

# Causes and Effects of Fitness Landscapes in System Test Generation: A Replication Study

Omur Sahin<sup>1</sup>, Man Zhang<sup>2\*</sup>, Andrea Arcuri<sup>3,4</sup>

<sup>1</sup>Erciyes University, Kayseri, Türkiye.

<sup>2</sup>Beihang University, Beijing, China.

<sup>3</sup>Kristiania University of Applied Sciences, Oslo, Norway.

<sup>4</sup>Oslo Metropolitan University, Oslo, Norway.

\*Corresponding author(s). E-mail(s): [manzhang@buaa.edu.cn](mailto:manzhang@buaa.edu.cn);  
Contributing authors: [omur@erciyes.edu.tr](mailto:omur@erciyes.edu.tr); [andrea.arcuri@kristiania.no](mailto:andrea.arcuri@kristiania.no);

## Abstract

Search-Based Software Testing (SBST) has seen several success stories in academia and industry. The effectiveness of a search algorithm at solving a software engineering problem strongly depends on how such algorithm can navigate the *fitness landscape* of the addressed problem. The fitness landscape depends on the used fitness function. Understanding the properties of a fitness landscape can help to provide insight on how a search algorithm behaves on it. Such insight can provide valuable information to researchers to being able to design novel, more effective search algorithms and fitness functions tailored for a specific problem. Due to its importance, few fitness landscape analyses have been carried out in the scientific literature of SBST. However, those have been focusing on the problem of *unit test* generation, e.g., with state-of-the-art tools such as EvoSuite. In this paper, we *replicate* one such existing study. However, in our work we focus on *system test* generation, with the state-of-the-art tool EVOMASTER. Based on an empirical study involving the testing of 23 web services, this enables us to provide valuable insight into this important testing domain of practical industrial relevance. Our results indicate that fitness landscapes are largely dominated by neutral regions (e.g., plateaus), which make the search process challenging. We observe that the presence of information content in the landscape can improve search guidance, while boolean flags are a primary contributor to neutrality. These findings confirm prior results in unit testing but also reveal system-level differences, particularly in how branch types impact search effectiveness. These insights suggest the need for improved fitness functions, testability transformations, and search operators tailored to system-level testing.

**Keywords:** Replication, Fitness Landscape, SBST, REST, API

## 1 Introduction

Many modern applications are built using web services such as REST (Fielding (2000)), SOAP (Curbera et al (2002)), or GraphQL (Quiña-Mera et al (2023)). In large and complex enterprise applications, these are structured into individual web services through a microservice architecture (Newman (2021)). This method minimizes the development and maintenance costs associated with monolithic applications and aims to create more robust solutions. Major companies such as Netflix, Uber, eBay, Amazon, and Nike have widely adopted this approach in the industry (Rajesh (2016)).

However, testing web services presents many challenges due to their complexity (Bozkurt et al (2013); Canfora and Di Penta (2009)). In particular for REST APIs, several techniques have been proposed in the literature (Golmohammadi et al (2023c)). Various tools have been proposed for fuzzing web services in recent years, including ARAT-RL (Kim et al (2023)), bBOXRT (Laranjeiro et al (2021)), DeepREST (Corradini et al (2024)), EVOMASTER (Arcuri et al (2025d)), Morest (Liu et al (2022)), ResTest (Martin-Lopez et al (2021)), RestCT (Wu et al (2022)), Restler (Atlidakis et al (2019)), RestTestGen (Viglianisi et al (2020)) and Schemathesis (Hatfield-Dodds and Dygalo (2022)). Apart from EVOMASTER, all these tools are *black-box*, unable to analyze the source code of the tested APIs to achieve better results. Although black-box testing has its place in industry (Arcuri et al (2025a)), results with *white-box* testing, when applicable, can be much better (Zhang and Arcuri (2023)).

A key component for white-box EVOMASTER, that enables it to achieve significantly better results, is the use of Search-Based Software Testing (SBST) techniques, whose applicability and effectiveness have been demonstrated in various studies (Ali et al (2010); Fraser and Arcuri (2013a); Harman et al (2012); Mao et al (2016)). In SBST, software test generation is cast into a search problem, which can then be tackled with various search algorithms, such as Genetic Algorithm (GA) (Holland (1992)), Many Objective Sorting Algorithm (MOSA) (Panichella et al (2018)), or Many Independent Objective (MIO) (Arcuri (2017)). These algorithms are guided by a *fitness function* to try to find the best solution in the *search space* of all possible test cases for the given problem.

Several studies in the literature of SBST demonstrate that these algorithms can effectively help test case generation while the fitness functions accurately direct the algorithm. However, existing fitness functions are not able to guide the algorithms in covering all test targets, such as covering all branches (Campos et al (2018); Albunian et al (2020)). In addition to familiar challenges, such as complex parameters that complicate covering all targets during a search, the lack of a comprehensive understanding of search behavior further complicates the identification of the factors that contribute to these difficulties (Albunian et al (2020)). One method used to understand search behavior is *fitness landscape analysis* (Zou et al (2022)), which helps to understand the difficulties underlying the problems. A deep understanding

of the problem’s search space enhances insight into algorithm behavior and helps in improving algorithms’ problem-solving capabilities. While prior studies have focused on fitness landscapes in unit-level test generation, system-level testing introduces additional challenges. These include the orchestration of multiple components, complex input-output interactions, and hidden system states. Understanding fitness landscapes at the system level is therefore crucial for assessing whether insights from unit testing can scale and would still be the same. System testing has wide application and use in industry. By extending the analysis to system-level testing, this work aims to fill this gap and provide a broader empirical foundation for enhancing SBST techniques in practice.

In this study, EvoMaster exercises the system as a whole, including database interactions, internal service coordination, and communication with external components deployed via Docker containers. The generated tests include not only API calls but also actions such as SQL insertions and setups of mock objects for external services. These aspects introduce additional behavioral complexity that cannot be captured at the unit level. Therefore, we adopt the term *system-level testing* rather than *API testing*, as it more accurately reflects the scope and depth of the testing performed in this study. However, how our results would generalize to other types of system-level testing (e.g., testing through GUI interfaces) is an important matter for future studies.

The two primary properties of fitness landscapes that significantly impact the optimization process are *ruggedness* and *neutrality* (Malan and Engelbrecht (2013)). The interaction between these properties has inspired the creation of various techniques to analyze their structure. The main purpose of this study is to explore the characteristics of the search space involved in system-level test generation by analyzing the fitness landscape. To achieve this, we measured six different proxy metrics commonly used to assess system tests’ ruggedness and neutrality properties. These calculations were carried out using the EVOMASTER tool to perform a random walk in the search space. Results and analyses were conducted on a total of 23 Systems Under Test (SUTs) featuring REST, GraphQL, and RPC applications.

This work aims at addressing the following research questions:

**RQ1:** What are the characteristics of the fitness landscape for system-level test case generation?

**RQ2:** How do the fitness landscape characteristics, like neutrality and ruggedness, relate in test generation?

**RQ3:** How are differences in fitness landscape characteristics and search outcomes associated with different types of branches?

**RQ4:** What are the differences in how the characteristics of the fitness landscape affect search-based testing in unit test generation versus system-level test generation?

This work is a *replication study* of a fitness landscape analysis of *unit test generation* (Albunian et al (2020)). However, in this study the difference is that we aim at analyzing the fitness landscape of *system-level test generation* to evaluate the complexity of branches. This research aims at improving SBST approaches, by collecting insights into the impact of neutrality factors, such as challenging preconditions, and boolean flags. The key research questions address how these neutrality factors

influence the search process and how enhancing fitness functions or *testability transformations* (Harman et al (2004); Arcuri and Galeotti (2021a)) can mitigate these challenges. This study employs the MIO algorithm to determine branch difficulty, followed by a Random Walk analysis for deeper insights. This work contributes to automated test generation methods to improve software reliability and maintainability.

This article’s contributions can be summarized as follows:

- We conduct the first fitness landscape analysis in the literature for system-level test generation of web services (REST, GraphQL, and Thrift).
- We replicate and extend previous research on unit testing fitness landscapes, to apply it to a more complex system testing context.
- We present an empirical study involving 23 web services, measuring six key landscape metrics (ruggedness and neutrality) through random walks using EVOMASTER.
- We examine the effect of code attributes, including complex preconditions and boolean flags, on the fitness landscape and search effectiveness.
- We analyze the correlation between neutrality and ruggedness properties and search difficulty, and suggest directions for improving Search-Based Software Testing (SBST) techniques through enhanced fitness functions and testability transformations.

The paper is organized as follows. Section 2 provides background information on SBST, on EVOMASTER and on the used algorithms in our study. Related work is discussed in Section 3. Section 4 presents the details of our replicated fitness landscape analysis. Empirical analyses are presented in Section 5. Our observations on these results follow in Section 6. Threats to validity are discussed in Section 7. Finally, Section 8 concludes the paper.

## 2 Background

### 2.1 Search-Based Software Testing

Testing is a crucial but challenging aspect of software development, often seen by developers as a tedious task, particularly for complex applications. Furthermore, besides being possibly tedious and expensive, *manual* testing can lack rigor and not be systematic, which can lead to poor verification effectiveness. For these reasons, lot of research has been carried out on how to test software *automatically* (Bertolino (2007)).

A common approach to ease this process is to use *random* test case generation (Duran and Ntafos (1984); Arcuri et al (2012)), which is suitable for simple cases but typically fails to provide effective coverage for complex scenarios. Search-Based Software Testing (SBST) approaches have been developed to address these complex scenarios effectively. In SBST, software testing is cast into an optimization problem, which can then be tackled with search algorithms (e.g., Genetic Algorithms (Holland (1992))). Many applications using the SBST approach have been developed (e.g., (McMinn (2004); Harman and Jones (2001); Harman et al (2012); Ali et al (2010))). Tools like EvoSuite (Fraser and Arcuri (2011)) and Pynguin (Lukasczyk and Fraser

```

1 public static void foo(int x){
2     if(x == 42){
3         print("The answer to the ultimate question of life, the universe, and
4             everything");
5     } else {
6         print("Wrong. Try again");
7     }
8 }

```

**Fig. 1:** Code snippet of a simple numerical function with a single `if` statement.

(2022)) for unit testing, Sapienz (Mao et al (2016)) for mobile testing, and EvoMaster (Arcuri et al (2025d)) for Web API testing use search algorithms to systematically explore input domains and maximize objectives like code coverage and fault detection.

The optimization process in SBST is driven by a *fitness function* that evaluates test cases based on how well they achieve testing goals. Techniques like evolutionary algorithms are commonly applied, where operators such as *crossover* and *mutator* help refine test cases over successive generations, improving their ability to explore the software’s behavior. This evolutionary approach allows SBST to navigate the *fitness landscape*, effectively identifying test cases for complex and hidden faults in the code.

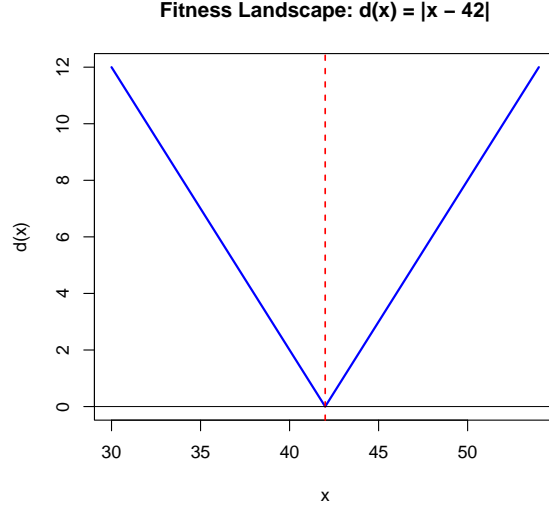
To achieve better results, various heuristics are used to “smooth” the search landscape, providing guidance to the search algorithm in finding an optimal solution. In the context of white-box testing, a well-known technique introduced in the 1990s by Korel (Korel (1990)) is called *Branch Distance*. This technique was initially developed to address predicates involving numerical comparisons (e.g.,  $a < b$ ) and was later refined to handle logical operators (Gallagher and Narasimhan (1997)) (e.g., AND and OR) as well as string comparisons (Alshraideh and Bottaci (2006)).

The branch distance enables the application of optimization methods by providing a smoother fitness landscape. The code snippet shown in Figure 1 is divided into two branches. If we consider here the “then” branch related to satisfying the constraint  $x==42$ , the branch distance can be defined as  $d(x) = |x - 42|$ . So, the code snippet has a landscape as given in Figure 2, and the algorithm performs its search in this search space of integer inputs. This is a rather straightforward search landscape. Small modifications to the inputs (e.g., via mutation operator) have clear gradient (i.e., fitness scores improve) towards the *global optimum*, without any risk of being stuck in *local optima* or in *fitness plateaus*.

However, in the general case, generating test cases can be complex due to combinatorial challenges, making the fitness landscape more intricate. To address this complexity, we use proxy methods for fitness landscape analysis, as explained in Section 4.

## 2.2 EvoMaster

EVOMASTER (Arcuri et al (2025d, 2021); Arcuri (2019)) is an open-source fuzzer for automated Web API fuzzing that supports black (Arcuri (2020)) and white-box (Arcuri and Galeotti (2020b)) testing modes. Black-box testing can be applied on APIs written in any programming language. For white-box testing, EVOMASTER is currently compatible with APIs developed in Java and Kotlin (Arcuri (2019)), although



**Fig. 2:** Fitness landscape of the code snippet in Figure 1.

in the past there was support for JavaScript/TypeScript (Zhang et al (2023b)) and C# (Golmohammadi et al (2023a)) (which are no longer maintained).

EvOMASTER generates test cases for Web APIs using evolutionary algorithms. It contains search methods, including the Many Independent Objective (MIO) (Arcuri (2018)) algorithm (which is set as the default), the Whole Test Suite (WTS) (Fraser and Arcuri (2013b)), and the Many-Objective Sorting Algorithm (MOSA) (Panichella et al (2018)). Additionally, to enhance API fuzzing, EvOMASTER has been improved with techniques such as testability transformations (Arcuri and Galeotti (2021b)), SQL handling (Arcuri and Galeotti (2020a)), adaptive hypermutation (Zhang and Arcuri (2021a)) and few others advanced white-box heuristics (Arcuri et al (2023a)).

EvOMASTER provides automated fuzzing for various types of APIs, including REST (Arcuri (2019); Zhang et al (2021); Zhang and Arcuri (2021b)), GraphQL (Belhadi et al (2023)), and RPC (Zhang et al (2023a)) APIs. Notably, it is the only fuzzer for Web APIs in the literature that supports white-box testing, using an automated method to capture runtime data such as SBST heuristics via bytecode manipulation (using the same approach as older SBST tools such as EvoSuite). EvOMASTER is a mature tool, downloaded thousands of times (Arcuri et al (2025d)) and used for example in Fortune 500 enterprises such as Meituan (Zhang et al (2023a, 2024)) and Volkswagen (Poth et al (2025); Arcuri et al (2025a)).

Due to all these features, in this study we selected EvOMASTER as the main tool for our fitness landscape analyses in the context of system test generation.

### 2.3 The Random Walk Algorithm

One essential tool for fitness landscape analysis is the concept of a walk (Pitzer and Affenzeller (2012)). Suppose we visualize a landscape similar to a real-world setting. In that case, the basic principle is to continually move a short distance from one solution candidate to a nearby one while closely monitoring the fitness progress. There are various methods such as random walk (RW), adaptive walk, reverse adaptive walk, uphill-down and neutral walk (Pitzer and Affenzeller (2012)). RW was used in this study. The RW algorithm is widely used for analyzing landscapes in large and complex problems. It starts from a randomly chosen point and makes random steps from there. The pseudo-code of the RW algorithm used in this study is given in Algorithm 1.

First, an individual is sampled randomly. Then, the same individual is mutated until the stopping criterion is met, and if there is an improvement in any target, it is added to the archive. Here, the stopping criterion is number of steps, i.e., the number of individual evaluations.

---

#### Algorithm 1: Pseudo-code of the RW Algorithm

---

**Input** : Stopping condition  $C$ , Fitness function  $\delta$ , Population size  $n$   
**Output** : Archive of optimized individuals  $A$

---

```

1  $p \leftarrow null$ 
2 while  $\neg C$  do
3   if  $p == null$  then
4      $p \leftarrow RandomIndividual()$ 
5   else
6      $p \leftarrow Mutate(p)$ 
7   end
8   foreach  $element\ k \in ReachedTargets(p)$  do
9     if  $NewTarget(k)$  then
10       $S \leftarrow S \cup T_k$ 
11    end
12     $T_k \leftarrow T_k \cup \{p\}$ 
13    if  $IsTargetCovered(k)$  then
14       $UpdateArchive(A, p)$ 
15       $S \leftarrow S \setminus T_k$ 
16    else if  $|T_k| > n$  then
17       $RemoveWorst(T_k, \delta)$ 
18    end
19  end
20 end

```

---

## 2.4 The Many Independent Objective Algorithm

The MIO algorithm (Arcuri (2017, 2018)) is designed specifically for white-box system-level test generation. It is the default search algorithm in EVOMASTER. The pseudo-code can be found in Algorithm 2. MIO is a genetic-based evolutionary algorithm inspired by the (1+1) Evolutionary Algorithm (EA) (Droste et al (2002)). This section will provide a brief discussion, but full details can be found in (Arcuri (2017, 2018)).

MIO is a multi-population algorithm designed for multiple test targets, with a separate population for each target. At the beginning of the search process, a random population is initialized based on a chromosome template using information obtained from the schema of the tested API.

---

### Algorithm 2: Pseudo-code of the MIO Algorithm Arcuri (2018)

---

**Input** : Stopping condition  $C$ , Fitness function  $\delta$ , Population size  $n$ ,  
Probability for random sampling  $P_r$ , Start of focused search  $F$

**Output** : Archive of optimized individuals  $A$

```

1  $S \leftarrow \text{SetOfEmptyPopulations}()$ 
2  $A \leftarrow \{\}$ 
3 while  $\neg C$  do
4   if  $P_r > \text{rand}()$  then
5      $p \leftarrow \text{RandomIndividual}()$ 
6   else
7      $p \leftarrow \text{SampleIndividual}(S)$ 
8      $p \leftarrow \text{Mutate}(p)$ 
9   end
10  foreach element  $k \in \text{ReachedTargets}(p)$  do
11    if  $\text{NewTarget}(k)$  then
12       $S \leftarrow S \cup T_k$ 
13    end
14     $T_k \leftarrow T_k \cup \{p\}$ 
15    if  $\text{IsTargetCovered}(k)$  then
16       $\text{UpdateArchive}(A, p)$ 
17       $S \leftarrow S \setminus T_k$ 
18    else if  $|T_k| > n$  then
19       $\text{RemoveWorst}(T_k, \delta)$ 
20    end
21  end
22   $\text{UpdateParameters}(F, P_r, n)$ 
23 end

```

---

During each iteration, MIO samples a target either randomly or by selecting from the population of targets that have not yet been covered. A test case consists of one or more actions representing HTTP calls used to test a web service. A new solution



is generated by applying only the mutation operator. Two types of mutations can be applied: structural mutation and internal mutation. Structural mutation changes the structure of the test case by adding or removing actions, while internal mutation changes the values of the genes of an action, such as flipping a boolean value between `true` and `false`.

The fitness value of each new test is calculated after it is sampled or mutated. If an improvement is observed in any target, it is recorded in the corresponding population, and the worst individuals are removed. When a test successfully covers a target, this individual is added to an archive. The relevant population is then reduced to one individual and no longer used for new sampling. At the end of the search, a test suite consisting of the best tests is generated.

### 3 Related Work

Regarding *fitness landscape analyses* in SBST, there are a few existing works. Waeselynck *et al.* (Waeselynck et al (2006)) analyze the search space’s ruggedness and size using a diameter metric to configure a simulated annealing algorithm for test generation. Lefticaru and Ipate (Lefticaru and Ipate (2008)) explore the local structure and size of the search space, similar to Waeselynck *et al.* (Waeselynck et al (2006)). They introduce a problem hardness measure called fitness distance correlation, which requires knowledge of the global optimum. Their analysis also evaluates various fitness functions for specification-based testing. Aleti *et al.* (Aleti et al (2017)) analyze the fitness landscape for generating test suites with EvoSuite using three metrics: population information content, negative slope coefficient, and change rate, primarily to assess solvability. Alunian *et al.* (Alunian et al (2020)) examine the landscape’s ruggedness and neutrality to generate tests with the Many-Objective Sorting Algorithm (MOSA) using EvoSuite. Vogel *et al.* conducted two different studies (Vogel et al (2019, 2021)) and improved the performance of their algorithms by conducting a fitness landscape analysis on Sapienz, which is used for mobile application testing. Besides software testing, in software engineering Aleti and Moser (Aleti and Moser (2015)) address the challenge of optimizing software architectures by focusing on the analysis of local structure, specifically ruggedness.

Another area of research examines the fitness landscape to determine whether it is elementary, or to construct elementary landscapes (Lu et al (2010); Chicano et al (2011)). Elementary landscapes represent a specific category of fitness landscapes that can guide the development of effective heuristics. These heuristics may be applicable to problems such as the next release problem Lu et al (2010) or test suite minimization (Chicano et al (2011)).

In the domain of system-level test generation, particularly for web services, several approaches have been proposed. Tools like ARAT-RL (Kim et al (2023)), bBOXRT (Laranjeiro et al (2021)), DeepREST (Corradini et al (2024)), Evo-MASTER (Arcuri et al (2025d)), Morest (Liu et al (2022)), ResTest (Martin-Lopez et al (2021)), RestCT (Wu et al (2022)), Restler (Atlidakis et al (2019)), RestTest-Gen (Viglianisi et al (2020)) and Schemathesis (Hatfield-Dodds and Dygalo (2022)) have employed black-box strategies to automatically generate test cases for APIs.

These tools typically lack access to source code and rely on specifications such as OpenAPI to guide the testing process. While effective in many scenarios, black-box techniques often struggle with achieving deep code coverage in complex systems.

Regarding *replication studies* in SBST, those are rare. Perhaps the most known example is the work on parameter tuning in SBST (Arcuri and Fraser (2013)), that was replicated in at least two studies (Sayyad et al (2013); Kotelyanskii and Kapfhammer (2014)). Other examples involve software effort estimation (Tawosi et al (2021)) and tool evolution (Golmohammadi et al (2023b)).

This study replicates the fitness landscape analysis approach of Albunian et al. (Albunian et al (2020)), originally conducted for unit-level test generation with EvoSuite. In contrast, our work focuses on system-level test generation using EvoMaster, targeting complete web services (REST, GraphQL, Thrift) instead of isolated classes. Furthermore, while Albunian et al. analyzed 331 classes for unit tests, we examine full APIs, dealing with complex interactions and stateful behaviors. In unit testing, tests are generated with direct access to code internals, such as branches and methods, making coverage easier to achieve. In system-level testing, interactions occur only via external APIs (e.g., `GET /users/1`), and internal functions are not directly accessible. Even if coverage data is available through instrumentation, reaching specific code paths requires navigating API-level constraints, making test generation significantly harder. Thus, while adopting a similar analysis methodology, our study extends the investigation to the more challenging context of system-level testing.

## 4 Fitness Landscape Analysis

The fitness function represents the objectives of an optimization problem in evolutionary computing and swarm intelligence algorithms. Fitness values are compared based on the principles of maximization or minimization during iterative processes, and these values are calculated according to the fitness function of the specific problem. Therefore, if the characteristic structure of an optimization problem can be understood in depth, the relationship of this problem to the algorithm behavior can also be understood more easily. Thus, this deep understanding can also help the algorithms’ ability to solve challenges effectively (Zou et al (2022)).

If we define the fitness landscape formally (Albunian et al (2020); Zou et al (2022)), it can be represented as  $(X, N, f)$ , where  $X$  denotes the set of potentially feasible solutions, and  $N$  is the neighborhood operator on  $X$  (e.g., a mutation operator in evolutionary algorithms).  $f$  is a fitness function ( $f : X \rightarrow \mathbb{R}$ ) that maps each genotype to a numerical fitness value. A fitness landscape is characterized by various features (Malan and Engelbrecht (2013)). Among these features, *ruggedness* and *neutrality* significantly influence the ability to find optimal solutions.

*Ruggedness* is one of the key characteristics of a fitness landscape. If a fitness landscape contains multiple local optima along with a single, isolated global optimum, and the fitness values of neighboring individuals are less correlated, it is considered “rugged”. In this scenario, finding the optimal solution can be challenging because an algorithm may become trapped in local optima.

Another important characteristic to consider is *neutrality*. When there are generally equal values throughout the search space, meaning the search surface consists of plateaus, ruggedness alone may not be sufficiently descriptive. In these situations, the fitness value can remain constant for extended periods of the search. When a mutation occurs in a neutral fitness landscape, it can lead to a change in position on the fitness map without affecting the fitness value. In this case, a neighboring solution of  $y$  is called a neutral neighbor if  $f(x) = f(y)$  at point  $y$ .

In this study, we calculate six metrics, including *Autocorrelation (AC)*, *Neutrality Distance (ND)*, *Neutrality Volume (NV)*, *Information Content (IC)*, *Partial Information Content (PIC)*, and *Density-basin Information (DBI)*. These are the same metrics used in the study we replicate (Albunian et al (2020)), where they were employed to assess the ruggedness and neutrality characteristics of the fitness landscape in unit test generation. These six metrics will be presented and briefly discussed in the following subsections. For more in-depth details on these metrics (including their motivation and justification), the interested reader is referred to Vassilev et al (2000) and Pitzer and Affenzeller (2012).

To calculate these metrics, as done in the literature, there is the need to use a *random walk*. In a random walk, a solution from  $X$  is randomly chosen (i.e., random starting point), and then the neighboring operator  $N$  (e.g., the mutation operator in EVOMASTER) is applied up to  $k$  times (e.g.,  $k = 1000$ ), each time computing the fitness scores with  $f$ , and keeping track of all of them (as needed to compute those six metrics).

An additional calculation is required to calculate the DBI, IC and PIC metrics (Vassilev et al (2000)) to obtain more advanced information. This measurement is calculated using fitness value sequences. However, instead of directly using a fitness value, a string expression is generated through a transformation, and the calculations are then performed based on this expression. Initially, each sequence of fitness values ( $\{f_t\}_{t=1}^k$ ) are transformed into a series of fitness changes using Eq. 1.

$$\Delta \{f_t\}_{t=1}^k := \{f_t - f_{t-1}\}_{t=2}^k \quad (1)$$

Next, a string expression is generated according to Eq. 2. In this context, the string expression can be defined as  $S(\epsilon) = s_1, s_2, s_3, \dots, s_k$ , where each expression is denoted as  $s_i \in \{\bar{1}, 0, 1\}$ .

$$s_i = \begin{cases} \bar{1}, & \text{if } x < -\epsilon \\ 0, & \text{if } |x| \leq \epsilon \\ 1, & \text{if } x > \epsilon \end{cases} \quad (2)$$

where  $x$  represents the changes in fitness. The parameter  $\epsilon$  is a real number from the interval  $[0, l_k]$ , and  $l_k$  represents the length of the interval of fitness values obtained by the random walk. The  $\epsilon$  parameter adjusts the landscape’s sensitivity. By changing  $\epsilon$ , one can “zoom in and out” to observe the same walk with varying levels of detail.

## 4.1 Autocorrelation

The AC is used to measure the correlation between two individuals. This correlation measurement is calculated for two individuals that differ by a step count of  $i$ . The calculation is made using Eq. 3. In this equation,  $s$  is the step size,  $f_i$  denotes the

fitness value of the  $i$ -th individual, and  $\bar{f}$  is the average fitness value of all individuals. The range of values for  $r(s)$  is between  $-1$  and  $1$ . “*The landscape is more rugged when the AC value is close to 0 meaning that the individuals of the random walk are less correlated*” (Albunian et al (2020)).

$$r(s) = \frac{\sum_{i=1}^{k-s} (f_i - \bar{f})(f_{i+s} - \bar{f})}{\sum_{i=1}^k (f_i - \bar{f})^2} \quad (3)$$

## 4.2 Neutrality Distance

The ND measures neutrality in landscapes by identifying the longest neutral step number during a random walk where no fitness values change. In other words, it represents the largest  $t$  value in the equation  $f(x_1) = f(x_2) = \dots = f(x_t)$ . The range of values for ND is between 0 and 1, and it can be calculated using Eq. 4. The landscape is more neutral when the ND value is close to 1.

$$ND = \frac{t}{k} \quad (4)$$

## 4.3 Neutrality Volume

The NV is one of the metrics used to measure neutrality. It is calculated based on the number of areas with equal fitness values during a random walk. For instance, if we have a fitness value sequence  $f_t = 0.3, 0.3, 0.3, 0.2, 0.2, 0.7, 0.7$ , we can identify  $z = 3$  distinct regions corresponding to the fitness values of 0.3, 0.2, and 0.7. The NV value is calculated by dividing the number of distinct regions  $z$  by the total number of steps  $k$ , i.e.,  $NV = z/k$ . The value ranges from 0 to 1, and neutrality increases as it approaches 0.

## 4.4 Information Content

The IC metric is determined by analyzing the diversity within the string  $S(\epsilon)$  to assess the ruggedness of the landscape. It is calculated using the entropy of consecutive symbols that differ from each other, based on Eq. 5.

$$H(\epsilon) = - \sum_{p \neq q} P_{[pq]} \log_6 P_{[pq]} \quad (5)$$

The probabilities  $P_{[pq]}$  represent the frequencies of possible blocks  $pq$  of elements from the set  $[\bar{1}, 0, 1]$ , as defined by Eq. 6.

$$P_{[pq]} = \frac{n_{[pq]}}{n} \quad (6)$$

Where  $n_{[pq]}$  refers to the number of times each  $pq$  appears in the string  $S(\epsilon)$ . The value range of the  $H(\epsilon)$  is  $[0, 1]$ . As the number of peaks in the landscape increases (which can imply a higher ruggedness), the value of  $H(\epsilon)$  also increases.

**Table 1:** Interpretation for each of the six employed metrics, based on their values.

Name	Range	Low	Medium	High
AC	$[-1, +1]$	Neutral	Rugged	
ND	$[0, 1]$		Neutral	
NV	$[0, 1]$			
IC	$[0, 1]$		Rugged	
PIC	$[0, 1]$		Rugged	
DBI	$[0, 1]$		Rugged	

## 4.5 Partial Information Content

The PIC metric is designed to analyze the *modality* in the landscape, which is related to the number of peaks (i.e., local optima) in it. Modality is often correlated with ruggedness. To calculate this metric, first, all zero values in  $S(\epsilon)$  and all values equal to its preceding symbol are removed, and a new  $S'(\epsilon)$  is calculated. Then, the PIC is calculated using Eq. 7.

$$M(\epsilon) = \frac{\mu}{n} \quad (7)$$

Where  $\mu$  is the length of the  $S'$  string, and  $n$  is the length of the  $S$  string. The range of values for  $M(\epsilon)$  is between 0 and 1. “If the landscape path is maximally multimodal,  $M(\epsilon)$  is 1 as the string  $S'(\epsilon)$  is identical to  $S(\epsilon)$  (i.e.,  $S(\epsilon)$  cannot be modified). In contrast, the landscape path is flat when the  $M(\epsilon)$  is 0 as there are no slopes in the landscape path” (Albunian et al (2020)).

## 4.6 Density-basin Information

The DBI method evaluates the variety of flat areas within a landscape by focusing on the characteristics of smooth points. It achieves this by examining consecutive equal symbols in a given string  $S$ . In this analysis, the only relevant sub-blocks identified in the string are composed of pairs  $[00, 11, \bar{1}\bar{1}]$ . This metric can be calculated with Eq. 8.

$$h(\epsilon) = - \sum_{p=q} P_{[pq]} \log_3 P_{[pq]} \quad (8)$$

The range of values for  $h(\epsilon)$  is between 0 and 1 and “it decreases when the number of groups increases, i.e., the density of peaks becomes lower” (Vassilev et al (2000)). In other words, high values for DBI typically represent landscapes that are rugged and with low neutrality.

## 4.7 Summary

Each of these six metrics does measure some specific properties of the search landscape. These properties are all somehow related to the concepts of neutrality and ruggedness. Considering they have different range values, and different interpretations for the minimal and maximal values, to clarify them we summarize them in Table 1.

**Table 2:** Systems Under Test (SUTs) used in our empirical study.

<i>SUT</i>	<i>Type</i>	<i>#LOCs</i>	<i>#SourceFiles</i>	<i>#Endpoints</i>	<i>Language</i>
<i>catwatch</i>	REST	9636	106	14	Java
<i>cwa-verification</i>	REST	3955	47	5	Java
<i>genome-nexus</i>	REST	30004	405	23	Java
<i>gestaohospital-rest</i>	REST	3506	33	20	Java
<i>graphql-ncs</i>	GraphQL	548	8	6	Kotlin
<i>graphql-scs</i>	GraphQL	577	13	11	Kotlin
<i>languagetool</i>	REST	174781	1385	2	Java
<i>market</i>	REST	9861	124	13	Java
<i>ocvn-rest</i>	REST	45521	526	258	Java
<i>patio-api</i>	GraphQL	18048	178	20	Java
<i>pay-publicapi</i>	REST	34576	377	10	Java
<i>petclinic-graphql</i>	GraphQL	5212	89	15	Java
<i>proxyprint</i>	REST	8338	73	74	Java
<i>reservations-api</i>	REST	1853	39	7	Java
<i>rest-ncs</i>	REST	605	9	6	Java
<i>rest-news</i>	REST	857	11	7	Kotlin
<i>rest-scs</i>	REST	862	13	11	Java
<i>restcountries</i>	REST	1977	24	22	Java
<i>rpc-thrift-ncs</i>	Thrift	585	9	6	Java
<i>rpc-thrift-scs</i>	Thrift	772	14	11	Java
<i>scout-api</i>	REST	9736	93	49	Java
<i>session-service</i>	REST	1471	15	8	Java
<i>timbuctoo</i>	GraphQL	107729	1113	18	Java
Total		471010	4704	616	

Recall that *neutrality* is not necessarily the opposite of *ruggedness*. A landscape could be for example neutral and rugged at the same time.

## 5 Empirical Study

To evaluate the fitness landscape characteristics of system-level test case generation, we conducted an empirical study to answer the following research questions:

**RQ1:** What are the characteristics of the fitness landscape for system-level test case generation?

**RQ2:** How do the fitness landscape characteristics, like neutrality and ruggedness, relate in test generation?

**RQ3:** How are differences in fitness landscape characteristics and search outcomes associated with different types of branches?

**RQ4:** What are the differences in how the characteristics of the fitness landscape affect search-based testing in unit test generation versus system-level test generation?

### 5.1 Case Study

In the experiments, we used a total of 23 APIs consisting of REST, GraphQL, and RPC applications present in the EMB corpus (Arcuri et al (2023b)). EMB is a collection of Web APIs consisting of REST, GraphQL, and RPC APIs, which we have gathered and expanded annually with new additions since 2017. It also features the EVOMASTER drivers, needed to enable white-box fuzzing for all these APIs. These drivers are configurations used to specify how to start, stop and reset these SUTs. They are also

responsible to automatically instrument the bytecode of these SUTs when started, to be able to calculate different kinds of SBST heuristics such as the branch distances.

Table 2 shows some statistics on these 23 APIs, including the number of source files, lines of code, and number of endpoints. This collection comprises 23 SUTs: 16 using REST, 5 using GraphQL, and 2 using Thrift RPC. They contain 471010 lines of code across 4704 source files, with 616 endpoints in total. Note that these code statistics reflect only what is contained in the business logic of the APIs. Data from third-party libraries, like HTTP servers and libraries to access SQL databases, is not accounted for here.

EMB provides APIs of varying sizes and complexities from diverse domains, addressing a broad range of APIs essential for scientific experimentation. A full description of these APIs can be found at [Arcuri et al \(2023b\)](#). This resource includes the source code, build scripts, and links to the original repositories from which these APIs have been gathered over the years.

## 5.2 Experiment Settings

To address our research questions, we execute the RW and MIO algorithms. The RW algorithm (Section 2.3) employs a mutation operator to explore the landscape randomly, while the MIO algorithm (Section 2.4) is used to assess the difficulty of branches. We follow the same kind of procedure done in [Albunian et al \(2020\)](#), where MOSA (the default algorithm in EvoSuite) was used to assess the difficulty of the branches.

The RW and MIO algorithms were executed 30 times each, as done in [Albunian et al \(2020\)](#), which is a common practice in software engineering research. The stopping criterion for each run was set at 1000 steps, indicating 1000 individual evaluations. This is the same number of steps used in [Albunian et al \(2020\)](#), which is based on common practice in fitness landscape analysis research ([Barnett et al \(1998\)](#)).

Note that, technically, this would not be a fair comparison between algorithms, as each test case can have a different number of HTTP calls (e.g., randomly between 1 and 10 in the default settings of EVOMASTER). However, we are not comparing algorithms (e.g., to show that MIO is better than RW), but rather use them to study the characteristics of the fitness landscape.

The experiments carried out 2 Configurations  $\times$  23 SUTs  $\times$  30 Runs  $\times$  1000 Evaluations = 1380000 steps, i.e., 1.3M fitness evaluations, with each single step recorded individually for every branch. These experiments lasted  $\approx$  106 hours, generating around 65GB of data to analyze.

The work was carried out with EVOMASTER version 3.4.0, and other control parameters are the default. The  $\epsilon$  parameter for IC (Section 4.4), PIC (Section 4.5), and DBI (Section 4.6) used in the analyses was selected as 0. Thus, analyses were performed at maximum sensitivity level.

## 5.3 Comparison of MIO and RW

This section provides a foundational comparison of the MIO and RW algorithms. Understanding the performance of these algorithms can establish a basis for investigating and interpreting the answers to the research questions. Table 3 presents

**Table 3:** Comparison of the MIO and RW algorithms.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	MIO	RW	$\hat{A}_{12}$	p-value	MIO	RW	$\hat{A}_{12}$	p-value	MIO	RW	Ratio
<i>catwatch</i>	43.5	34.6	<b>0.12</b>	< 0.001	94.4	152.1	<b>0.10</b>	< 0.001	2541	8451	332.61
<i>cwa-verification</i>	43.5	36.4	<b>0.28</b>	0.003	2.4	0.8	<b>0.10</b>	< 0.001	4343	4174	96.11
<i>genome-nexus</i>	34.6	28.5	<b>0.15</b>	< 0.001	0.0	0.0	0.50	1.000	2795	2150	76.93
<i>gestahospital-rest</i>	35.2	17.9	<b>0.00</b>	< 0.001	20.5	9.4	<b>0.05</b>	< 0.001	4544	2400	52.83
<i>graphql-ncs</i>	82.1	50.7	<b>0.00</b>	< 0.001	6.2	4.6	<b>0.06</b>	< 0.001	4975	8000	160.81
<i>graphql-scs</i>	69.5	49.3	<b>0.00</b>	< 0.001	11.0	6.0	<b>0.00</b>	< 0.001	3123	8000	256.15
<i>language-tool</i>	9.3	5.4	<b>0.00</b>	< 0.001	5.5	4.3	<b>0.34</b>	0.023	1246	1340	107.47
<i>market</i>	36.6	34.4	<b>0.33</b>	0.023	44.3	24.8	<b>0.25</b>	< 0.001	2894	7920	273.62
<i>ocvn-rest</i>	20.5	14.1	<b>0.00</b>	< 0.001	431.4	68.0	<b>0.00</b>	< 0.001	1484	3736	251.76
<i>patio-api</i>	16.2	12.3	<b>0.22</b>	< 0.001	33.2	16.7	<b>0.00</b>	< 0.001	4967	8000	161.05
<i>pay-publicapi</i>	13.0	13.0	<b>0.08</b>	< 0.001	41.5	24.8	<b>0.00</b>	< 0.001	1597	2501	156.62
<i>petclinic-graphql</i>	44.2	25.7	<b>0.00</b>	< 0.001	19.9	7.3	<b>0.00</b>	< 0.001	5316	8000	150.49
<i>proxypoint</i>	27.6	9.6	<b>0.00</b>	< 0.001	122.0	19.2	<b>0.00</b>	< 0.001	1693	8407	496.60
<i>reservations-api</i>	53.1	45.0	<b>0.08</b>	< 0.001	106.0	7.8	<b>0.00</b>	< 0.001	1430	1773	123.92
<i>rest-ncs</i>	84.2	15.9	<b>0.00</b>	< 0.001	0.0	0.0	0.50	1.000	1000	1000	100.00
<i>rest-news</i>	58.9	36.7	<b>0.00</b>	< 0.001	1.8	0.3	<b>0.07</b>	< 0.001	3218	4122	128.11
<i>rest-scs</i>	62.6	11.7	<b>0.00</b>	< 0.001	3.1	18.8	<b>0.15</b>	< 0.001	1000	1000	100.00
<i>restcountries</i>	68.2	15.0	<b>0.00</b>	< 0.001	23.0	1.0	<b>0.00</b>	< 0.001	1000	1000	100.00
<i>rpc-thrift-ncs</i>	88.1	85.0	<b>0.03</b>	< 0.001	10.0	10.0	0.50	1.000	5945	4915	82.67
<i>rpc-thrift-scs</i>	71.7	74.5	<b>0.83</b>	< 0.001	3.0	3.0	0.50	1.000	5512	5064	91.88
<i>scout-api</i>	41.6	25.0	<b>0.00</b>	< 0.001	43.7	8.7	<b>0.00</b>	< 0.001	2666	7318	274.48
<i>session-service</i>	64.7	57.7	<b>0.28</b>	0.003	18.7	10.8	<b>0.04</b>	< 0.001	2467	1767	71.65
<i>timbuctoo</i>	21.1	18.5	<b>0.00</b>	< 0.001	24.7	8.6	<b>0.00</b>	< 0.001	2795	8000	286.25
Average	47.4	31.2	0.10		46.4	17.7	0.14		2980	4741	170.96
Median	43.5	25.7	0.00		19.9	8.6	0.05		2795	4174	128.11

comparison of the two algorithms. We follow the statistical guidelines from [Arcuri and Briand \(2014\)](#), reporting  $p$ -values of Mann-Whitney-Wilcoxon U tests and Vargha-Delaney standardized  $\hat{A}_{12}$  effect sizes. Results are compared in terms of line coverage and detected faults. To better understand the results, we also report the number of HTTP calls each algorithm execute, as each test case can have a different number of calls (from 1 to 10). The computation cost and duration of a fitness function evaluation is directly related to the number of HTTP calls done in it. In this case, RW takes longer, roughly 50% more time.

When the line coverage results are examined, it is evident that the MIO algorithm achieves better outcomes in all cases except *rpc-thrift-scs*, and all results are statistically significant. Regarding the number of detected faults, both algorithms exhibited the same performance in *genome-nexus*, *rest-ncs*, *rpc-thrift-ncs*, and *rpc-thrift-scs* problems. While MIO finds more faults in most problems, RW finds more faults than the MIO algorithm in the *catwatch* and *rest-scs* problems. The key result here is that according to the  $\hat{A}_{12}$  metric, the MIO algorithm detects more faults in all problems. However, in some outlier runs, the number of faults detected by the RW algorithm is very high (e.g., 2700 in *catwatch*, 202 in *rest-scs* in one of the runs), and this increased the average. These outlier cases suggest that RW, due to its purely random nature, may occasionally discover faults that the guided search of MIO fails to reach, possibly because MIO gets trapped in local optima or follows suboptimal paths. Nevertheless, based on the  $\hat{A}_{12}$  effect size metric, MIO consistently outperforms RW across all SUTs, indicating that while RW can occasionally achieve better results in isolated runs, MIO provides more reliable and systematic fault detection overall.

As MIO provides better results than RW on average, we can use these results to support the choice of MIO to analyze the difficulty of branches compared to RW.



**Table 4:** The number of branches reached and the number of branches that the algorithms reached but could not cover in any run.

SUT	#Reached	#Never Covered
<i>catwatch</i>	298	41
<i>cwa-verification</i>	96	13
<i>genome-nexus</i>	652	102
<i>gestaohospital-rest</i>	50	9
<i>graphql-ncs</i>	176	0
<i>graphql-scs</i>	244	11
<i>languagetool</i>	5083	1462
<i>market</i>	30	3
<i>ocvn-rest</i>	155	31
<i>patio-api</i>	18	4
<i>pay-publicapi</i>	14	1
<i>petclinic-graphql</i>	22	1
<i>proxyprint</i>	340	100
<i>reservations-api</i>	16	5
<i>rest-ncs</i>	168	0
<i>rest-news</i>	104	21
<i>rest-scs</i>	216	15
<i>restcountries</i>	212	10
<i>rpc-thrift-ncs</i>	168	0
<i>rpc-thrift-scs</i>	218	7
<i>scout-api</i>	279	41
<i>session-service</i>	14	1
<i>timbuctoo</i>	464	130
Total	9037	2008

## 5.4 Selected Branches For Analysis

Each branching statement/operation in the code has two possible outcomes: the constraint (i.e., a guarding condition that resolves to a boolean predicate) is satisfied and the “then” branch is followed, or it is not, and then the “else” branch is followed. Unless an exception is thrown when evaluating the guarding condition, one of the two paths (i.e., “then” or “else”) is necessarily taken. In a fitness evaluation a branching condition might be evaluated  $k > 1$  times, e.g., if in a loop (e.g., `for` and `while` loops) or if in a function called more than once. For calculating the fitness score, the highest value out of the  $k$  branch distance evaluations is taken in EVOMASTER.

Note that, in SBST tools such as EvoSuite and EVOMASTER that work on JVM bytecode, branch coverage is computed at the bytecode level, and not at the code level. For example, a single `if` statement in a Java class could result in many bytecode branch instructions, possibly one for each clause of the predicate in the guarding instruction. In other words, the “branch coverage” criterion in EvoSuite (used in [Albunian et al \(2020\)](#)) and in EVOMASTER (used in this study) is stronger, and more akin to the “all-clauses” coverage criterion.

To better clarify our analyses, let us introduce the concepts of *reached* and *covered* when discussing branches. If a test case generated during the search executes a branching statement, both the two resulting branches are marked as *reached*. If the predicate of that branch is resolved, then the “then” branch of the two will also be marked as *covered*. Otherwise, the “else” branch is marked as *covered*. If a branch is *covered*, then it implies that it was also *reached*. The inverse is not necessarily true. Furthermore,

some branches may not have been ever reached during the search. This happens for example in branches inside nested blocks, when the “then” branch is never entered and executed. In this case, these branches are marked as *never reached*. A *never reached* branch is implied to be *never covered*. However, a *never covered* branch could had been *reached*.

In our fitness landscape analyses, we only consider fitness scores for *reached* branches. All other *never reached* branches would have the same score of 0 in all runs, so they would provide no useful information for our analyses. Furthermore, we also excluded all *reached* but *never covered* branches. These could be branches that are infeasible, or too hard to cover branches. As their achieved data would not be enough to study their fitness landscape, those were excluded. We also excluded branches that are covered when the SUT starts and boots up, before any test cases is executed.

Table 4 provides details about the resulting analyzed branches of each SUT. Although the algorithms reach up to 9037 branches in the study, 2008 of these reached branches remain uncovered by any run or algorithm. Therefore, 7029 branches were used in the end for our fitness landscape analyses.

In the study we replicated, it is specified that 331 classes were employed for the analyses, although no details on the number of branches was provided in the text. By analyzing the processed data in their replication study,<sup>1</sup> it seems 3202 branches were used in the analyses (although we cannot be 100% sure of this number). If this is so, in our study we are looking at more than twice as many branches.

For each branch, the branch difficulty was assessed with the same approach as done in Alunian et al (2020), with four different labels. A branch is classified as “Easy” if both the MIO and RW algorithms achieve at least a 50% success rate (SR), i.e., if in both algorithms (total 60 runs) it was covered in at least 15 out of the 30 runs (per algorithm) of our experiments. Conversely, if their success rates for both algorithms fall below 50%, it is labeled “Hard”. If the success rate of the RW is under 50% and the MIO’s is above, it is labeled “Search”; if the situation is reversed, it is called “RW”.

Note that the data for MIO is only used to assess the difficulty of the branches. When computing the different metrics to study the characteristics of the fitness landscape, only the data for RW is used.

## 5.5 Results for the RQ1

Table 5 presents the results of the six fitness landscape measures applied to the 7029 branches in our analyses. The results are also visualized in Figure 3 and Figure 4. The data given in both figures is the average metric value (i.e., the arithmetic mean) of each branch over 30 runs for RW.

According to Table 5, the lowest AC value is 0.687, while the average is 0.865. These results indicate that the fitness landscape is highly correlated and smooth, with few peaks.

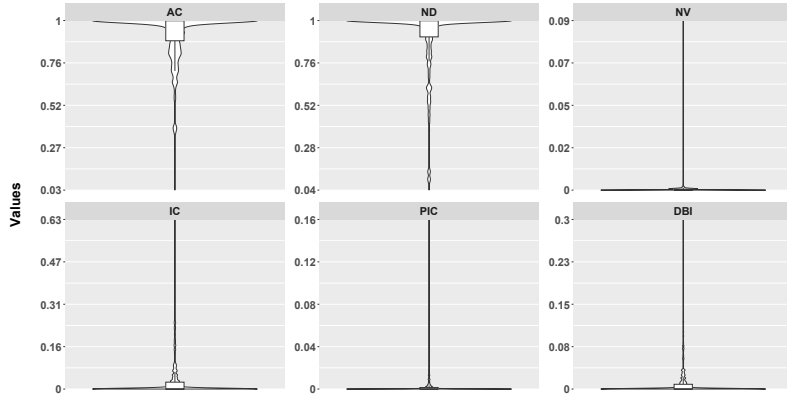
When the ND metric is examined, the lowest value is 0.240 only in the *rpc-thrift-ncs* problem, while the others are greater than 0.5. This shows that the majority of the steps are neutral, and the fitness landscape generally consists of plateaus. According to the average values, 0.82 of the steps consist of neutral steps.

---

<sup>1</sup><https://github.com/nasser-alunian/fitness-landscape-study>

**Table 5:** Mean six fitness landscape measurements for each SUT, gathered using the RW algorithm.

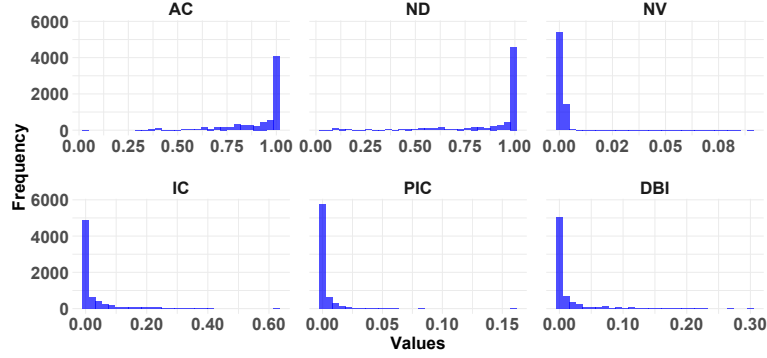
<i>SUT</i>	<i>AC</i>	<i>ND</i>	<i>NV</i>	<i>IC</i>	<i>PIC</i>	<i>DBI</i>
catwatch	0.836	0.774	0.002	0.046	0.002	0.017
cwa-verification	0.876	0.759	0.002	0.069	0.003	0.028
genome-nexus	0.884	0.874	0.002	0.034	0.002	0.012
gestaohospital-rest	0.804	0.932	0.002	0.015	0.001	0.005
graphql-ncs	0.736	0.785	0.006	0.116	0.012	0.058
graphql-scs	0.815	0.631	0.003	0.144	0.014	0.069
language-tool	0.984	0.994	0.001	0.002	0.000	0.001
market	0.933	0.984	0.001	0.006	0.000	0.002
ocvn-rest	0.714	0.889	0.001	0.026	0.003	0.009
patio-api	0.975	0.979	0.001	0.006	0.000	0.003
pay-publicapi	0.785	0.819	0.002	0.015	0.001	0.004
petclinic-graphql	0.890	0.718	0.001	0.083	0.005	0.035
proxyprint	0.977	0.980	0.001	0.004	0.000	0.002
reservations-api	0.874	0.848	0.001	0.039	0.002	0.016
rest-ncs	0.956	0.989	0.005	0.007	0.001	0.005
rest-news	0.796	0.812	0.002	0.053	0.007	0.023
rest-scs	0.953	0.927	0.004	0.033	0.005	0.018
restcountries	0.960	0.948	0.001	0.018	0.003	0.008
rpc-thrift-ncs	0.850	0.240	0.015	0.190	0.007	0.089
rpc-thrift-scs	0.847	0.511	0.005	0.089	0.003	0.036
scout-api	0.933	0.950	0.001	0.022	0.003	0.010
session-service	0.819	0.801	0.001	0.064	0.004	0.026
timbuctoo	0.687	0.711	0.002	0.082	0.009	0.034
Average	0.865	0.820	0.003	0.051	0.004	0.022
Median	0.874	0.848	0.002	0.034	0.003	0.016



**Fig. 3:** Results of the six fitness landscape measures applied on the 7029 branches. It is calculated by taking the average of 30 runs of RW for each branch.

When the NV metric is examined, similar results are seen. The highest value is 0.015, which shows that the landscape generally consists of neutral areas.

When the IC metric is examined, the highest value is 0.190, while the average is 0.051. In general, the values are quite close to 0. This shows that most branches have very low peaks and many flat areas.



**Fig. 4:** Histogram of the six fitness landscape measures applied on the 7029 branches. It is calculated by taking the average of 30 runs of RW for each branch.

According to the PIC metric, the highest value is 0.014, and the average is 0.004. This shows that branches generally consist of flat areas with very few slopes.

When the DBI metric is examined, the highest value is 0.089, and the average is 0.022. As for IC and PIC, the low values for the DBI metric do not suggest that the landscape is rugged (recall Table 1).

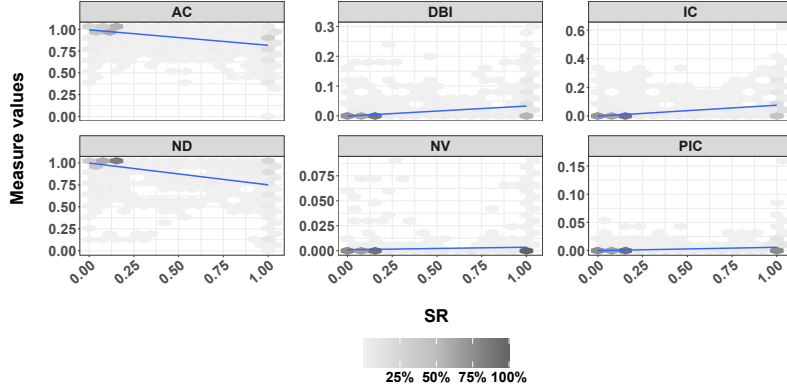
When analyzing Figures 3 and 4, we see that although the branches' values for AC and ND metrics fall within the range of  $[0, 1]$ , the majority have a value of 0. Similarly, for the NV metric, which ranges from  $[0, 0.09]$ , most branches again only show a value of 0. For the IC metric, values are within  $[0, 0.63]$ , but again, the majority of branches have a value of 0. The PIC metric ranges from  $[0, 0.16]$ , yet most branches still have a value of 0. Lastly, regarding the DBI metric, although values vary from  $[0, 0.3]$ , most branches again have a value of 0.

**RQ1:** *Neutrality prevails across the fitness landscape, where most branches appear relatively smooth. Only very few branches seem rugged.*

## 5.6 Results for the RQ2

Spearman correlation values between success rates (SR) and each metric were calculated. Spearman's correlation is a statistical measure that quantifies the monotonic relationship between two variables. A monotonic relationship indicates that as one variable increases, the other tends to increase (positive correlation) or decrease (negative correlation). According to these calculations (Table 6), the correlation coefficient of SR and AC is  $-0.58$ , ND is  $-0.58$ , NV is  $0.61$ , IC is  $0.60$ , PIC is  $0.59$ , and DBI is  $0.60$ . Figure 5 examines the effect of fitness landscape features.

When Figure 5 is examined, we can see that the SR increases as the AC metric decreases. In other words, it can be said that test generation becomes more difficult in cases where fitness values have a high correlation. When the ND metric is examined, the success rate decreases as the neutrality distance increases. When examining the NV metric, the diversity of fitness values makes it easier to reach a solution. The ability



**Fig. 5:** The Spearman correlation of Success Rate (SR) with each of the six measures for all of the 7029 branches. While the Y-axis presents the value of each metric, the X-axis shows the SR. Each hexagon represents a group of runs for different branches, with the density of the hexagons increasing as the number of runs within a hexagon grows.

**Table 6:** Spearman correlation coefficients ( $\rho$ ) between SR and each measure.

Measure	Spearman's $\rho$	p-value
AC	-0.58	< 0.001
DBI	0.60	< 0.001
IC	0.60	< 0.001
ND	-0.58	< 0.001
NV	0.61	< 0.001
PIC	0.59	< 0.001

to generate a solution increases as the NV value increases. Analyzing the IC reveals that greater information content enhances the SR. Examining the PIC shows that an expanded landscape modality also boosts the SR. When looking at the DBI metric, a higher variety of flat areas within the landscape leads to increased success. As a result, while there is a negative relationship between the AC and ND metrics and SR, there is a positive relationship between the NV, IC, PIC, and DBI metrics.

Taking into account the interpretations of these metrics from Table 1, and taking into account the obtained Spearman correlation values, an increase in neutrality (i.e., high ND and low NV) leads to worse results (i.e., lower SR). However, more ruggedness (i.e., AC closer to 0, and higher values for IC, PIC and DBI) seems to lead to better results.

**Table 7:** Six fitness landscape measurements for each group, gathered using the RW algorithm. *NB* represent the size of each group (i.e., number of branches in each group), considering a total of 7029 branches.

<i>GROUP</i>	<i>NB</i>	<i>AC</i>	<i>ND</i>	<i>NV</i>	<i>IC</i>	<i>PIC</i>	<i>DBI</i>
Easy	1429	0.763	0.651	0.004	0.099	0.007	0.043
Search	1047	0.908	0.923	0.003	0.029	0.003	0.014
Hard	4437	0.981	0.980	0.001	0.005	0.000	0.002
RW	116	0.787	0.639	0.002	0.090	0.007	0.037

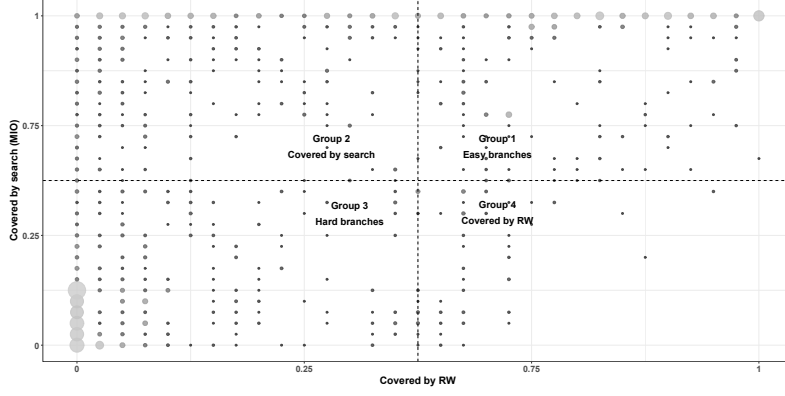
**RQ2:** *Increasing the correlation between solutions ( $AC$ ) or the number of neutral steps ( $ND$ ) makes system-level test generation more challenging. Conversely, increasing the number of distinctive regions ( $NV$ ), the number of peaks in the landscape ( $IC$ ), the modality ( $PIC$ ), or the diversity of neutral areas ( $DBI$ ) makes system-level test generation easier. In our analyses, while neutrality leads to more challenges in the search, an increase in ruggedness seems on the other hand to make it is easier.*

## 5.7 Results for the RQ3

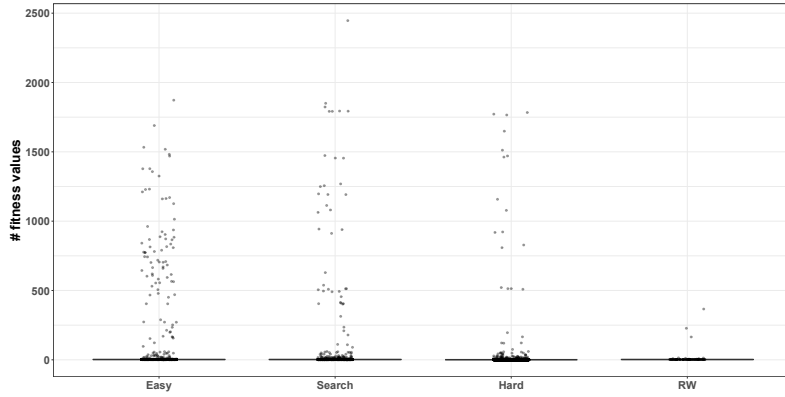
In the previous sections, we have observed how features of the fitness landscape can influence search outcomes. In this section, we will investigate which features of the code affect the search process. To detail this, we categorize each branch based on the success of the search algorithm (MIO) compared to the random walk (RW).

As discussed in Section 5.4, the “Easy” group contains branches where both the RW and MIO algorithms have success rates greater than 50%. The “Hard” group includes branches with success rates below 50% for both algorithms. The “Search” group is characterized by a success rate above 50% for the MIO algorithm and below 50% for the RW algorithm, while the “RW” group is the opposite. Table 7 and Figures 6, 7, and 8 are created according to these distinctions.

When the Table 7 is examined, we can see that in the Search group both  $AC$  (0.908) and  $ND$  (0.923) values are relatively high, indicating that the landscape is smoother and contains a greater density of neutral areas. The  $NV$  value (0.003) is higher than that of the RW and Hard groups but lower than that of the Easy group. According to the  $IC$  metric (0.029), the landscape’s ruggedness, measured by the number of peaks, is lower than that of the Easy and RW groups but higher than that of the Hard group. The  $PIC$  metric (0.003) indicates that the modality is greater than that of the Hard group, while the  $DBI$  metric (0.014) shows that the diversity of flat areas is more than in the Hard group but less than in the Easy and RW groups. When examining the Hard group, which contains most of the branches, it is observed that the landscape is smooth ( $AC = 0.981$ ), with a very high level of neutral areas ( $ND = 0.980$ ). However, the  $NV$  is relatively low at 0.001, suggesting that it typically has a single fitness value that cannot be improved. The number of peaks in this landscape ( $IC = 0.005$ ) and the modality ( $PIC = 0.000$ ) are both low. Additionally, the diversity of flat areas is quite limited ( $DBI = 0.002$ ). Finally, as anticipated, there are only few branches in the RW group. Generally, the metrics for this group are similar to those of the Easy group, with the  $NV$  metric (0.002) being the main difference. While the diversity of neutral



**Fig. 6:** Four groups of the branches based on their coverage by MIO and RW where a large bubble size indicates a high number of branches



**Fig. 7:** Number of discrete fitness values obtained by the random walk for each branch in the four groups

areas in RW group is lower than that in the Easy group, it remains higher than in the other groups. However, the diversity of fitness values is lower, which may have caused challenges for the search algorithms in covering it. As a result, these branches may have been more effectively addressed by the RW group, which generates more random solutions.

Upon examining Figure 6, it becomes clear that most results fall within the Hard class (bottom left). In this category, most branches remain uncovered by either algorithm in most of the runs. It is important to note that branches not covered by any algorithm across their 30 runs were excluded from the evaluation, i.e., the data for Hard branches still include at least one successful run out of 60 runs. Another significant group comprises the Easy branches covered by both algorithms (top right).

Figure 7 gives the number of distinct fitness values of branches in different groups. The number of different fitness values is generally close to 1 in all groups. It should

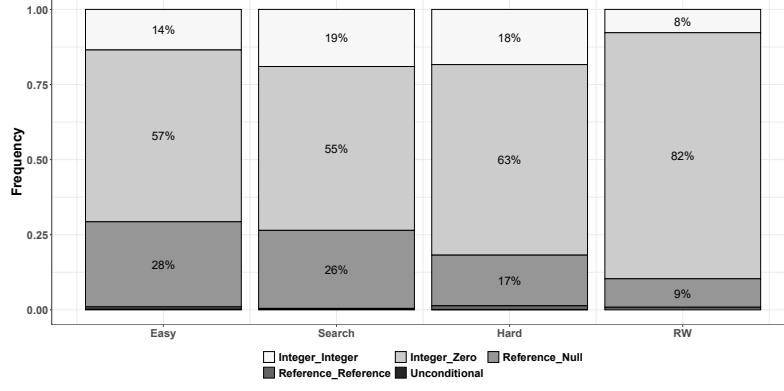


Fig. 8: Classifications of the branch types in the four groups

be noted here that a fitness value is in the range of  $[0, 1]$  and is based only on branch distance. This restricts the diversity of fitness values considerably. A slight slope in the landscape allows for different fitness values. There are a high number of different fitness values for only a limited number of branches. This shows that the landscape’s slope is quite low and generally consists of flat areas. It is seen that the highest number of different fitness values are in the Easy group and the lowest in the RW group. Unsurprisingly, the fitness diversity in the RW group is low because a search algorithm like MIO directs the search based on gradient information and tries to find the appropriate solution. Since RW is based on random mutations, it does not need gradient information or direction mechanisms like fitness function. Therefore, the diversity created by mutations can potentially overcome the limitations of gradient-based or search-based optimization methods.

All groups, except for RW, show similar numbers of discrete fitness function values. This suggests that the ruggedness of the groups does not differ significantly. Therefore, a more detailed analysis of the branches is necessary to evaluate their performance differences. Figure 8 provides this detailed examination, where each branch is categorized based on their bytecode type, as specified in more details in reference [Shamshiri et al \(2018\)](#).

One thing to consider is that EVOMASTER has reported an “Unconditional” branch type that includes the “goto” statement. This is usually related to `continue` and `break` statements in loops. As this is not technically considered as a branch, it can be considered as a fault in EVOMASTER, which will be fixed in a future release. Therefore, although for consistency these “branches” are still listed in the following tables, they are not discussed in any further detail. As only 10 cases fall into this category, they do not significantly impact any of our results.

Upon examining Figure 8, it becomes evident that most groups consist of branches of the “Integer\_Zero” type. This branch type primarily corresponds to the  $if(x)$  boolean expression translated by the Java compiler, contributing to the presence of plateaus in the fitness landscape ([McMinn \(2004\)](#)), i.e., a typical source of *flag problem* ([Harman et al \(2002\)](#)). In the Search group, the number of “Integer\_Integer” type branches is



slightly higher than in the other groups. Notably, this branch type is the only one that creates a clear gradient in the fitness landscape.

Conversely, in the Easy group, there is a higher prevalence of “Reference\_Null” type branches compared to the other groups. Although the “Reference\_Null” and “Reference\_Reference” type benchmarks exhibit lower discrete fitness values, they are anticipated to be complicated. Interestingly, most branches of the RW are of the “Integer\_Zero” type, which may indicate the presence of complex predicates that are challenging for an optimization algorithm to resolve due to a lack of fitness gradient.

**Table 8:** Comparison of the performance of the RW and MIO algorithms in different branch types. NB represents the number of branches, and SR indicates the success rate. The “Integer\_Integer” column illustrates branches where two integer values are compared, “Integer\_Zero” shows comparisons between an integer value and 0, “Reference\_Reference” signifies the comparison of two reference values, “Reference\_Null” compares a reference value to a null value, and the final category covers the “Unconditional” which is “goto” statement.

	Integer_Integer				Integer_Zero				Reference_Reference				Reference_Null				Unconditional			
SUTS	NB	SR <sub>RW</sub>	SR <sub>MIO</sub>	$\hat{A}_{12}$	NB	SR <sub>RW</sub>	SR <sub>MIO</sub>	$\hat{A}_{12}$	NB	SR <sub>RW</sub>	SR <sub>MIO</sub>	$\hat{A}_{12}$	NB	SR <sub>RW</sub>	SR <sub>MIO</sub>	$\hat{A}_{12}$	NB	SR <sub>RW</sub>	SR <sub>MIO</sub>	$\hat{A}_{12}$
<i>catwatch</i>	14	0.32	0.47	0.70	151	0.39	0.51	<b>0.68</b>	1	0.10	0.30	1.00	91	0.43	0.63	<b>0.72</b>	0	NA	NA	NA
<i>cwa-verification</i>	6	0.28	0.47	0.67	46	0.43	0.55	0.62	6	0.59	0.65	0.44	25	0.51	0.61	0.59	0	NA	NA	NA
<i>genome-nexus</i>	78	0.42	0.61	<b>0.66</b>	262	0.47	0.62	<b>0.64</b>	0	NA	NA	NA	210	0.56	0.77	<b>0.75</b>	0	NA	NA	NA
<i>gestaohospital-rest</i>	6	0.33	0.80	0.85	25	0.27	0.59	<b>0.77</b>	0	NA	NA	NA	10	0.37	0.72	<b>0.79</b>	0	NA	NA	NA
<i>graphql-ncs</i>	72	0.41	0.79	<b>0.83</b>	104	0.45	0.85	<b>0.87</b>	0	NA	NA	NA	0	NA	NA	NA	0	NA	NA	NA
<i>graphql-scs</i>	50	0.33	0.60	<b>0.70</b>	181	0.49	0.48	<b>0.43</b>	1	0.40	1.00	1.00	1	0.60	1.00	1.00	0	NA	NA	NA
<i>languagetool</i>	643	0.01	0.11	<b>0.88</b>	2278	0.03	0.14	<b>0.89</b>	55	0.03	0.15	<b>0.96</b>	641	0.08	0.22	<b>0.89</b>	4	0.00	0.15	<b>1.00</b>
<i>market</i>	0	NA	NA	NA	20	0.33	0.44	0.61	0	NA	NA	NA	7	0.39	0.53	0.63	0	NA	NA	NA
<i>ocvn-rest</i>	11	0.27	0.56	0.69	52	0.33	0.74	<b>0.79</b>	0	NA	NA	NA	61	0.48	0.91	<b>0.90</b>	0	NA	NA	NA
<i>patio-api</i>	0	NA	NA	NA	14	0.12	0.48	<b>0.83</b>	0	NA	NA	NA	0	NA	NA	NA	0	NA	NA	NA
<i>pay-publicapi</i>	2	0.50	0.50	0.50	9	0.57	0.67	0.54	0	NA	NA	NA	2	0.57	1.00	0.75	0	NA	NA	NA
<i>petclinic-graphql</i>	0	NA	NA	NA	4	0.39	1.00	<b>1.00</b>	0	NA	NA	NA	17	0.38	0.93	<b>0.94</b>	0	NA	NA	NA
<i>proxyprint</i>	14	0.07	0.14	<b>0.23</b>	105	0.07	0.09	<b>0.18</b>	1	0.00	0.03	1.00	120	0.11	0.53	<b>0.65</b>	0	NA	NA	NA
<i>reservations-api</i>	0	NA	NA	NA	8	0.65	0.70	0.62	0	NA	NA	NA	3	0.63	1.00	0.83	0	NA	NA	NA
<i>rest-ncs</i>	72	0.09	0.79	<b>0.95</b>	96	0.09	0.81	<b>0.92</b>	0	NA	NA	NA	0	NA	NA	NA	0	NA	NA	NA
<i>rest-news</i>	0	NA	NA	NA	29	0.40	0.82	<b>0.85</b>	0	NA	NA	NA	48	0.37	0.80	<b>0.85</b>	6	0.57	0.83	0.76
<i>rest-scs</i>	46	0.06	0.57	<b>0.78</b>	154	0.08	0.46	<b>0.66</b>	1	0.10	1.00	1.00	0	NA	NA	NA	0	NA	NA	NA
<i>restcountries</i>	30	0.07	0.94	<b>1.00</b>	155	0.07	0.89	<b>0.99</b>	0	NA	NA	NA	17	0.16	0.91	<b>1.00</b>	0	NA	NA	NA
<i>rpc-thrift-ncs</i>	72	0.79	0.84	0.52	96	0.78	0.86	0.54	0	NA	NA	NA	0	NA	NA	NA	0	NA	NA	NA
<i>rpc-thrift-scs</i>	46	0.70	0.70	0.52	164	0.75	0.67	0.45	1	1.00	1.00	NA	0	NA	NA	NA	0	NA	NA	NA
<i>scout-api</i>	39	0.11	0.45	<b>0.75</b>	117	0.12	0.54	<b>0.83</b>	1	0.00	0.13	1.00	81	0.18	0.68	<b>0.88</b>	0	NA	NA	NA
<i>session-service</i>	0	NA	NA	NA	8	0.32	0.40	0.52	1	0.97	1.00	1.00	4	0.29	0.50	0.50	0	NA	NA	NA
<i>timbuctoo</i>	15	0.54	0.66	0.63	219	0.54	0.78	<b>0.74</b>	3	0.78	0.99	1.00	97	0.70	0.96	<b>0.86</b>	0	NA	NA	NA

**Table 9:** Branch types of the reached but never covered branches, for each SUT.

SUTS	Integer_Integer	Integer_Zero	Reference_Reference	Reference_Null	Unconditional
<i>catwatch</i>	0	11	1	29	0
<i>cwa-verification</i>	0	4	0	9	0
<i>genome-nexus</i>	6	38	0	58	0
<i>gestaohospital-rest</i>	0	3	0	6	0
<i>graphql-scs</i>	0	9	1	1	0
<i>languagetool</i>	185	818	41	412	6
<i>market</i>	0	0	0	3	0
<i>ocvn-rest</i>	1	11	0	19	0
<i>patio-api</i>	0	4	0	0	0
<i>pay-publicapi</i>	0	1	0	0	0
<i>petclinic-graphql</i>	0	0	0	1	0
<i>proxyprint</i>	0	27	3	70	0
<i>reservations-api</i>	0	2	0	3	0
<i>rest-news</i>	0	1	0	20	0
<i>rest-scs</i>	0	14	1	0	0
<i>restcountries</i>	0	7	0	3	0
<i>rpc-thrift-scs</i>	2	4	1	0	0
<i>scout-api</i>	0	11	1	29	0
<i>session-service</i>	0	0	1	0	0
<i>timbuctoo</i>	1	67	3	59	0
Total	195	1032	53	722	6

Table 8 compares the performance of the RW and MIO algorithms in different branch types. In the table, NB represents number of branches, while SR means the success rate. The average success rates of the branches are presented in the table. Statistical analyses were performed using the average success rate value of each branch. In cases where the Mann-Whitney-Wilcoxon test result is  $p < 0.05$ , the  $\hat{A}_{12}$  value is shown in bold.

When “Integer\_Integer” type branches are examined, it is observed that the success rate of the MIO algorithm is significantly better. The most interesting of these results is *proxyprint* in terms of  $\hat{A}_{12}$ . There are 14 branches in total, and in 2 of these branches, the MIO algorithm has a 100% success rate, while in the remaining 12 branches, it has a 0% success rate. RW could not achieve a 100% success rate in any branch, but its success rate is higher than 0 in 12 branches (usually, it is successful in 1 out of 30 runs). Therefore, MIO is seen to be more successful on average, while RW shows better performance according to  $\hat{A}_{12}$  values. When “Integer\_Zero” branches, which generally cause boolean flags, are examined, it is noted that most branches are “Integer\_Zero” type branches. This may indicate that the landscape has boolean flags in general. Except for *graphql-scs*, *proxyprint*, and *rpc-thrift-scs*, it is observed that the MIO algorithm has a higher success rate. When we examine “Reference\_Reference” type branches, it is noted that very few branches are of this reference type. Statistically, it is the branch type where the MIO and RW algorithms can come closest to each other. When we examine “Reference\_Null” type branches, it is seen that MIO is generally more successful.

Finally, Table 9 provides information about the branches that are reached but never covered, grouped according to their branch type. According to this table, the most uncovered branch belongs to the *languagetool* SUT. Here, the most frequently encountered branch type is the “Integer\_Zero” type. Algorithms struggle to find solutions due to plateaus created by boolean flags and the complexity of predicates. Second in line are the “Reference\_Null” type branches. These branches are usually caused by null checks that prevent application crashes caused by null pointer exceptions

```

1 CharSequence var7 = (CharSequence)file;
2 Regex var8 = new Regex(".");
3 byte var9 = 0;
4 Collection $this$toTypedArray$iv = (Collection)var8.split(var7, var9);
5 int $i$f$toTypedArray = false;
6 Object[] var10000 = $this$toTypedArray$iv.toArray(new String[0]);
7 if (var10000 == null) {
8     throw new NullPointerException("null cannot be cast to non-null type kotlin.
9     Array<T of kotlin.collections.ArraysKt__ArraysJVMKt.toTypedArray>");
10 } else {
11     fileparts = (String[])var10000;
12     ...
13 }

```

Fig. 9: Code snipped from FileSuffix class in graphql-scs problem.

```

1 aload 9
2 iconst_0
3 anewarray java/lang/String
4 invokeinterface java/util/Collection.toArray([Ljava/lang/Object;)[Ljava/lang/
5   Object;
6 dup
7 ifnonnull L12
8 pop
9 new java/lang/NullPointerException
10 dup
11 ldc "null cannot be cast to non-null type kotlin.Array<T of kotlin.collections.
12   ArraysKt__ArraysJVMKt.toTypedArray>" (java.lang.String)
13 invokespecial java/lang/NullPointerException.<init>([Ljava/lang/String;)V
14 athrow

```

Fig. 10: Bytecodes of the FileSuffix class given in Figure 9

or null check transformations in null safety languages such as Kotlin. These scenarios occur when a `null` value is not anticipated within the program's normal flow.

For instance, in the *graphql-scs* problem, the code expression "`fileparts = file.split(".").toRegex()).toTypedArray()`" has been translated into the statements depicted in Figure 9, which has subsequently been converted to bytecode shown in Figure 10. Covering the branch in this expression proves to be quite challenging. Furthermore, branches that could be covered in unit testing (e.g., by passing a `null` as input parameter) might be *infeasible* in system testing, as any input to the function has to go through the user input interfaces (e.g., HTTP calls in the case of REST APIs), which might do some input sanitization and filtering.

**RQ3:** *Most plateaus in the fitness landscape result from boolean flags and null checks used to prevent crashes in the code. This creates branches that are difficult to cover, or simply infeasible in system testing.*

## 5.8 Results for the RQ4

Table 10 presents the results of six different fitness landscape metrics published in the baseline study [Albunian et al \(2020\)](#). Recall that the values reported by the various groups in this study are outlined in Table 7. Examining both tables reveals

**Table 10:** Comparison of six fitness landscape measurements for each group in the baseline (unit) and system-level test generation studies [Albunian et al \(2020\)](#).

	<i>GROUP</i>	<i>AC</i>	<i>ND</i>	<i>NV</i>	<i>IC</i>	<i>PIC</i>	<i>DBI</i>
<b>Unit Test</b>	Easy	0.652	0.114	0.009	0.401	0.092	0.872
	Search	0.829	0.129	0.005	0.125	0.058	0.904
	Hard	0.898	0.516	0.002	0.076	0.028	0.960
	RW	0.852	0.258	0.004	0.098	0.040	0.928
<b>System Test</b>	Easy	0.763	0.651	0.004	0.099	0.007	0.043
	Search	0.908	0.923	0.003	0.029	0.003	0.014
	Hard	0.981	0.980	0.001	0.005	0.000	0.002
	RW	0.787	0.639	0.002	0.090	0.007	0.037

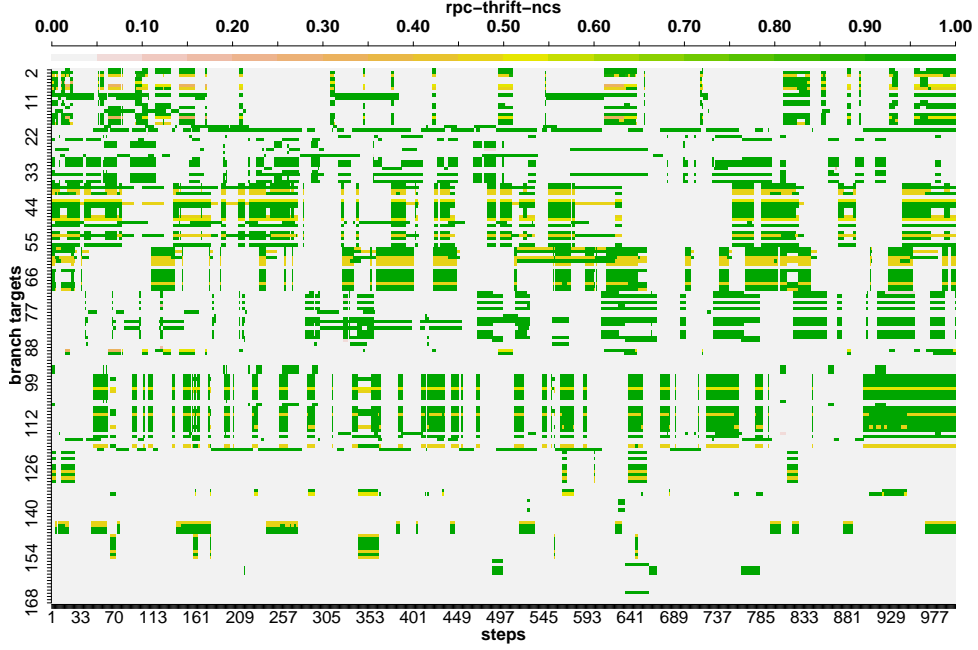
a parallelism in AC values, except for the RW algorithm. While the RW algorithm falls in the middle of the Search and Hard groups in the baseline study, it aligns more closely with the Easy groups in our study. This discrepancy can be attributed to the inherent differences between unit and system-level testing. Functions can be called directly in unit testing, once a valid instance is created. In contrast, system-level testing requires the use of a proxy method, i.e., data is sent from the user interfaces. Web APIs differ significantly from object-oriented software in terms of testing. In object-oriented programming, operations on an object share the same state by default. For example, calls like “`x.foo()`; `x.bar()`;” both operate on the same object “`x`”. However, Web APIs may involve different states when randomizing actions (e.g., the endpoint `/x/{id}` requires a specific ID to reference the same state). Consequently, mutation actions occurring in system-level tests are less directed, possibly leading to surfaces with more plateaus.

A similar pattern is observed with the ND parameter. In system-level tests, the amount of neutral distance is significantly higher. In fact, most of the search occurs in neutral areas, making it challenging to obtain a fit solution. When examining the neutral volumes of the solutions, we find that results are similar for both types of tests. However, unit test exhibits a greater neutral volume, allowing the fitness function to guide the search more effectively.

When we analyze the IC metric, we see more information content in unit tests across all groups. This indicates that the peak values are higher in unit tests. Conversely, the peak number in system-level tests is considerably lower, particularly within the Easy group, where the difference is approximately four times lower, and the information level is significantly lower. Notably, the information level of the branches where RW is better is almost comparable to that of the other methods.

Regarding the PIC metric, system-level tests show a lower modality level than unit tests.

The most significant difference between the two test types is evident in the DBI metric, which illustrates the distribution of continuous actions. The DBI metric measures the behaviors that a fitness value performs continuously (Recall Eq. 8). For instance, continuous increases (a sequence of “11”), continuous decreases (a sequence of “11”), or a constant value (a sequence of “00”) during the search are calculated based on fitness values. Greater diversity in these sequences results in values approaching 1,



**Fig. 11:** Heatmap representing the fitness values of 168 branches (y-lab is *branches*) tracked over 1000 steps (x-lab is *steps*) during fuzzing with RW on *rpc-thrift-ncs*. Note that each row illustrates the fitness values of a single branch at each of 1000 steps.

while a lack of diversity (e.g., no change) yields values closer to 0. In system-level tests, the diversity of these continuous values is relatively low, with most steps consisting solely of “00” sequences, leading to a DBI value near 0. In contrast, unit-level tests demonstrate a high level of diversity in these continuous sequences. Compared to unit testing, fuzzing Web APIs has a much larger search space, and the test suite consists of a greater number of test cases. Each test typically targets a small segment of code related to the API. Thus, during fuzzing, each test generated at each step optimizes different branches, while many other branches have a fitness value of 0. Over the course of the search, a branch may only be affected in a few optimization steps, and since unaffected branches have a fitness value of 0, this results in the DBI for each single branch being close to 0 in our context.

To investigate the results of the DBI metric, we analyze the fitness values of 168 branches of *rpc-thrift-ncs* (achieved the highest DBI, i.e., 0.089) over 1000 steps using a heatmap (see Figure 11). Based on the distribution of fitness values shown in Figure 11, each test generated at a step impacts only a subset of the branches, ranging from 12 ( $12/168 = 7.1\%$ ) to 90 ( $90/168 = 53.6\%$ ) targets on *rpc-thrift-ncs*. This is because the branches are associated with distinct endpoints of Web APIs and can only be executed if the corresponding endpoint is requested in the test. As a result, the fitness value for branches unaffected by the test at each step is 0.

Differences between unit and system test generation are expected. However, such extreme and sharp difference in DBI values was unexpected, and quite surprising. The three authors have reviewed all of our code, possibly to see if there was any clear error in our analyses. For the sake of completeness, we also contacted the original authors of [Albunian et al \(2020\)](#), asking if the raw data of experiments (missing in the replication package) was still accessible somewhere, or if the data was transformed with some pre-processing technique not discussed in the article and not present in the replication package. This could have helped in shedding light regarding if there was any error in ours or in their study. One of such authors replied, unfortunately stating he had no longer access to that data (which is not unreasonable, as few years have passed at the time of writing, and such raw data can take a lot of space). Still, considering Table 1, low values for DBI are consistent with the low values we obtained for IC and PIC and the high values for AC, as they all four indicate a lack of ruggedness. This is what motivated us to wonder whether there was any issue in the handling and interpretation of DBI in [Albunian et al \(2020\)](#).

**RQ4:** *System-level fitness landscape characterization includes more plateaus and lower information levels than in unit testing.*

## 6 Discussion

First, in this study we compared the RW and MIO algorithms to understand better the differences between both these algorithms. As expected, guiding the search with a fitness function yielded positive results in most cases, with the MIO algorithm outperforming the RW algorithm in the majority of the SUTs. Notably, the MIO algorithm achieved these results while sending significantly fewer HTTP requests than the RW algorithm. This suggests that using a fitness function enhances coverage and leads to a more efficient use of resources.

Next, we examined the general characteristics of the fitness landscape. Our results were based on a total of 9037 branches, of which 7029 were analyzed. We excluded 2008 branches that were not covered by either the RW or MIO algorithms during any of their 30 runs. Most of these branches contain neutral areas of the fitness landscape, with only a tiny portion classified as rugged.

Then, the fitness landscape features of the branches were examined, and their contribution to the success rate was investigated. Accordingly, high AC and ND values make generating tests at the system-level difficult. The reason for this is that the fitness values have a high level of correlation with each other, which makes it challenging to direct the search. In addition, the increase in the ND metric, which is the number of steps between two different fitness values during the search, makes the search difficult, as expected. High NV, IC, PIC, and DBI metrics make it easier to generate system-level tests. As the NV metric increases, the difference in the number of fitness values in the landscape increases. Searching in a landscape with different fitness values will be much easier. The increase in the IC metric means that the landscape has higher information. While this can sometimes make the search difficult, the increase in information in system-level test generation, where the landscape generally consists of neutral areas, has been quite important for directing the search. PIC provides information about

the modality of the landscape. The effect of single-modal or multimodal problems on the difficulty may vary depending on the type of problem. When the effect on system-level test case generation is examined, it can be said that it is easier to cover multimodal branches. The search becomes more manageable with the increase of the DBI metric. DBI helps us understand the diversity of flat areas. It is seen that flat areas negatively affect the search. At least the diversification of these flat areas affects the search positively. When all these metrics are examined, it is seen that any piece of information that can guide the search in the landscape positively affects the search.

The next phase of the research focuses on investigating how source code features influence search processes. Various types of branches were examined, leading to specific findings. It is observed that fitness diversity is significantly lower in groups where the RW algorithm outperforms the MIO algorithm. This suggests the presence of branches that cannot be influenced by fitness evaluation and can only be addressed through random mutations. One key finding is that most of these branches are of the “Integer.Zero” type. This explains the high number of neutral areas present in the search landscape. Branches in the “Integer.Zero” category typically arise from the Java compiler’s conversion of boolean predicates, and these boolean predicates often lead to plateaus in the fitness landscape (McMinn (2004)). Additionally, when examining the branches that are never covered, the most common branches are “Reference.Null” branches that follow the branches in the “Integer.Zero” group. These branches cause boolean flags and plateaus in the fitness landscape. This situation often arises from `null` checks implemented to prevent program crashes or due to the requirements of null-safe programming languages, making it challenging to cover branches associated with these `null` values.

Finally, the differences between unit and system-level test generation are compared using results obtained from the replicated study. As in system-level test generation, while the increase in AC and ND values negatively affects the search, the increase in NV, IC, PIC values positively affects the search. While the values are generally similar, there are some differences. A possible explanation for these differences is that specific methods can be called directly when generating a unit test. This allows the fitness function to be used more effectively. In system-level test generation, it is realized through proxy methods. This makes it challenging to perform the search. This may be the main reason for the differences between metric values.

The most obvious of these differences is the DBI metric. While it is very close to the value of 1 in unit test generation, it is close to 0 in system-level test generation. This is related to the variety of flat areas obtained during the search. As in unit testing, the system-level fitness landscape generally consists of flat areas. However, the variety of these flat areas is quite limited in system-level testing. They mainly consist of “00” sequences, meaning no change. As a result, while the fitness landscape has more plateaus in system-level test generation compared to unit test generation, the level of information is also less. Alternative, a possible explanation for this huge differences between DBI metric values could be due to possible errors in our analyses or in the analyses carried out in Albunian et al (2020).

Although system-level testing is conducted through external interfaces such as APIs, EvoMaster is integrated with white-box heuristics that guide and evaluate the



test generation process using internal control-flow information, e.g., how heuristically close the current execution is to covering branches. This allows us to assess how effectively the generated system-level tests explore the internal logic of the system. While branch-level coverage is traditionally associated with unit testing, it remains a meaningful criterion at the system level in white-box testing, where the goal is to exercise as many distinct code paths as possible, but through system interfaces rather than individual methods as in unit testing. White-box system-level testing via system interfaces enables longer execution paths that facilitate the testing of complex decision points, control flows, and interactions among components. After all, a fault cannot be detected if the code in which it lies is never executed by any test case. Where in system testing tens or hundreds of thousands of lines can be executed with a single test case, a manual tester might not aim at maximizing branch coverage in their manual testing, as there would be too many thousands to consider. However, automated techniques like SBST can scale to such large and complex contexts. Furthermore, system-level testing also allows for the use of additional oracles beyond internal execution, such as HTTP status codes, error messages, and security-related behaviors. These oracles complement structural coverage metrics and provide a richer view of test effectiveness across different layers of the system.

In future studies, it will be possible to define a fitness function appropriate for a solution’s landscape’s ruggedness or neutrality characteristics that can help guide the search more effectively. Additionally, a new solution generation mechanism can be developed. For instance, in a landscape characterized by many plateaus, the effectiveness of the crossover operator is limited, making it generally more meaningful to generate new individuals using the mutation operator. Besides, solution mechanisms like crossover, which allows two individuals to exchange information, are expected to yield better results in information-rugged landscapes. Currently, MIO and EVOMASTER do not use any kind of crossover operator, due to the complexity of defining a meaningful one for this problem domain. The results of this study points to the possible need to develop novel mutation and crossover operators, customized for this problem domain, taking into account these fitness landscape characteristics. Also, in retrospective this can explain existing results. For example, the introduction of hypermutation in EVOMASTER (Zhang and Arcuri (2021a)) led to significantly better results, which can be explained due the neutrality characteristics of this problem’s domain fitness landscape.

## 7 Threats To Validity

To address issues with internal validity, we carefully tested our EVOMASTER implementation with thousands of unit tests and end-to-end tests. Also, EVOMASTER is released as open source, and anyone who wants to can check its source code.

To address the randomness of the applied algorithms, each experiment was repeated 30 times and appropriate statistical tests were performed following common literature guidelines (Arcuri and Briand (2014)). In particular, we follow the guidelines in Arcuri and Briand (2014) that explicitly recommend against using p-value correction

mechanisms, like for example the Bonferroni one, as they are considered harmful (Perneger (1998); Nakagawa (2004)).

The external validity threats concern whether our results can be generalized to other SUTs and SBST problems. We conducted our study on 23 different SUTs prepared for different kinds of APIs, including REST, GraphQL, and Thrift. This provides a large variety of different kinds of SUTs, which strengthen our results. However, system-level testing is computationally costly, and a limited number of SUTs can be used to complete the experiments within reasonable time considering an academic context. In addition, although especially REST APIs are frequently used in industry, it can be challenging to find and set up open-source SUTs suitable for experiments. Furthermore, as with the original study that used EvoSuite, we replicate our work based on white-box test generation tools that target Java/Kotlin systems, specifically EVOMASTER. Consequently, our findings may be influenced by artifacts unique to the JVM ecosystem and white-box testing strategies. While the general insights into fitness landscape characteristics are likely applicable across different contexts, caution is necessary when extrapolating these results to black-box testing tools or systems developed in other technological environments. Further research is required to evaluate the extent to which our observations generalize beyond the current setting.

## 8 Conclusions

An in-depth evaluation of the performance of search algorithms used in system-level test generation requires a thorough understanding of the fitness function landscapes. This study examined fitness landscapes regarding ruggedness and neutrality, comparing them to the landscape features found in unit test generation. This study on system-level testing with EVOMASTER is a replication of an existing study on unit-level testing with EvoSuite (Albunian et al (2020)), using 23 Web APIs for the experiments.

Our results indicate that fitness landscapes are dominantly characterized by neutral regions, such as plateaus, which complicate the search process. The presence of information content within the landscape can help with the search. One primary reason for this neutral landscape is the presence of boolean flags.

Our results confirm the existing results reported in the literature for unit testing, albeit with some differences. Based on these findings, developing new fitness functions to extract more detailed information from the SUTs, implementing innovative testability transformations and designing novel mutation and crossover operators will be important priorities to enhance the search process.

## Data Availability Statement

EVOMASTER is open-source on GitHub,<sup>2</sup> with each new release, such as version 3.4.0, automatically published on Zenodo (Arcuri et al (2025c)). Likewise, the corpus EMB used for our experiments is open-source,<sup>3</sup> as well as being stored on Zenodo (Arcuri et al (2025b)).

---

<sup>2</sup><https://github.com/WebFuzzing/EvoMaster>

<sup>3</sup><https://github.com/WebFuzzing/EMB>

To enable the replicability of this study, and simplify comparisons, all the scripts used to setup our experiments and analyze the results are present in a replication package on Zenodo ([Sahin et al \(2025\)](#)). This includes as well all the raw data resulting from the experiments.

## Author Contribution Statement

The study was led by Omur Sahin, which did the majority of the work. Andrea Arcuri did the original planning and design of the work. All authors contributed with the implementation and extension of EVOMASTER needed to carry out the experiments. The first draft of the article was written by Omur Sahin, where the co-authors iterated over the draft to improve it and add any missing parts. Experiments were run and first analyzed by Omur Sahin. All authors contributed to the critical analysis of the results. Man Zhang further reviewed and analyzed discrepancies in the data (e.g., regarding the DBI metric).

## Acknowledgments

Omur Sahin is supported by the TÜBİTAK 2219 International Postdoctoral Research Fellowship Program (Project ID: 1059B192300060). Man Zhang is supported by State Key Laboratory of Complex & Critical Software Environment (SKLCCSE, grant No. CCSE-2024ZX-01) and the Fundamental Research Funds for the Central Universities. Andrea Arcuri is funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

## References

- Albunian N, Fraser G, Sudholt D (2020) Causes and effects of fitness landscapes in unit test generation. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp 1204--1212
- Aleti A, Moser I (2015) Fitness landscape characterisation for constrained software architecture optimisation problems. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, pp 11--20
- Aleti A, Moser I, Grunske L (2017) Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering* 24:603--621
- Ali S, Briand L, Hemmati H, et al (2010) A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36(6):742--762
- Alshraideh M, Bottaci L (2006) Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability (STVR)* 16(3):175--203

- Arcuri A (2017) Many Independent Objective (MIO) Algorithm for Test Suite Generation. In: International Symposium on Search Based Software Engineering (SSBSE), pp 3--17
- Arcuri A (2018) Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104:195--206
- Arcuri A (2019) RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28(1):3
- Arcuri A (2020) Automated black-and white-box testing of restful apis with evomaster. *IEEE Software* 38(3):72--78
- Arcuri A, Briand L (2014) A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24(3):219--250
- Arcuri A, Fraser G (2013) Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)* 18(3):594--623
- Arcuri A, Galeotti JP (2020a) Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29(4):1--31
- Arcuri A, Galeotti JP (2020b) Testability transformations for existing apis. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, pp 153--163
- Arcuri A, Galeotti JP (2021a) Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31(1):1--34
- Arcuri A, Galeotti JP (2021b) Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31(1):1--34
- Arcuri A, Iqbal MZ, Briand L (2012) Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)* 38(2):258--277
- Arcuri A, Galeotti JP, Marculescu B, et al (2021) EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6(57):2153
- Arcuri A, Zhang M, Galeotti JP (2023a) Advanced white-box heuristics for search-based fuzzing of rest apis. *arXiv preprint arXiv:230908360*
- Arcuri A, Zhang M, Golmohammadi A, et al (2023b) EMB: A curated corpus of web/enterprise applications and library support for software testing research. In:

- 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, pp 433–442
- Arcuri A, Poth A, Rjfolli O (2025a) Introducing black-box fuzz testing for rest apis in industry: Challenges and solutions. In: IEEE International Conference on Software Testing, Verification and Validation (ICST)
- Arcuri A, Zhang M, Golmohammadi A, et al (2025b) Webfuzzing/emb: v3.4.0. <https://doi.org/10.5281/zenodo.14597431>, URL <https://doi.org/10.5281/zenodo.14597431>
- Arcuri A, Zhang M, Seran S, et al (2025c) Webfuzzing/evomaster: v3.4.0. <https://doi.org/10.5281/zenodo.14597412>, URL <https://doi.org/10.5281/zenodo.14597412>
- Arcuri A, Zhang M, Seran S, et al (2025d) Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis. *Automated Software Engineering* 32(1):1–11
- Atlidakis V, Godefroid P, Polishchuk M (2019) Restler: Stateful REST API fuzzing. In: ACM/IEEE International Conference on Software Engineering (ICSE), p 748–758
- Barnett L, et al (1998) Ruggedness and neutrality-the nkp family of fitness landscapes. In: *Artificial Life VI: Proceedings of the sixth international conference on Artificial life*, pp 18–27
- Belhadi A, Zhang M, Arcuri A (2023) Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Transactions on the Web*
- Bertolino A (2007) Software testing research: Achievements, challenges, dreams. In: *Future of Software Engineering, 2007. FOSE’07*, IEEE, pp 85–103
- Bozkurt M, Harman M, Hassoun Y (2013) Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability (STVR)* 23(4):261–313
- Campos J, Ge Y, Albunian N, et al (2018) An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology (IST)* 104:207–235
- Canfora G, Di Penta M (2009) Service-oriented architectures testing: A survey. In: *Software Engineering*. Springer, p 78–105
- Chicano F, Ferrer J, Alba E (2011) Elementary landscape decomposition of the test suite minimization problem. In: *Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings 3*, Springer, pp 48–63
- Corradini D, Montolli Z, Pasqua M, et al (2024) Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning. In: *Proceedings of*

- the 39th IEEE/ACM International Conference on Automated Software Engineering, pp 1383--1394
- Curbera F, Duftler M, Khalaf R, et al (2002) Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet computing* 6(2):86--93
- Droste S, Jansen T, Wegener I (2002) On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science* 276:51--81
- Duran JW, Ntafos SC (1984) An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)* 10(4):438--444
- Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine
- Fraser G, Arcuri A (2011) 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*, pp 416--419
- Fraser G, Arcuri A (2013a) Evosuite at the SBST 2013 tool competition. In: *International Workshop on Search-Based Software Testing (SBST)*, pp 406--409
- Fraser G, Arcuri A (2013b) Whole test suite generation. *IEEE Transactions on Software Engineering* 39(2):276--291
- Gallagher MJ, Narasimhan VL (1997) Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering (TSE)* 23(8):473--484
- Golmohammadi A, Zhang M, Arcuri A (2023a) .NET/C# instrumentation for search-based software testing. *Software Quality Journal* pp 1--27
- Golmohammadi A, Zhang M, Arcuri A (2023b) On the Impact of Tool Evolution and Case Study Size on SBSE Experiments: A Replicated Study with EvoMaster. In: *International Symposium on Search Based Software Engineering*, Springer, pp 108--122
- Golmohammadi A, Zhang M, Arcuri A (2023c) Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* <https://doi.org/10.1145/3617175>, URL <https://doi.org/10.1145/3617175>
- Harman M, Jones BF (2001) Search-based software engineering. *Journal of Information & Software Technology* 43(14):833--839
- Harman M, Hu L, Hierons R, et al (2002) Improving evolutionary testing by flag removal. In: *Genetic and Evolutionary Computation Conference (GECCO)*, pp 1351--1358

- Harman M, Hu L, Hierons R, et al (2004) Testability transformation. *IEEE Transactions on Software Engineering* 30(1):3--16
- Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45(1):11
- Hatfield-Dodds Z, Dygalo D (2022) Deriving semantics-aware fuzzers from web api schemas. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, pp 345--346
- Holland JH (1992) Genetic algorithms. *Scientific american* 267(1):66--73
- Kim M, Sinha S, Orso A (2023) Adaptive rest api testing with reinforcement learning. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 446--458
- Korel B (1990) Automated software test data generation. *IEEE Transactions on software engineering* 16(8):870--879
- Kotelyanskii A, Kapfhammer GM (2014) Parameter tuning for search-based test-data generation revisited: Support for previous results. In: 2014 14th International Conference on Quality Software, IEEE, pp 79--84
- Laranjeiro N, Agnello J, Bernardino J (2021) A black box tool for robustness testing of rest services. *IEEE Access* 9:24738--24754
- Lefticaru R, Ipate F (2008) A comparative landscape analysis of fitness functions for search-based testing. In: 2008 10th international symposium on symbolic and numeric algorithms for scientific computing, IEEE, pp 201--208
- Liu Y, Li Y, Deng G, et al (2022) Morest: Model-based restful api testing with execution feedback. In: ACM/IEEE International Conference on Software Engineering (ICSE)
- Lu G, Bahsoon R, Yao X (2010) Applying Elementary Landscape Analysis to Search-Based Software Engineering. In: International Symposium on Search Based Software Engineering (SSBSE), pp 3--8
- Lukasczyk S, Fraser G (2022) Pynguin: automated unit test generation for Python. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp 168--172
- Malan KM, Engelbrecht AP (2013) A survey of techniques for characterising fitness landscapes and some possible ways forward. *Information Sciences* 241:148--163
- Mao K, Harman M, Jia Y (2016) Sapienz: Multi-objective automated testing for android applications. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), ACM, pp 94--105

- Martin-Lopez A, Segura S, Ruiz-Cortés A (2021) RESTest: Automated Black-Box Testing of RESTful Web APIs. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA). ACM, pp 682--685
- McMinn P (2004) Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14(2):105--156
- Nakagawa S (2004) A farewell to Bonferroni: the problems of low statistical power and publication bias. *Behavioral Ecology* 15(6):1044--1045
- Newman S (2021) Building microservices. " O'Reilly Media, Inc."
- Panichella A, Kifetew F, Tonella P (2018) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering (TSE)* 44(2):122--158
- Perneger T (1998) What's wrong with Bonferroni adjustments. *British Medical Journal* 316:1236--1238
- Pitzer E, Affenzeller M (2012) A comprehensive survey on fitness landscape analysis. *Recent advances in intelligent engineering systems* pp 161--191
- Poth A, Rrjolli O, Arcuri A (2025) Technology adoption performance evaluation applied to testing industrial rest apis. *Automated Software Engineering* 32(1):5
- Quiña-Mera A, Fernandez P, García JM, et al (2023) GraphQL: A systematic mapping study. *ACM Computing Surveys* 55(10):1--35
- Rajesh R (2016) Spring Microservices. Packt Publishing Ltd
- Sahin O, Zhang M, Arcuri A (2025) Replication package for causes and effects of fitness landscapes in system test generation: A replication study. <https://doi.org/10.5281/zenodo.14764981>, URL <https://doi.org/10.5281/zenodo.14764981>
- Sayyad AS, Goseva-Popstojanova K, Menzies T, et al (2013) On Parameter Tuning in Search Based Software Engineering: A Replicated Empirical Study. In: 2013 3rd International Workshop on Replication in Empirical Software Engineering Research, IEEE, pp 84--90
- Shamshiri S, Rojas JM, Gazzola L, et al (2018) Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability* 28(4):e1660
- Tawosi V, Sarro F, Petrozziello A, et al (2021) Multi-objective software effort estimation: A replication study. *IEEE Transactions on Software Engineering* 48(8):3185--3205
- Vassilev VK, Fogarty TC, Miller JF (2000) Information characteristics and the structure of landscapes. *Evolutionary computation* 8(1):31--60



- Viglianisi E, Dallago M, Ceccato M (2020) Resttestgen: Automated black-box testing of restful apis. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE
- Vogel T, Tran C, Grunske L (2019) Does diversity improve the test suite generation for mobile applications? In: International Symposium on Search Based Software Engineering, Springer, pp 58--74
- Vogel T, Tran C, Grunske L (2021) A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Information and Software Technology* 130:106436
- Waeselynck H, Fosse PT, Kaddour OA (2006) Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering* 12(1):35--63
- Wu H, Xu L, Niu X, et al (2022) Combinatorial testing of restful apis. In: ACM/IEEE International Conference on Software Engineering (ICSE)
- Zhang M, Arcuri A (2021a) Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31(1)
- Zhang M, Arcuri A (2021b) Enhancing Resource-Based Test Case Generation for RESTful APIs with SQL Handling. In: International Symposium on Search Based Software Engineering, Springer, pp 103--117
- Zhang M, Arcuri A (2023) Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* <https://doi.org/10.1145/3597205>, URL <https://doi.org/10.1145/3597205>
- Zhang M, Marculescu B, Arcuri A (2021) Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26(4):1--61
- Zhang M, Arcuri A, Li Y, et al (2023a) White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology* 32(5):1--38
- Zhang M, Belhadi A, Arcuri A (2023b) Javascript sbst heuristics to enable effective fuzzing of nodejs web apis. *ACM Transactions on Software Engineering and Methodology*
- Zhang M, Arcuri A, Teng P, et al (2024) Seeding and mocking in white-box fuzzing enterprise rpc apis: An industrial case study. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp 2024--2034

Zou F, Chen D, Liu H, et al (2022) A survey of fitness landscape analysis for optimization. *Neurocomputing* 503:129--139. <https://doi.org/10.1016/J.NEUCOM.2022.06.084>