies focused on coverage-oriented heuristics to test and fuzz *Solidity* smart contract. However, they do not directly handle transaction-reverting statements, a vital mechanism within *Solidity* to protect the contract against invalid requests. To overcome this limitation, we proposed a novel fitness function based on interprocedural dependency analysis and context-sensitive instrumentation to exercise and test directly these statements.

We implemented the novel fitness function in the *SynTest-Solidity* [12] testing framework. The framework implements the state-of-the-art testing algorithm, called *DynaMOSA* [13], guided by well-established unit-level fitness functions. Our results show that our interprocedural fitness function improves the number of the vulnerabilities detected as well as structural coverage compared to the state-of-the-art unit-level alternative. Our results suggest that our approach has a wide range of applications being able to improve both test case generation and fuzzing algorithms.

Given our promising results, there are multiple potential directions for future work, including (i) a topology study on common transaction-reverting statement vulnerabilities and their prevalence, and (ii) constructing a build pipeline for smart contracts to prevent vulnerable contracts to go live.

References

[1] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[2] V. Dwivedi, V. Deval, A. Dixit, and A. Norta, "Formal-verification of smart-contract languages: A survey," in *International Conference on Advances in Computing and Data Sciences*. Springer, 2019, pp. 738–747.

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

[4] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," *arXiv preprint arXiv:2005.12156*, 2020.

[5] 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.

[6] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer 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. 1398–1409.

[7] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[8] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.

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

[10] "Expressions and control structures." [Online]. Available: https://docs.soliditylang.org/en/v0.8.7/control-structures.html#error-handling-assert-require-revert-and-exceptions

[11] L. Liu, L. Wei, W. Zhang, M. Wen, Y. Liu, and S.-C. Cheung, "Characterizing transaction-reverting statements in ethereum smart contracts," *arXiv preprint arXiv:2108.10799*, 2021.

[12] M. Olsthoorn, D. Stallenberg, A. van Deursen, and A. Panichella, "Syntest-solidity: Automated test case generation and fuzzing for smart contracts," in *The 44th International Conference on Software Engineering-Demonstration Track*. IEEE/ACM, 2022.

[13] 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, Feb 2018.

[14] ——, "Reformulating branch coverage as a many-objective optimization problem," in *Proceedings of the International Conference on Software Testing, Verification and Validation, (ICST'15)*, Graz, Austria, 2015, pp. 1–10.

[15] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[16] M. Olsthoorn, A. van Deursen, and A. Panichella, "Replication package of "guiding automated test case generation for transaction-reverting statements in smart contracts"." [Online]. Available: https://doi.org/10.5281/zenodo.6787666

[17] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" in *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[18] J. Ye, M. Ma, Y. Lin, Y. Sui, and Y. Xue, "Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2020, pp. 274–275.

[19] 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*, 2020.

[20] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.

[21] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, 2021.

[22] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th {USENIX} security symposium ({USENIX} security 19)*, 2019, pp. 1591–1607.

[23] M. Soltani, A. Panichella, and A. Van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1294–1317, 2018.

[24] J. Zhu, K. Hu, M. Filali, J.-P. Bodeveix, and J.-P. Talpin, "Formal verification of solidity contracts in event-b," 2020.

[25] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from ocl constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[26] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 3–45.

[27] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[28] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Towards integration-level test case generation using call site information," *CoRR*, vol. abs/2001.04221, 2020. [Online]. Available: https://arxiv.org/abs/2001.04221

[29] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[30] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[31] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, no. 3, pp. 611–639, 2015.

[32] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[33] A. Arcuri and J. P. Galeotti, "Handling sql databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.

[34] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[35] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018.

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

[37] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[38] D. W. Binkley, M. Harman, and K. Lakhotia, "Flagremover: a testability transformation for transforming loop-assigned flags," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, pp. 1–33, 2011.

[39] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 440–451.

[40] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach," *ACM SIGSOFT software engineering notes*, vol. 29, no. 4, pp. 108–118, 2004.

[41] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper, "Testability transformation–program transformation to improve testability," in *Formal methods and testing*. Springer, 2008, pp. 320–344.

[42] X. Devroey, S. Panichella, and A. Gambi, "Java unit testing tool competition: Eighth round," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 545–548.

[43] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "Sbst tool competition 2021," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 20–27.

[44] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[45] M. Rai, "Robust optimal aerodynamic design using evolutionary methods and neural networks," in *42nd AIAA Aerospace Sciences Meeting and Exhibit*, 2004, p. 778.

[46] K. Deb and D. Deb, “Analysing mutation schemes for real-parameter genetic algorithms,” *International Journal of Artificial Intelligence and Soft Computing*, vol. 4, no. 1, pp. 1–28, 2014.
[47] L. C. Briand, Y. Labiche, and M. Shousha, “Using genetic algorithms for early schedulability analysis and stress testing in real-time systems,” *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006.
[48] H. Eskandari and C. D. Geiger, “A fast pareto genetic algorithm approach for solving expensive multiobjective optimization problems,” *Journal of Heuristics*, vol. 14, no. 3, pp. 203–241, 2008.
[49] H. G. Cobb and J. J. Grefenstette, “Genetic algorithms for tracking changing environments.” Naval Research Lab Washington DC, Tech. Rep., 1993.
[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast elitist multi-objective genetic algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.
[51] A. Panichella and U. R. Molina, “Java unit testing tool competition-fifth round,” in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2017, pp. 32–38.
[52] A. Arcuri, “Test suite generation with the many independent objective (mio) algorithm,” *Information and Software Technology*, vol. 104, pp. 195–206, 2018.
[53] L. Alt and C. Reitwiessner, “Smt-based verification of solidity smart contracts,” in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 376–388.
[54] P. Chapman, D. Xu, L. Deng, and Y. Xiong, “Deviant: A mutation testing tool for solidity smart contracts,” in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 319–324.
[55] W. J. Conover, *Practical nonparametric statistics*. John Wiley & Sons, 1998, vol. 350.
[56] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *2011 33rd international conference on software engineering (ICSE)*. IEEE, 2011, pp. 1–10.
[57] A. Vargha and H. D. Delaney, “A critique and improvement of the cl common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[58] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.