Enhancing Search-Based Testing With Testability Transformations For Existing APIs

ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Oslo, Norway
JUAN P. GALEOTTI, Depto. de Computación, FCEyN-UBA, and ICC, CONICET-UBA. Argentina

Search-based software testing (SBST) has been shown to be an effective technique to generate test cases automatically. Its effectiveness strongly depends on the guidance of the fitness function. Unfortunately, a common issue in SBST is the so-called *flag problem*, where the fitness landscape presents a plateau that provides no guidance to the search. In this paper, we provide a series of novel *testability transformations* aimed at providing guidance in the context of commonly used API calls (e.g., strings that need to be converted into valid date/time objects). We also provide specific transformations aimed at helping the testing of REST Web Services. We implemented our novel techniques as an extension to EvoMaster, a SBST tool that generates system level test cases. Experiments on nine open-source REST web services, as well as an industrial web service, show that our novel techniques improve performance significantly.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Search-based software engineering**.

Additional Key Words and Phrases: SBST, test generation, testability transformation, system testing, REST

ACM Reference Format:
Andrea Arcuri and Juan P. Galeotti. 2021. Enhancing Search-Based Testing With Testability Transformations For Existing APIs. 1, 1 (May 2021), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Search-based software testing (SBST) [4, 37] has been shown to be an effective technique to automatically generate test cases. Examples include unit testing of Java software with open-source tools like EvoSuite [28], and testing of mobile applications with the Sapienz tool at Facebook [5].

When doing white-box testing, different techniques are used to define heuristics to smooth the search landscape. The most common in the literature of SBST is the so-called *Branch Distance* [41], which is used in tools like EvoSuite and EvoMaster [8, 17] (the latter targeting system test generation for web services). Given a boolean predicate in the code of the system under test (SUT), the branch distance provides a heuristic value to guide the search toward solving such constraints.

Unfortunately, a common issue is the so-called *flag problem* [21], where the branch distance is not able to provide any gradient. An approach to address this issue is to transform the code of the SUT to improve the fitness function, using so-called *Testability Transformations* [36]. For example, consider a scenario where our target program is dealing with string operations returning booleans [6], like comparisons of two strings for equality. By default, a predicate like `x.equals("foo")` would just be a flag, returning true or false. But testability transformations can be used to replace such functions

Authors' addresses: Andrea ArcuriKristiania University College and Oslo Metropolitan University, Oslo, Norway, andrea.arcuri@kristiania.no; Juan P. GaleottiDepto. de Computación, FCEyN-UBA, and ICC, CONICET-UBA. Argentina, jgaleotti@dc.uba.ar.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
XXXX-XXXX/2021/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(or insert probes) to provide better heuristic values, with different kinds of string distances. For example, EvoSuite [28] uses bytecode manipulation to replace all boolean methods in the class `java.lang.String` with its own custom versions. However, it can do this only when such methods are used directly before a bytecode branch instruction (e.g., derived from an `if` statement), and not under other circumstances (e.g., when stored in a boolean variable).

Unfortunately, there are several APIs in the core libraries of the different programming languages (e.g., Java) that can result in the flag problem. This is not only the case for methods that return a boolean, but also for methods that throw an exception when provided with an invalid input. For example, this can happen when we use API calls to transform a string into a date object (e.g., `LocalDate.parse(dateString)`), or into a number (e.g., `Integer.parseInt(intString)`). Here we would need a way beyond the branch distance to guide the search to find the right inputs that lead these APIs calls to execute without throwing an exception. Our techniques can be successfully applied to handle these cases as well.

When dealing with the system testing of REST web services [11], the test generation tools need to access to a schema (e.g., in OpenAPI/Swagger format [1]) to know which endpoints are available, and which type of query parameters and body payloads are expected as input. Unfortunately, schemas could be under-specified (i.e., some important info might be missing), especially when they are automatically derived based only on method signatures (e.g., when using libraries like SpringFox or SpringDoc for popular enterprise frameworks like Spring [3]). Testability transformations can be used here to analyze how HTTP requests are handled at runtime in the SUT, and use this info to improve the search. Furthermore, when dealing with testing of a RESTful API (and also web applications in general), for performance reasons there are behaviours in the SUT that should be avoided, like the HTTP server closing the TCP connections on user errors. This can also be solved with Testability Transformations, especially in the cases in which those servers do not expose such functionalities via configurations.

In this paper we provide the following main contributions:

- A novel approach for testability transformations of commonly used APIs. The presented approach is not limited to transforming methods only before jump instructions and it can also supports methods that throw exceptions (such as `parseInt`). It can be used to handle functions that return booleans and/or throw exceptions on invalid inputs.
- A novel technique to track test inputs that are used directly in our transformed methods without modifications. In such cases, feedback is given to the search algorithm to generate the needed data directly.
- A novel testability transformation which is specific for REST web services, and that is able to detect when the SUT is using HTTP query parameters, headers and body payloads not specified in the schema of the web service. This feedback is then used to include such HTTP inputs as part of the search.

We implemented our novel techniques as an extension of the EvoMaster tool, which is open-source. We evaluated our techniques on nine open-source REST web services, as well as an industrial web service. Our experiments show that our novel techniques improve performance significantly, both in terms of fault finding and code coverage. For example, on the industrial case study, with our novel techniques, it was possible to automatically detect seven new faults in it. The presented techniques could be used in other testing contexts (e.g., unit test generation) and programming languages (e.g., JavaScript and C#). However, without sound empirical evaluations, we cannot be sure that they will be as effective.

This paper is an extension of a conference submission [16]. In addition to providing more details and explanations of our techniques, we improved the handling of input tracking for strings when

two tracked inputs are compared with each other (Section 5). We also added handling of JSON parsing (Section 6), and transformations for improving TCP connection handling (Section 7). To better support the validity of our results, we employed a much larger empirical study (Section 8).

This paper is organized as follows. Section 2 provides background information to better understand the rest of the paper. Related work is discussed in Section 3. Our novel techniques for testability transformations are presented in Section 4. How to improve the performance with input tracking is discussed in Section 5. How to deal with under-specified REST API schemas using genotype expansion follows in Section 6. Section 7 shows how we can use testability transformations to improve performance issues during the search due to TCP connection handling. Section 8 presents our empirical study. Threats to validity are discussed in Section 9. Finally, Section 10 concludes the paper.

2 BACKGROUND

2.1 Branch Distance

In SBST, test generation is cast to an optimization problem, where we try to maximize metrics like code coverage. However, the control flow of a program could depend on complex predicates, like conditions in `if` statements. Only a tiny subset of the input space could lead to data for which such predicates evaluate to true. Without any heuristics, it would be unlikely to find the right data to maximize code coverage. Consider a trivial example like `if(x==42)`. Assuming `x` being an integer, then there is only 1 possibility out of 4 billions (i.e., 2^{32}) to get the right data, if chosen at random.

To help the search to find such data, heuristics are used to ‘‘smooth’’ the search landscape. Instead of a binary decision ‘‘branch covered’’ vs. ‘‘branch not covered’’, which would lead to fitness plateaus in the fitness landscape, we can provide heuristics to state how ‘‘close’’ an input is from making the predicate true. For example, a value 50 is heuristically closer than 100000 in covering `if(x==42)`, although the predicate evaluates as false in both cases.

In the 90s, Korel [41] defined a series of heuristics for numerical constraints, called *branch distance*. For example, $d(x == 42) = |x - 42|$. The idea is that the predicate is solved when $d = 0$, and the search process can use such distance as a metric to minimize. Besides numerical constraints, the branch distance has been then extended for logical operators [31] and string comparisons [6].

2.2 Java Bytecode

Java programs are compiled into an intermediary representation, that is called *bytecode*. Java programs are then run in a virtual machine, called JVM. Different programming languages do compile to JVM bytecode, like Kotlin and Scala.

The JVM is a stack-based machine, where bytecode instructions push and pop data from a stack (one per execution thread). Bytecode instructions are interpreted, and executed one at a time by the JVM. But for performance reasons, the JVM can decide to compile some parts of the bytecode into native code, on the fly, in the so-called Just-In-Time (JIT) compilation.

Consider a simple function like:

```
public boolean foo(int x){
        if(x==42){
            return true;
        }
        return false;
}
```

Its bytecode will look like:

```
L0
```

```
    LINENUMBER 2 L0
    ILOAD 1
    BIPUSH 42
    IF_ICMPNE L1
L2
    LINENUMBER 3 L2
    ICONST_1
    IRETURN
L1
    LINENUMBER 5 L1
    ICONST_0
    IRETURN
```

Here, the `LINENUMBER` instruction is used for debugging, to keep track in which source file the instruction was defined, and to give labels for jump instructions. Then, the content of input parameter `x` is pushed on the stack with the instruction `ILOAD 1`, followed by the pushing of the constant 42 with `BIPUSH`. Then, `IF_ICMPNE L1` pops the 2 top values from the stack: if they are not equal, then the execution flow jumps to the bytecode instruction labeled with `L1`, otherwise it continues with the next one (`L2` in this case). Here, either the constant `0` or `1` is pushed with `ICONST_*`, and then such value is returned with `IRETURN`. Note that the JVM has no concept of boolean values, and rather it uses the number `0` for false, and `1` for true.

When a Java (or Kotlin/Scala) program is compiled, `.class` files will be generated, which can be archived together into JAR/WAR files. During the program execution, the JVM will load such bytecode with the so-called *Class Loaders*. The JVM allows intercepting the loading of all classes with the so-called *Java Agents*, which allow manipulating such bytecode at runtime before is loaded into the JVM by the class loaders (e.g., using the library ASM[1]). This approach is useful for code coverage tools, e.g., to add coverage probes, and also by test generation tools like EvoSuite and EvoMaster to instrument the code, e.g., for branch distance calculations.

2.3 REST APIs

Currently, the most popular kind of web services are REST APIs [27], used, for example, by Google[2], Amazon[3], Twitter[4], Reddit[5], LinkedIn[6], etc. Besides providing functionality over the internet (e.g., see API portals like *ProgrammableWeb*[7]), REST APIs are also very common in enterprise backends, when microservice architectures are used [48, 50].

REST is not a protocol, but rather a set of guidelines on how to structure resources that are accessed on a network over HTTP(S). Resources are identified with URLs, and can be manipulated using the semantics of HTTP, e.g., `GET` requests to fetch data, `POST` to create new data, `PUT/PATCH` to modify existing data and `DELETE` to remove it. Inputs can be given in requests via path elements in the URLs, query parameters, HTTP headers and body payloads. Data can be transferred in any format, albeit currently one of the most common is JSON.

To make REST API easier to learn and use, it is a common practice to provide *schemas*, describing what endpoints are available in an API, and which input formats are supported for the inputs (e.g., the name and type of the query parameters, if any). Different schema standards exist, where

[1]https://asm.ow2.io/
[2]https://developers.google.com/drive/v2/reference/
[3]http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
[4]https://dev.twitter.com/rest/public
[5]https://www.reddit.com/dev/api/
[6]https://developer.linkedin.com/docs/rest-api
[7]https://www.programmableweb.com/api-research

OpenAPI/Swagger [1] is currently the most widely used. This format uses either JSON or YAML to define such schemas. Schemas can be written either manually, or automatically generated from the source code of the APIs (e.g., using SpringFox or SpringDoc for the popular enterprise framework Spring [3]).

2.4 EvoMaster

The open-source tool EvoMaster [8, 17] aims at automated test generation of system-level test cases for REST APIs, which is an important task to automate [9]. It is a SBST tool, using evolutionary algorithms like MIO [10] to evolve test cases maximing code coverage and fault finding metrics. It supports both white-box [11] and black-box [12] testing. For white-box testing, it uses several different SBST heuristics like the branch distance, not only applied on the predicates in the control flow of the SUT, but also in all SQL commands executed over a database (if any) [14]. To generate better test cases, EvoMaster can also exploit dependencies among the API resources [57].

EvoMaster is divided in two main parts: (1) a *core* process and (2) a *driver*. The core process contains all the basic functionality for a SBST tool, like the search algorithms, fitness functions, test generation outputs, etc. On the other hand, the driver is provided as a library, which engineers need to use to specify how to start, stop and reset the SUT. This is done with short configuration classes, that need to be implemented manually. However, the driver is also responsible of instrumenting the bytecode, which is done automatically. The core and the driver will run as separated processes, communicating over HTTP. This architecture will enable to support different programming languages, as a new supported language would just require a new driver library for it.

For white-box testing, EvoMaster currently supports APIs that run on the JVM, e.g., written in Java or Kotlin. It can output test suites in JUnit format, using the library RestAssured [2] for making the HTTP calls toward the SUT. The generated tests will use the manual configuration classes which use the driver library. This means that the generated tests are self-contained: the test suite files can start the SUT before any test is run, reset the state of the SUT before/after each test case execution (to make them independent), and stop the SUT once all tests are run. This means that the generated test cases can be used as well for regression testing, as can be added to the repository of the SUT, and run as part of a Continuous Integration process. Note that, for black-box testing, there is no need of any driver as EvoMaster could be run on any type of REST API regardless of their programming language. However, generated test suites would still be output in either Java or Kotlin.

3 RELATED WORK

3.1 Testability Transformations

"A testability transformation is a source-to-source transformation that aims to improve the ability of a given test generation method to generate test data for the original program" [36]. In the literature, different transformations have been proposed [34, 35], mainly to deal with *flag conditions* [21, 33] (i.e., branches in the code that depend on the value of a boolean constant). Flags in the code could depend on string operations [6], loop assignments [20, 22], nested predicates [47], calls to boolean functions [42, 43, 56] and non-integer comparisons [42].

Testability transformations can also be used to generate pseudo-oracles [46], which can be helpful to detect numerical inaccuracies and race conditions. Furthermore, besides SBST, testability transformations can also be useful for Dynamic Symbolic Execution [25].

A testability transformation does not need to preserve the original semantics of the transformed code, as it only needs to *"preserve test sets that are adequate with respect to some chosen test adequacy criterion"* [34]. Most of the aforementioned work apply non-semantics preserving transformations,

as targeting test generation for only single targets (e.g., a specific branch in the SUT). In contrast, in our work we aim at system testing where whole test suites are generated (e.g., as done in [29] for unit testing) aimed at maximizing coverage over the whole SUT. In such context, many of those non-semantics preserving transformations would not be applicable. This is because every single line and branch is a testing target in our context. When applying those transformations on a given target, the other targets could be affected as well, as such transformations could remove some of those other lines/branches, or modify them in a way that the adequacy of the chosen test criteria could be no longer assessed (as those transformations do not preserve the original semantics). For this reason, a main difference in our work is that we must preserve the semantics of the SUT. Furthermore, we provide several novel transformations for API calls that have not be investigated before in the literature.

3.2 Taint Analysis and Seeding

"Dynamic taint analysis (also known as dynamic information flow analysis) consists, intuitively, in marking and tracking certain data in a program at run-time" [24]. Taint analysis has been used in many different contexts, such as information security, program understanding, software testing and debugging [53]. Taint analysis has been particular useful for security, as it *"establishes whether values from untrusted methods and parameters may flow into security-sensitive operations"* [54]. In the literature, there has been many applications of taint-analysis [53], and in particular in recent years on security evaluations of Android applications (e.g., [38, 44, 49]).

In this paper, we use a basic, lightweight form of taint analysis, in which we track input variables when used in API calls. The novel contribution in this paper is on how we use such information to enhance SBST. We feed this information back to the search at runtime, modifying the genotype of the evolving test cases and the type of mutation operators applied to them, based on how such inputs are used in the SUT.

In some regards, our novel technique could be considered as an improvement over our previous *seeding* of values observed at runtime [52], done for unit testing. There, constants (e.g., strings and numbers) found in the bytecode (or observed at runtime in methods like `String.equals`) are added to a pool, and the search can use values from such a pool when sampling new individuals. On the one hand, this lacks the link to the actual inputs that should use such values (i.e., there is no tracing), and the use of the right value for a given input in the test is based on chance. The bigger the pool of constants is, the lower is the probability of randomly sampling the right value. Furthermore, these seeding techniques would not directly work (without novel extensions) for dealing with methods like `Integer.parseInt` or `LocalDate.parse`. On the other hand, these seeding strategies can be applied even when inputs are modified before being used in the API calls (which is a current limitation of our novel approach). Therefore, those seeding strategies could be still useful in this context, although how to best integrate them would be a matter of future work.

3.3 Testing REST APIs

In recent years, there has been an increasing interest in the research community about test automation for REST APIs, with several techniques that have been proposed [18, 26, 32, 39, 45, 55], with open-source tools like RESTest[8] and RESTler[9]. However, all of these techniques are black-box. To the best of our knowledge, only EvoMaster does support white-box testing, in which the source code of the SUT can be analyzed to achieve better results. Comparisons with other white-box techniques are therefore not possible. Currently, EvoMaster is also the only tool that uses SBST

[8]https://github.com/isa-group/RESTest
[9]https://github.com/microsoft/restler-fuzzer

Table 1. Roadmap for the different testability transformations presented in this paper.

Section	Objective	Involved Classes
4	Flag Problem	Boolean, Byte, Collection, Date, Double, Float, Integer, LocalDate, LocalDateTime, LocalTime, Long, Map, Matcher, Object, Pattern, Short, String
5	Input Tracking	Same as in Section 4
6	Under-Specified Schemas	GSON, HttpServletRequest, WebRequest
7	TCP Issues	AbstractEndpoint, Http11Processor

and the advances of evolutionary computation, e.g., by using specialized search algorithms like MIO [10].

4 TESTABILITY TRANSFORMATIONS USING METHOD REPLACEMENTS

To collect coverage metrics, tools like EvoMaster need to instrument the code of the SUT. This can be done automatically by manipulating the bytecode of the Java classes when they are first loaded into memory. In our approach, we implemented a series of custom classes with method replacements for some existing APIs in the JDK. Every time a SUT class is loaded, we check if it uses any method M for which we have a replacement R. If so, we remove M from the bytecode, and replace it with R. For example, a call to `String.equals` would be replaced by our custom `StringClassReplacement.equals`.

These method replacements are employed to achieve different objectives: address the flag problem with new distance heuristics (Section 4), enable input tracking (Section 5), deal with under-specified OpenAPI schemas (Section 6) and issues with the closing of TCP connections (Section 7). Table 1 provides a roadmap for what will be covered in these sections. The heuristics involving strings, presented here in Section 4, are based on existing work [7]. All the rest (especially what presented in Section 5 to 7) are novel contributions of this article.

To deal with the flag problem in existing APIs, all our replacement methods are static, and have the same return types as their original versions. However, the inputs are different. If the original method is non-static, then the first parameter in R is the caller of the original method. The last parameter is a string object with a unique id, based on the SUT class name and instruction line in which M was called. For example, given in `java.lang.String` the signature:

```
public boolean equals(Object anObject)
```

then our method replacement R for it would have signature:

```
public static boolean equals(String caller,
  Object anObject,
  String idTemplate)
```

A call to `x.equals("foo")` would hence be transformed into a call to `StringClassReplacement.equals` with arguments `x`, `"foo"` and `"name-line"`, where `"name-line"` would be an actual class name and line number where in the SUT the replacement is done. Such `idTemplate` needs to be pushed on the JVM stack before doing the `INVOKESTATIC` call on `StringClassReplacement.equals`.

All of our method replacement Rs have the same semantics as the original version Ms. For the same input, they either give the same output, or throw exactly the same kind of exception. Every time a replaced R is called at runtime during the search, before returning its value (or before throwing an exception), we compute a heuristic distance h. Such distance is similar in concept to

the *branch distance* [41], and it aims at determining how far was the input from making R returning either `true` or `false`. The goal here is to prevent fitness plateaus, where we want to reward test inputs that get heuristically "closer" to cover the given testing target.

Once such heuristic distance is computed, we create two new testing targets based on `idTemplate`, e.g., named `"name-line-true"` and `"name-line-false"`. These new testing targets will be added to the fitness function, and will be part of the search, like any other coverage target (e.g., for lines and branches). In other words, the search will reward the generation of at least one test case in which R evaluates to `true`, and at least one test case in which R evaluates to `false`. These tests can then be useful during the search to cover other targets, like branches.

Besides replacing methods that return a boolean, a novel contribution in this article is that we also handle exceptions. The bytecode manipulation is the same, it is just that the computed distance has a different meaning, i.e., how far the input was from throwing an exception and not throwing an exception. For example, given the method `parse` in class `java.time.LocalDate` with the following signature :

```
public static LocalDate parse(CharSequence text)
```

we provide a replacement class named `LocalDateClassReplacement` with a method:

```
public static LocalDate parse(CharSequence input, String idTemplate)
```

In this new method, a heuristic distance is computed to check how close a string is from being a valid date in the format `YYYY-MM-DD`. Two new targets are then added to the search.

Technically speaking, such method replacements could be provided for all the methods in the API of the JDK that either return a boolean, or throw an exception if the input is invalid. The challenge then would be to define the right heuristic distances for the different kinds of methods. In this paper, we provide method replacements for some of the most common APIs, e.g., all of the API method calls that were present in the SUTs of our case study.

In EvoMaster, each testing target has a heuristic score $h \in [0,1]$, where 1 means a target is covered. All other values represent the target not being covered, where 0 is the worst heuristic score. Values closer to 1 still represent non-covered targets, but they are heuristically closer to cover the target. For each method replacement we create two targets, e.g., one target to represent the evaluation of the function to `true`, whereas the other target to represent the evaluation to `false`. Similarly, we create targets for representing a method throwing or not throwing an exception. If such a call is never reached during the test execution, both targets get score 0. If on the other hand that method call is reached, one of the two targets will necessarily get the score 1 (representing the target being covered), whereas the other target will get a score lower than 1 (representing the target not being covered), e.g., $b = 0.1$. Observe that, if the value 0 is used to represent a covered target, it would just be a matter of a simple function transformation to compute the branch distance d, (e.g., $d = 1 - h$). Therefore, adapting the presented heuristic distances to other contexts is rather straightforward.

Here, we discuss some of the details of the heuristic h computations for the `true` targets. The values for the `false` targets will usually be just some constant greater than 0 (to distinguish from the cases in which the method replacement execution is not even reached, e.g., $b = 0.1$) if the method returns true (and 1 otherwise). Similarly, we discuss the cases of the heuristic computations for the targets representing a method non-throwing an exception.

In the case of equality comparisons between two elements, we compute a distance d, where $d = 0$ means the two elements are the same. For example, comparing two numbers x and y would have distance $d = |x - y|$ (i.e., the same as in the typical branch distance [41]). Mapping it into $h \in [0,1]$, where 1 means the two elements are the same, can be done simply with $h = \frac{1}{1+d}$ (so, the

larger the distance d, the smaller h will be). However, to handle the case in which the distance cannot be computed (e.g., due to one element being null), we still need to provide a minimum b value to distinguish from the case in which the method is not even called because the execution flow does not reach it. Therefore, in this cases we use $h = b + (1 - b)\frac{1}{1+d} = b + \frac{1-b}{1+d}$. Note the scaling factor $(1 - b)$, as we need to guarantee $h \in [0,1]$.

Our method replacements are as follows:

Collection.isEmpty. Computes $h = \frac{1}{1+l}$, where l is the length of the caller collection. If $l = 0$, then the heuristic value is $h = 1$. The greater the collection's length, the closer to 0 the heuristic value will be. Here, given a non-empty collection for which this function returns false, our heuristic h rewards mutation operations that remove elements from such collection (as h increases), while penalizing operations that add new elements in it (as h would decrease).

String.isEmpty. Computes the same heuristic value as for `Collection`'s `isEmpty`. Therefore, mutation operations that add chars into the string will be penalized, whereas operations that remove chars from the string making it shorter will result in a better fitness score (as h increases).

String.equals. Similarly to what presented in [6], given two strings, the distance d is calculated as the sum of the character distance between each position in the strings. If string lengths differ, the maximum character distance (i.e., 2^{16}, as in the JVM chars use 16 bits) is used for the missing positions. In turn, the heuristic value computed is $h = b + \frac{1-b}{1+d}$, or $h = b$ if the argument is null or not a valid string. The more the second string becomes similar to the first, the better heuristic score would be obtained.

String.equalsIgnoreCase. The heuristic value is simply the value computed by `String.equals` using the lowercased versions of both strings.

String.contentEquals. The method compares the caller string to the `toString` representation of the input object parameter. The computed heuristic value h is therefore the same as the value computed by `String.equals`.

String.startsWith and endsWith. This method returns true if the string argument is a prefix (respectively suffix) of the caller string; false otherwise. The heuristic value is the one computed by `String.equals` using a prefix (respectively suffix) of the caller string with the argument's length n, i.e., the char distance is computed only for the first (and respectively last for `endsWith`) n chars. When the argument is a matching prefix, the distance on each of the first n chars would be zero, and so $h = 1$. Each mutation operation that leads to a closer match will result in a better h heuristics.

String.contains. Computes all the heuristic values of `String.equals` for all substrings of the caller string with argument's length. Given the caller string with length l, and input argument with length n, $l - n + 1$ heuristics h' based on `String.equals` will be computed (if $l < n$, only one is computed). The output heuristic value h for `contains` will be the highest of such heuristic values h'. The goal here is to reward mutations of the string where any substring becomes closer/similar to the input argument. However, we only need one match in the caller string, and so we can focus on the closest one to get a match (which is the substring with highest value h').

Byte.equals, Short.equals, Integer.equals, Long.equals, Float.equals, Double.equals, Char.equals. For numeric values, we compute the absolute difference between the two compared values, i.e., $d = |x - y|$, and then $h = b + \frac{1-b}{1+d}$, where b is needed for when the input argument is null, is of the wrong type, or any of the two values is infinite. Note that chars can be treated as 16-bit numbers when compared.

Date.equals, LocalTime.equals, LocalDate.equals, LocalDateTime.equals. If the compared reference is `null` or it is a non-null value that is not an instance of `Date`, then the computed heuristic is $h = b = 0.1$. Otherwise, both `Date` instances are compared as long values through the `getTime()` method, by calculating a distance d as absolute difference between those two long values. Then, $h = b + \frac{1-b}{1+d}$. The closer the two dates are, the smaller the difference between their two time instants will be, and so h increases. The approach for the other time-related classes (i.e., `LocalTime`, `LocalDate` and `LocalDateTime`) is the same, although the instant values are retrieved with their own available methods (e.g., `LocalTime.toSecondOfDay`).

Objects.equals. We leverage on the `equals` method replacements previously defined, by using `instanceof` to determine the type of the two input `Objects` that are compared. For example, in case two strings are compared, the `String.equals` heuristic value is computed. Same approach is used for all the other `equal` methods for which we have defined an heuristic, e.g., `Integer`, `Date`, and `Long`. If for the type of compared objects we have no custom heuristics, then we simply assign a flag: 1 for covered, and 0.1 for not covered.

Boolean.parseBoolean. This method returns `true` if the input string is equal to the string `"true"` without considering casing, otherwise it always returns `false`. Therefore, the heuristic for `parseBoolean` returns the heuristic value of `String.equals` to the lowercased input to the constant `"true"`.

Integer.parseInt, Long.parseLong. Each method parses a string into an `int` or a `long` value respectively. If the input string cannot be parsed or the parsed value cannot be represented within the range of values, a `NumberFormatException` is thrown. Our heuristic optimizes for a string containing an optional minus symbol (i.e., `"-"`), and a non-empty list of digits (i.e., `"0..9"`). For each symbol in the current string, the distance d' of each character (as a char) is compared to the expected symbol for that position, and sum together into a value d. The resulting heuristic value is then $h = b + \frac{1-b}{1+d}$. The distance d' for the first character in the input string will be the minimum of the char distance from `"-"` and any of the digits in `"0..9"`, but only if the string is of at least length equal to 2 (i.e., the string `"-"` itself does not represent a valid number, whereas `"-1"` is valid). For example, `"ab"` would have distance $d = min(|'a' - '-'|, |'a' - '0'|, \ldots, |'a' - '9'|) + min(|'b' - '0'|, \ldots, |'b' - '9'|)$. In general, given an input string s, where with $s[i]$ we specify the character of s at position i, we have $d = \sum_i min_{c \in C(i)}(|s[i] - c|)$, where $C(i)$ is the set of valid characters at position i. Here, we want to reward modifications of the input string to get it closer to a valid numeric representation. However, at each position in the string, the set of valid characters $C(i)$ can be different (e.g., `"-"` is allowed only in the first position). So, we focus on the most promising character (which is the one with lowest distance, and that is why we take the minimum).

Float.parseFloat, Double.parseDouble. Each method parses a string into a `float` or a `double` value, respectively. If the string cannot be parsed, a `NumberFormatException` is thrown. If the parsed value it is too large or it is too small (in absolute terms) it will be represented as $\pm\infty$ or ± 0, respectively. For simplicity, we do not currently support the exponential notation (i.e., inputs such as `"9.18E+09"`). The distance d for these methods is similar to the `parseInt` and `parseLong` but also expecting a single dot symbol. In other words, our heuristic optimizes for a string containing an optional minus symbol (i.e., `"-"`), followed by a non-empty list of digits (i.e., `"0..9"`) with an optional dot symbol (i.e., `"."`). If the input string contains no dot `"."` symbol, then, for each position, we need to compute the distance from `"."` as well (and take the minimum to compute d'). If there is one or more `"."`, such distance for `"."` is not computed, where d' for the its first

occurrence will be 0. As with `parseInt` and `parseLong`, the resulting heuristic value is $h = b + \frac{1-b}{1+d}$ where d is the distance computed as described above.

DateFormat.parse. This method returns a `Date` instance if the input string is parseable on the date format defined in this object. When the input string is not parseable, the method will signal a `ParseException`. If the pattern of the format matches *"year-month-day"* (e.g., `"2019-10-14"`) or *"year-month-day hour:minute"* (e.g., `"2019-10-14 13:45"`), a distance d is computed that represents how far is the input string to satisfy of the corresponding pattern. This is done similarly to what done for `Integer.parseInt`, by using the equation $d = \sum_i min_{c \in C(i)}(|s[i] - c|)$. To compute $C(i)$, the first 4 chars can be any digit, the 5th can only be a `"-"`, and so on. In turn, the final heuristic value is computed as $h = b + \frac{1-b}{d+1}$. If the pattern used by the `DateFormat` cannot be obtained, or the format does not match any of those supported (i.e., *"year-month-day"* or *"year-month-day hour:minute"*), we currently provide no gradient, i.e., simply $h = 1$ for covered and $h = 0.1$ for non-covered target.

LocalDate.parse. Similarly to the `parse` method of `DateFormat`, it parses an input string expected in the *"year-month-day"* into a `LocalDate` instance (e.g., `"2019-10-14"`). If the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as with `DateFormat.parse`.

LocalTime.parse. This method parses an input string of the format *"hour:minute"* or *"hour:minute:second"* into a `LocalTime` instance (e.g., `"13:45"`, `"13:45:30"`). Similar to the `LocalDate` parse method, if the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as with `DateFormat`'s `parse` method.

LocalDateTime.parse. This method parses an input string of the format *"year-month-day*T*hour:minute"* into a `LocalDateTime` instance (e.g., `"2019-10-14T13:45"`). Observe that the "T" character in the input string is case insensitive. If the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as the aggregation of `LocalDate` and `LocalTime` parse methods.

Date.after and before. This method tests if the current date is after (respectively before) the specified date. The heuristic value for this method is based on the branch distance to $v_1 > v_2$ (respectively $v_1 < v_2$), where v_1,v_2 are the return values of method `getTime()` on each `Date` instance.

LocalDate/LocalTime/LocalDateTime.isAfter and isBefore. These methods are equivalent to `Date.after` and `Date.before`, and our computed heuristic is the same (based on the numeric difference in the instant values of the two compared dates).

Collection.contains. If the argument is an instance of a class for which we provide custom heuristics for its `equals` method (e.g., `String` and `Integer`), then the heuristic value is the highest heuristic between the argument and any instance of that data type stored in the caller collection. If no such values matching the data type are stored, or simply the collection is empty, the returned heuristic value is 0.1. Similarly in concept to `String.contains`, we just need only one match on the whole collection to make this function to return true. Therefore, once the heuristic distance is computed on each element, the final h is based on the most promising, i.e., the closer to make such predicate true.

Map.containsKey. For handling this method, we compute the heuristic value of `contains` over the set of keys obtained by invoking `keySet` method.

Pattern, Matcher and String.matches. For matching regular expressions, we apply the same distance d as presented in [6], where then we have $h = b + \frac{1-b}{1+d}$.

Matcher.find. Gradient for the `find` method is provided by wrapping the input regular expression with the leading and trailing regular expression *"(.*)"* and reusing the gradient for the `matches`. With this transformation, the heuristic value will return the value to matching the input regular expression in any substring of the caller input.

When we have a replacement method R for M in a class C, we apply the replacement even for all implementations of M in the subclasses of C. For example, the method replacement for `Collection.contains` is applied to all the collections in Java that inherits from `java.util.Collection` (e.g., `ArrayList.contains`).

One case we do not handle is when a SUT class extends a JVM class for which we have replacement methods, and the keyword `super` is used. For example, the SUT might have have a class `MyList` that extends `java.util.Collection`. In its code, it could have a call like `super.contains(x)` inside its `contains` method. In this case, we apply no transformation. The problem is that, in bytecode, that call is an `INVOKESPECIAL`, which can be used only in the subclass. If we made a naive replacement with `CollectionClassReplacement` using `contains(caller, value, idTemplate)` (where `caller` is an instance of `MyList`), then we would not be able to call `caller.contains(value)` after computing the heuristic distance, as we would otherwise end up in an infinite recursion. In the JVM, it is not possible to call the actual implementation of `Collection.contains` on an instance of a subclass `MyList` outside of `MyList` itself. Problem is, that in our implementations of R we still need to call M to make sure that, given the same input, R returns exactly the same value as M.

An approach to solve this issue would be to replicate (e.g., copy&paste) the code of M inside R, and apply Java reflection to manipulate the internal variables of the `caller` instance. However, it would significantly complicate our techniques, making any tool deciding to using them harder to implement and maintain (e.g., if any new JDK release does change such code).

5 IMPROVING SEARCH THROUGH INPUT TRACKING

Assume that in the code of the SUT there is a string comparison like `x.equals("foo")`. Now, that variable `x` might depend on some of the inputs of the test case. For example, in the testing of a REST web service, `x` could be a query parameter in the request URL, or a JSON field in the HTTP body payload. Or maybe `x` is not related at all with the input of the current HTTP request. For example, a previous `POST` request could have written such value into a SQL database, and then a HTTP `GET` request would just read it afterwards.

Even if `x` depends directly on the input `z`, the value itself could go throughout several changes before reaching the statement `x.equals("foo")`. For example, different strings could be concatenated. Therefore, using `"foo"` directly as a value for `z` does not necessarily mean that `x.equals("foo")` will be evaluated as true. Here, the heuristic distance plays a major role to guide the search algorithm to find the right value of `z` to get `x.equals("foo")` true.

But what about the cases in which the input value is not changed during execution, and it is used directly as it is in some of our replacement methods (recall Section 4)? Our heuristic distances can provide gradient to the search, but it would be more efficient to just use `"foo"` directly as input.

Our approach is to analyze if any such case does indeed happen, by tracking all the inputs to our replacement methods. If indeed we detect that any such input is used directly in a replacement method, then we can mutate directly the test cases to use the needed values.

```java
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;
import java.time.LocalDate;
import java.util.Arrays;
import java.util.List;

@RestController
@RequestMapping(path = "/api/testability")
public class TestabilityRest {
   @GetMapping(path="/{date:\\d{4}-\\d{1,2}-\\d{1,2}}/{number}/{setting}",produces
     =APPLICATION_JSON_VALUE)
 public String get(@PathVariable("date")String date,
  @PathVariable("number")String number,
  @PathVariable("setting")String setting) {
  LocalDate d = LocalDate.parse(date);
  int n = Integer.parseInt(number);
  List<String> list = Arrays.asList("Foo", "Bar");
  if(d.getYear() == 2019 && n == 42 && list.contains(setting))
   return "OK";
  else
   return "ERROR";
 }
```

Fig. 1. First code example of a REST controller in Java Spring framework, with one HTTP GET endpoint.

Figure 1 shows a code example for a Java class, in which a REST endpoint is defined using the popular Spring Framework [3]. When a HTTP call is made toward such endpoint, three `String` variables are extracted from the URL of the HTTP request. One of the strings is parsed to a date, whereas another to a number. The string `"OK"` is returned only if the date is a valid date with year 2019, the number is 42, and the last string is present in an existing list of strings. Generating a test case that returns `"OK"` is not trivial, because `LocalDate.parse` and `Integer.parseInt` will throw exceptions when called with non-valid inputs. Furthermore, although there exist testability transformations for `String.equals`, those cannot be directly used in the case of *List.contains*, because API classes loaded with the Java bootstrap classloader cannot be instrumented.

Consider such example in Figure 1, where based on the REST schema the search evolves three different string variables (*genotype*), which then are used to create a valid URL when a HTTP call is made in the fitness evaluation (*phenotype*). Consider the variable `date`, that based on the REST schema would be of type *string*. Once EvoMaster executes an HTTP call as part of a test case fitness evaluation, the SUT running Spring will read such incoming HTTP request (done in the HTTP server Tomcat), and extract all the needed information (e.g., the *path* component of the HTTP request). Based on the path component, the Spring framework would determine that `TestabilityRest.get` should be the handler for such HTTP request. The string object *date* would then be checked against a regular expression (see Figure 1). And such check would be made in a `Matcher.matches` call. Such method is one for which we do provide a replacement (recall Section 4). As the string variable `date` in the genotype of the test case is used directly in the replacement method for `Matcher.matches`, we could inform the search to directly evolve strings for `date` based on the regular expression $\backslash d\{4\} - \backslash d\{1,2\} - \backslash d\{1,2\}$. As EvoMaster has support for sampling and mutating strings based on regular expressions, this would be more efficient than mutating a standard string and hoping it would match such regular expression by chance.

To achieve this, there are some challenges that we need to solve first. For example, a check on inputs can be outside the main code of the SUT, i.e., in a third-party library. This is the case of `Matcher.matches` deep inside the Spring's library code. On the one hand, we need to make such method replacements even in third-party libraries. On the other hand, we do not want to create new search targets for those libraries, as that is not the software we want to maximize coverage for. The amount of code of the third-party libraries can be much, much more than the code of the SUT (we will show some statistics on this in Section 8). Even if specialized search algorithms such as MIO [10] can handle large numbers of optimization objectives, adding unnecessary search targets could hamper the search. Furthermore, these extra targets could lead to generate more test cases in the final test suite given as output at the end of search, covering different paths in such third-party libraries which might not be of interest to the user. In this case, the solution is relatively simple: we apply the method replacements to all classes, both in the SUT and the third-party libraries; however, the new testing targets are only created if the method replacement is done in a SUT class; finally, our *input tracking* in the method replacements is done in all the classes.

A further issue is how to detect that a string value in the execution of the SUT is related to a variable evolved in the genotype of the test case inside the search algorithm. For example, for that endpoint in Figure 1, EvoMaster would have three genes of type *string*. However, what the SUT would see is just an HTTP message read by the HTTP Tomcat server. If the genotype of a test case is, for example, the three string values (‘‘*a*”,‘‘*b*”,‘‘*c*”), what the SUT would read from the TCP socket would just be:

```
GET /api/testability/a/b/c HTTP/1.1\r\n
Host:localhost\r\n
\r\n
```

In other words, we need some way to inform our instrumentation runtime of the three distinct values (‘‘*a*”,‘‘*b*”,‘‘*c*”) to track. Unfortunately, as in EvoMaster the search algorithm and the SUT are running on different processes [8, 11], sending such information over TCP before a test evaluation might incur in a non-negligible time overhead. Furthermore, even if such information was sent, then there would still be the problem to determine inside the replacement of `Matcher.matches` if a given string value ‘‘*a*” is part of the test inputs, or if just an unrelated string constant used in the code of the SUT or a third-party library (and so changing the genotype by adding the regular expression info could have negative side effects).

A complete static analysis of the SUT and all third-party libraries might answer this kind of questions. However, for what we need for test generation, a simpler, light-weighted approach would suffice. Given a certain probability P, when in EvoMaster we mutate a string gene, in our approach we rather replace it with the value `"evomaster_x_input"`, where x is a unique number. The idea is to create string values that are very unlikely to be used in the SUT source code, and that then can be easily detected by our replacement methods. For example, we could check if the given input in a replacement method does match the regular expression $evomaster_\backslash d+_input$. This has the benefit of not needing to send any extra information to the instrumentation runtime before a test case is evaluated. Although it would be very unlikely that a modified/changed or unrelated input would match that regular expression, even if it happens it would not be a major issue. If the fitness value of the test would not improve, the test case would simply die out during the evolutionary search.

With such an approach, given the test case (‘‘*a*”,‘‘*b*”,‘‘*c*”), a string gene like ‘‘*c*” could be mutated into ‘‘*z*” (e.g., a regular, standard mutation in the default version of EvoMaster), while ‘‘*a*” could be mutated into `"evomaster_0_input"` with some probability P. The resulting test case would hence be (‘‘*evomaster_0_input*”,‘‘*b*”,‘‘*z*”). When this test case is going to be evaluated, the value

`"evomaster_0_input"` would be extracted from the HTTP request, and then would end up as input to our `MatcherClassReplacement.matches`, which can recognize it is a variable that should be tracked. Once the test case is completed, the instrumentation runtime would then inform back the search algorithm that the input `"evomaster_0_input"` was matched against the regular expression $\backslash d\{4\} - \backslash d\{1,2\} - \backslash d\{1,2\}$. This information is then used to change the gene type of that input from *string* to *regex*. Next time the test case is mutated, the value `"evomaster_0_input"` would be automatically mutated into a valid string matching that regular expression (e.g., `"3156-42-77"`). Further mutations on this gene would still generate strings matching that regular expression, like the `"77"` could be mutated into a `"78"`, but not into a `"7a"`.

Mutating a string into a `"evomaster_x_input"` value can be useful at the beginning of the search, but it could be disruptive for the cases in which we need to evolve complex strings for constraints not handled by our *input tracking* technique. Therefore, given a starting probability P, we gradually decrease it throughout the search down to 0 when in the MIO [10] algorithm (which is the default one used in EvoMaster) its ‘‘focused search’’ starts after 50% of search budget is used.

One problem though is that a variable could be used as input in many replacement methods. As soon as a `"evomaster_x_input"` would be mutated into an appropriate string for the first replacement method, the following replacement methods would not be able to recognize it any more, as likely not matching $evomaster_\backslash d + _input$.

Let us keep considering the case of the variable `date` in Figure 1. After its value was mutated into a string matching regular expression like `"3156-42-77"` due to its use as an argument in `Matcher.matches`, it is later used as input to `LocalDate.parse`. However, `"3156-42-77"` is not a valid date. Our novel heuristics in Section 4 would give gradient to evolve a string representing a valid date. However, as such input arrives to `LocalDate.parse` unmodified, it would be more efficient to rather inform the search to handle that gene not as a *regex*-type gene, but rather as a *date*-type gene (EvoMaster has support for mutating and sampling genes representing valid date strings). Such gene would then still be needed to evolve, due to the constraint `d.getYear() == 2019` in the SUT. However, that constraint is numerical (i.e., comparison of two integers) inside an `if` statement, and so would be handled directly by the *branch distance* (i.e., there would be direct gradient to modify the year component of the *date*-type gene into the value 2019, whereas mutations on the month and day components would have no influence on the fitness).

As regular expression constraints are not uncommon in REST web services, and because they can be quite complex, we applied the following approach. If based on our *input tracking* technique we transform a *string*-type gene into a *regex*-type gene, then we will track the values of those *regex*-type genes (and only those) as well besides $evomaster_\backslash d + _input$. This means that, before we evaluate the fitness of a test case, we need to send over TCP to the instrumentation runtime the information of which extra values to track, like `"3156-42-77"` in our example.

Not all the replacement methods in Section 4 need to handle our *input tracking* technique (e.g., `Collection.isEmpty`). As already stated, parsing of regular expressions would be mapped to *regex*-type genes, whereas parsing of dates would me mapped to *date*-type genes. Strings parsed to numbers are mapped to the respective number gene types, e.g., the input to a `Double.parseDouble` is mapped to a *double*-type gene, but still treated as string, e.g., quoted as a string `"42"` instead of just 42. String comparisons are mapped to *enum*-type genes, composed of all the constants the input strings are compared to. `String.equalsIgnoreCase` is treated specially, as it gets mapped into *regex*-type gene matching the string constant, but ignoring its case. For example, given the call `x.equalsIgnoreCase("a+b")` where x is an input represented by a *string*-type gene, then our *input tracking* technique would transform it into a *regex*-type gene for the regular expression

```
@ApiOperation(value="Returns success/insuccess",
  notes="This method allows consumer registration.")
@RequestMapping(value = "/consumer/register",
  method = RequestMethod.POST)
public String addUser(WebRequest request) {

 boolean success = false;
 JsonObject response = new JsonObject();
 String username = request.getParameter("username");
 Consumer c = consumers.findByUsername(username);

 if (c == null) {
  String pass = request.getParameter("password");
  String email = request.getParameter("email");
  String name = request.getParameter("name");
  String lat = request.getParameter("latitude");
  String lon = request.getParameter("longitude");
  c=new Consumer(name,username,pass,email,lat,lon);
```

Fig. 2. Snippet of a REST controller from the class `ConsumerController` in the case study *proxyprint*.

$(a|A)\backslash Q+\backslash E(b|B)$. In other words, each character with different lower and upper case is in an or | between such two cases, while regex control characters are quoted inside a $\backslash Q \backslash E$.

Let us consider the case when two different tracked inputs are compared, e.g., `x.equals(y)`. We can detect these cases, and handle them specially, by marking the two variables in the test case as ‘‘bound’’. During the search, when we mutate any string gene, we check if any other gene is bound to it. If so, those genes get modified as well as part the mutation and assume the exact same value as the mutated string gene.

6 GENOTYPE EXPANSION FOR REST APIS

To generate test cases for a REST web service, tools like EvoMaster need to know which endpoints are available, which query parameters they expect, what is the type and structure of the body payloads, etc. All this information can be provided with a schema, where OpenAPI/Swagger is likely the most used [1].

There are two approaches to generate a schema: either by writing it manually, or by generating it automatically from the SUT code using a tool. In both approaches there can be a mismatch between what specified in the schema and what the web service actually does. Even when an automated tool is used to infer the schema from the SUT code, not only such tool might have bugs, but also there might be cases in which it cannot infer the whole correct schema.

In the case of a Spring application, libraries like Springfox can generate the schema automatically, by analyzing static information like annotations such as `@PathVariable` and `@GetMapping` (e.g., recall Figure 1). However, they would not be able to determine information that depends on the code execution of the SUT. Consider, for example, the code snippet in Figure 2, from the case study *proxyprint*. Here, the needed data from the HTTP request is not injected directly in the controller (which is the common practice, e.g., Figure 1). On the other hand, a whole `WebRequest` is injected, from which query parameters like `"username"` are then extracted. A schema generation tool such as Springfox might not be able to infer that parameters such as `"username"` should be part of the schema. This is because it would not be enough to just use Java reflection to analyze the annotations on the SUT methods and their parameters. It would require code analysis of all

```
@ApiOperation(value = "Updates the consumer information",
              notes = "This method allows consumers to update their personal
    information.")
@Secured("ROLE_USER")
@RequestMapping(value = "/consumer/info/update", method = RequestMethod.PUT)
public String updateConsumerInfo(Principal principal, HttpServletRequest request)
    {
     boolean success = false;
     JsonObject response = new JsonObject();
     String requestJSON = null;
     Consumer editedConsumer= null;
     try {
          requestJSON = IOUtils.toString(request.getInputStream());
          editedConsumer = GSON.fromJson(requestJSON, Consumer.class);
     } catch (IOException e) {
          e.printStackTrace();
     }
     if(principal.getName()!=null) {
          Consumer c = consumers.findByUsername(principal.getName());
          if(c!=null) {
               if(!c.getName().equals(editedConsumer.getName())) {
                    c.setName(editedConsumer.getName());
               }
```

Fig. 3. Snippet of a REST controller from the class `ConsumerController` in the case study *proxyprint*.

the method instructions. Furthermore, whether some query parameters are read might depend on non-trivial computation, like for example the `if(c == null)` branch depending on a SQL query into the database, i.e., the `consumers.findByUsername` in Figure 2. In that specific case study *proxyprint*, no query parameter information was inferred by Springfox and added to the schema.

To overcome this limitation, we track all usages of the `WebRequest` objects at runtime. This is done by a simple testability transformation in which we store the input values of all method calls like `getParameter` and `getHeader`, by replacing them with our own custom static methods. Once a test case execution is completed, and any such method was called, we *"expand"* the genotype of the test cases by adding genes representing those query parameters. For example, once such `/consumer/register` endpoint is called for the first time with no query parameters (as such info is not in the schema), the test case will get a new gene for the query parameter `username`, initialized with a random string. To avoid modifying the phenotype of an evaluated test, such new genes are marked as a *optional* genes (see [11] for details), disabled by default. When this test case is going to be mutated during the search, its optional genes could be mutated from disabled to enabled (and so added to the URL of the HTTP request), and their string content will be mutated as well as part of the search. This *genotype expansion* is applied every time new headers and query parameters are accessed. For example, an expansion occurs when the `if(c == null)` branch is evaluated as true, and so all the other parameters such as `password` and `email` are accessed in that code branch.

Besides query parameters, there can also be challenging to infer the proper type of the input body payloads of the HTTP requests (if any). Consider the code snippet in Figure 3, which shows one of the other endpoints in the *proxyprint* case study. Here, such endpoint expects a JSON payload with information about a consumer. However, the endpoint does not autowire such body payload directly, but rather it injects the whole HTTP request object, i.e., `HttpServletRequest request`. Therefore, Springfox is not able to infer the schema of the expected body payload. In the business

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@Table(name = "consumers")
public class Consumer extends User {

    @Column(name = "name", nullable = false)
    private String name;
    @Column(name = "email", nullable = true)
    private String email;
    @Column(name = "latitude", nullable = true)
    private String latitude;
    @Column(name = "longitude", nullable = true)
    private String longitude;

    @JoinColumn(name = "consumer_id")
    @OneToMany(cascade = CascadeType.ALL)
    private Set<PrintingSchema> printingSchemas;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "consumer")
    @Exclude
    private Set<PrintRequest> printrequests;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "consumer")
    @Exclude
    private List<Notification> notifications;

    @Column(unique = true)
    private Money balance;
```

Fig. 4. Snippet of the fields in the class `Consumer` in the case study *proxyprint*.

logic of the endpoint, the body payload is read as a byte-stream, converted into a string, and then a library is used to convert from such string into a `Consumer` object, which expects several different kinds of fields (see Figure 4).

To handle these cases, we use our testability transformations to intercept all the calls to `getInputStream()` in the HTTP request objects, and track them. Then, we consider the library GSON for JSON parsing, and intercept all its calls to `fromJson()`. If during a test case execution such methods are called, our instrumentation will inform the search about it. Depending on which class definition is used as input to `fromJson()` (e.g., the class `Consumer` in the previous example), we automatically create a schema definition for it, in the same format as in OpenAPI/Swagger. When creating such schema, we follow the same algorithm as in the GSON library, i.e., we use reflection to analyze all the fields, and skip the ones that are transient, or marked for exclusion via annotations (e.g., `@JsonIgnore`). With these object schemas, the search can then generate valid body payloads in the HTTP requests of the following test cases, e.g., by providing valid JSON objects that will not crash GSON when they are unmarshalled into a `Consumer` instance. These objects will then be evolved in the same way as if such definition was provided in the original OpenAPI/Swagger schema of the REST API. This is important, as the fields in such object can then be used in the control flow of the SUT (see Figure 3), e.g., in `if` statements.

Note that the JDK does not provide any native support for JSON handling. On the JVM, the two most popular libraries for JSON handling are GSON and Jackson. We currently provide support only for GSON. Supporting also Jackson is just a matter of technical/implementation effort, as it would follow the same approach.

7 TESTABILITY TRANSFORMATIONS FOR TCP PERFORMANCE

During the search, EvoMaster can generate millions of HTTP calls toward the SUT, depending on for how long the search is left running. Each HTTP call will be sent over a TCP connection. For performance reasons, unless one is doing performance or stress testing, it makes sense to re-use the same TCP connection. This can be achieved by adding, at each HTTP request, the header `Connection: Keep-Alive`. This tells the server to keep the TCP connection open, as more HTTP requests will be coming in shortly on it.

However, `Connection: Keep-Alive` is not enforced, it is just a "suggestion" for the server. To avoid denial of service (DoS) attacks, it is common for web servers to close the TCP connections in certain cases. For example, the popular Tomcat (which is the default server for SpringBoot), automatically closes the TCP connections after every 100 HTTP calls, even if those calls come with a `Connection: Keep-Alive` header. Furthermore, Tomcat also closes the TCP connections for any request that results in any of the following HTTP status codes: 400, 408, 411, 413, 414, 500, 501, 503.

For tools like EvoMaster that generate a high number of HTTP calls, this is a problem. TCP connections are Operating System (OS) resources. Allocating and de-allocating such resources can be expensive. For example, it can take a couple of minutes before the ports of a closed TCP connection can be re-used and marked as available by the OS. When making a TCP connection to a server, not only we need to specify the port on the server, but also a port on the local machine is opened to read the responses. Typically, such port would be an ephemeral one (i.e., a port not currently in use by the OS). In the OS, there can be only up to $2^{16} = 65536$ possible TCP ports. However, the ephemeral ones are even less: e.g., by default on Windows they are typically in the range $1025 - 5000$, whereas Linux uses $32768 - 60999$. If during the search the server keeps closing TCP connections, a new connection needs to be established by EvoMaster for the following HTTP calls using an available ephemeral port, possibly running out of local ephemeral ports if the OS is not fast enough in releasing them once their TCP connections are closed by the SUT server. If no new TCP connection can be established, then the search has to stop, as no new test case can be evaluated. This problem is further exacerbated when several instances of EvoMaster are run in parallel: e.g., during experiments by researchers on their local machines, or by practitioners that have several different APIs they want to test at the same time on the same machine.

When it comes to Tomcat, the closing of connections to prevent DoS attacks can be configured via the setting `MaxKeepAliveRequests` (which has default value 100), e.g., a value like -1 will simply tell Tomcat to deactivate such feature. However, closing the connection due to error status codes is not something that can be currently deactivated, as hardcoded in a function inside Tomcat.

Fortunately, testability transformations can be used here to prevent such issues. In particular, for Tomcat, we replace all the calls to `AbstractEndpoint.getMaxKeepAliveRequests()` with a function that always return -1, and all calls to `Http11Processor.statusDropsConnection(code)` to always return false.

8 EMPIRICAL STUDY

In this paper, we aim at answering the following research questions:
RQ1: How effective, in terms of code coverage, are our novel testability transformations on simple code examples?
RQ2: How often can our testability transformations be applied?

Table 2. REST web services used in the empirical study. We report the number of Java/Kotlin classes, lines of code (LOC), and number of HTTP endpoints.

Name	Classes	LOC	Endpoints
catwatch	69	5442	13
features-service	23	1247	18
languagetool	814	125862	2
proxyprint	68	7534	74
rest-ncs	9	602	6
rest-news	10	718	7
rest-scs	13	862	11
restcountries	21	1516	22
scout-api	75	7479	49
industrial	75	5687	20
Total	1177	156949	222

RQ3: How effective, in terms of code coverage and fault finding, are our novel testability transformations on open-source software?
RQ4: How effective, in terms of code coverage and fault finding, are our novel testability transformations on industrial software with complex input validation?

8.1 Artifact Selection

Our case study is divided in three parts:

- a set of three toy code examples,
- a set of nine open-source REST web services, and
- an industrial web service provided by one of our industrial partners.

We created three small code examples to make sure that our novel techniques work as intended. These toy examples are also helpful to better understand the addressed problem. The idea is that, without our novel techniques, these code examples should be hard to solve for an automated test generation tool. And indeed this was the case for EvoMaster, which was not able to fully cover these examples. On the other hand, once our novel techniques are employed, they should become trivial.

Table 2 shows some statistics on the RESTful APIs, like the number of classes (not including tests, nor third-party libraries) and HTTP endpoints in these SUTs. Due to a NDA, the industrial web service is simply referred with the label *industrial*. Regarding the nine open-source projects, these are a superset of the ones we used in our previous work on EvoMaster [10, 11]. To ease their re-use for experimentation, we collected them in a single repository on GitHub[10]. In this selection, there is a variety in terms of lines of code, number of endpoints and popularity (e.g., measured in number of stars on GitHub). For example, *restcountries* and *languagetool* are widely used (1.8k and 4.5k stars on GitHub, respectively, at the time of this writing).

8.2 Results on Code Examples

Our first example is the one we already discussed in Figure 1. Even if we do not consider REST APIs, this simple example would be challenging even for unit testing. For example, Figure 5 shows the result of running the unit test generation tool EvoSuite [28] on the `TestabilityRest` class for

[10] https://github.com/EMResearch/EMB

```
@Test(timeout = 4000)
public void test0() throws Throwable {
 TestabilityRest testRest0 = new TestabilityRest();
 try {
  testRest0.get("", "", "");
  fail("Expecting:DateTimeParseException");
  } catch(DateTimeParseException e) {
   verifyException("java.time.format.DateTimeFormatter", e);
  }
}
@Test(timeout = 4000)
public void test1() throws Throwable {
 TestabilityRest testRest0 = new TestabilityRest();
 try {
  testRest0.get(null,"TestabilityRest","PATCH");
  fail("Expecting: NullPointerException");
 } catch(NullPointerException e) {
  verifyException("java.util.Objects", e);
 }
}
```

Fig. 5. Tests generated by EvoSuite on the `TestabilityRest` class from Figure 1.

```
@Test
public void test_0() {
 given().accept("application/json")
 .get(sutUrl+"/api/testability/2019-03-04/42/Foo")
 .then()
 .statusCode(200)
 .assertThat()
 .contentType("application/json")
 .body(containsString("OK"));
}
```

Fig. 6. A test generated by EvoMaster, with our novel testability transformations, on the Spring application where the `TestabilityRest` class from Figure 1 is used.

one hour. Only two low-coverage unit tests were generated, throwing exceptions. This is because EvoSuite has no gradient in its fitness function to generate inputs to reach the `return "OK";` statement.

When dealing with system testing, there is a further complication. When Spring creates a REST handler based on the `@RestController` and `@GetMapping` annotations, it executes a validation check on the `date` variable. In particular, the `date` parameter is checked against the regular expression $\backslash d\{4\} - \backslash d\{1{,}2\} - \backslash d\{1{,}2\}$ even before the SUT method `TestabilityRest.get` is called. The check is done deep inside the Spring framework's library, using `Matcher.matches`. Furthermore, a string matching that regular expression is not necessarily a valid date. For example, a value like 42 for the month or day would be wrong (i.e., `LocalDate.parse` will throw an exception), although it would still match $\backslash d\{1{,}2\}$.

Figure 6 shows one system-level test case generated by EvoMaster when extended with our novel techniques. It was possible to generate the right HTTP call (using the library `RestAssured`) that returns `"OK"` in less than one minute. By applying our novel testability transformations, it was

```
@RestController
public class TTPaperParam {

    @GetMapping("/api/param")
    public String get(WebRequest wr){

        String param = wr.getParameter("param");

        if(param.equals("FOO")){
            return "OK";
        }

        return null;
    }
}
```

Fig. 7. Second code example of REST controller.

```
@Test
public void test_1()  {

    given().accept("*/*")
            .get(baseUrlOfSut + "/api/param?param=FOO")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("text/plain")
            .body(containsString("OK"));
}
```

Fig. 8. Test generated by EvoMaster on the API depicted in Figure 7.

possible to achieve full coverage on such SUT. Nine different test cases were generated covering different scenarios (but not displayed here in this paper due to space limitations).

In Figure 7, there is another code example, in which the value `"OK"` is returned only if a query parameter called `param` is given as input with value `"FOO"`. However, a `WebRequest` is used. This means that automated tools that generate OpenAPI schemas will miss the presence of such parameter. However, our testability transformation has no issue in handling these cases, which leads EvoMaster to quickly generate (in a few seconds) test cases like the one in Figure 8.

In the third and last example in Figure 9, we have a `POST` request that expects a JSON body payload, which is then unmarshalled into a `BodyDto` class instance. However, as the handler takes as input a `HttpServletRequest` instead of delegating to Spring to extract the body payload and unmarshall it into `BodyDto` (which would be simply achieved by giving as input `@RequestBody BodyDto dto`), such definition is missing from the generated OpenAPI schema. Without our testability transformations, tools like EvoMaster (and any black-box testing tool) cannot predict what is the proper input for the body payloads. However, with our novel techniques, such case becomes trivial, and can be solved in a few seconds, e.g., by generating tests like the one in Figure 10.

> **RQ1**: *thanks to our testability transformations, code examples which we could not fully cover before become now trivial.*

```
@RestController
public class TTPaperBody {

    @PostMapping(path = "/api/body"
            , consumes = MediaType.APPLICATION_JSON_VALUE)
    public String post(HttpServletRequest request) throws Exception{

        String json = IOUtils.toString(request.getInputStream()
                , StandardCharsets.UTF_8);

        BodyDto dto = new Gson().fromJson(json, BodyDto.class);

        if (dto.x > 0) return "OK";
        else return null;
    }
}

class BodyDto{
    public Integer x;
}
```

Fig. 9. Third example of REST API, with also a DTO (Data Transfer Object) class representing the input body payload.

```
@Test
public void test_1()  {

    given().accept("*/*")
            .contentType("application/json")
            .body(" { " +
                    " \"x\": 985.0 " +
                    " } ")
            .post(baseUrlOfSut"+/api/body")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("text/plain")
            .body(containsString("OK"));
}
```

Fig. 10. Test generated by EvoMaster on the API depicted in Figure 9.

8.3 REST API Experiment Settings

For the RESTful APIs, we carried out two distinct sets of experiments. In the first one, we run EvoMaster on the nine open-source services, considering five different configurations:

- **Base**, the default version of EvoMaster, with none of our novel techniques presented in this article.
- **M**: using *Method Replacement* (Section 4) and *Genotype Expansion* (Section 6).
- **M+T**: as in *M*, but also using *Input Tracking* (Section 5). For the probability *P* of applying the mutation, we considered three different values: 0.1, 0.5 and 0.9.

Table 3. Number of times, per SUT, in which a method was replaced with our testability transformations (Section 4). We distinguish whether this happened in one of the core classes of the SUT, or in a third-party library. We also report how often the testability transformations were used to collect data on what accessed from the HTTP requests (Section 6) and to improve the TCP handling (Section 7). For the replacements done in the SUTs, we also report the average number of such replacements per class, and also how many x LOCs a transformation is applied on average. Note: due to the results for *rest-ncs*, the aggregated statistics show the median of the averages instead of the average of the averages.

SUT	Replacement		HTTP	SUT Replacement	
	SUT	Third-party		Avg. Per-Class	Avg. 1-per-x LOCs
catwatch	33	4522	3	0.47	164.9
features-service	6	4081	3	0.26	207.8
languagetool	2794	662	0	3.43	45.0
proxyprint	113	4076	24	1.66	66.6
rest-ncs	0	2690	3	0.00	-
rest-news	4	4305	3	0.40	179.5
rest-scs	86	2678	3	6.61	10.0
restcountries	46	3581	3	2.19	32.9
scout-api	52	3718	0	0.69	143.8
Total/Median	3134	30313	42	0.69	143.8

To take into account the randomness of the search algorithm, each experiment was repeated 30 times with different random seeds, for a total of $9 \times 5 \times 30 = 1350$ runs. As stopping criterion, we chose the same amount of fitness evaluations as done in our previous work [10, 11], which consisted of 100,000 HTTP calls per run. Depending on the SUT and the hardware employed, each run would take roughly between 20 and 60 minutes.

The second set of experiments on the industrial case study was with the same kind of configurations, i.e., *Base*, M and $M + T$, but only one value for P (so, three configurations in total). As such industrial case study does complex, heavy input validation on most of its endpoints, we experimented with longer search budget. In particular, we considered a stopping criterion of 1,000,000 HTTP calls, which roughly required 12 hours per run on the employed hardware. Experiments were repeated 30 times, for a total of $3 \times 30 = 90$ runs.

For both sets of experiments we relied on EvoMaster with its default settings except for the HTTP call budget. The results of the experiments were analyzed following the guidelines in [13]. In particular, we used the Wilcoxon-Mann-Whitney U-test, and the Vargha-Delaney $\hat{A}_{12}$ effect size.

8.4 Results on Open-Source Software

Table 3 shows how often our transformations were applied on the open-source SUTs. These only count the number of times a method was replaced in classes loaded by the SUT during the search. For example, if a class is never used (and so never loaded in the JVM of the SUT), then it is not instrumented by the EvoMaster runtime, and would not be counted in that table. The numbers reported are the total number considering all the 1350 runs.

The number of applied testability transformations varies significantly from SUT to SUT, like from 0 for *rest-ncs* to 2794 for *languagetool*. However, it is clear that there is a very large number of third-party libraries involved in the execution of the SUT. On each case study, there are between 662 and 4522 method calls that were replaced with our testability transformations. This is not surprising,

Table 4. Average (over 30 runs) bytecode-level line coverage per configuration, on each of the open-source SUTs. Results are ranked per SUT, from 1 (best configuration) to 5 (worst configuration). Values in bold are the best (i.e., rank 1).

SUT	Base	M		$M+T$	
			0.1	0.5	0.9
catwatch	503.6 (4)	507.0 (3)	500.4 (5)	507.3 (2)	**509.3 (1)**
features-service	169.9 (5)	**189.7 (1)**	187.5 (2)	185.9 (3)	180.9 (4)
languagetool	4864.3 (5)	6820.9 (4)	9891.1 (3)	10137.8 (2)	**10156.0 (1)**
proxyprint	744.3 (4)	768.6 (2)	**787.7 (1)**	763.2 (3)	738.4 (5)
rest-ncs	250.8 (5)	251.2 (2.5)	251.1 (4)	251.2 (2.5)	**251.3 (1)**
rest-news	**94.3 (1)**	92.7 (3)	92.0 (4)	91.6 (5)	92.9 (2)
rest-scs	180.5 (5)	220.7 (4)	236.8 (3)	240.5 (2)	**240.6 (1)**
restcountries	385.0 (5)	396.2 (3)	395.9 (4)	397.6 (2)	**398.8 (1)**
scout-api	631.8 (5)	632.1 (4)	**640.6 (1)**	635.3 (3)	635.4 (2)
Average Rank	4.3	2.9	3.0	2.7	2.0

Table 5. Average (over 30 runs) number of found faults per configuration, on each of the open-source SUTs. Results are ranked per SUT, from 1 (best configuration) to 5 (worst configuration). Values in bold are the best (i.e., rank 1).

SUT	Base	M		$M+T$	
			0.1	0.5	0.9
catwatch	20.9 (5)	21.1 (4)	21.3 (3)	**21.7 (1)**	21.4 (2)
features-service	33.9 (3.5)	34.0 (2)	**34.0 (1)**	33.9 (3.5)	33.8 (5)
languagetool	8.1 (5)	25.2 (4)	52.5 (3)	**61.4 (1)**	56.0 (2)
proxyprint	95.4 (5)	101.9 (2)	**102.3 (1)**	100.4 (3)	99.4 (4)
rest-ncs	**5.1 (1)**	5.1 (2)	5.0 (4)	5.0 (4)	5.0 (4)
rest-news	7.0 (2)	**7.1 (1)**	7.0 (3)	6.9 (4)	6.9 (5)
rest-scs	**11.0 (1)**	11.0 (2)	10.8 (3)	10.7 (4)	10.4 (5)
restcountries	**23.0 (3)**	**23.0 (3)**	**23.0 (3)**	**23.0 (3)**	**23.0 (3)**
scout-api	102.9 (4)	102.6 (5)	105.5 (2)	103.2 (3)	**105.9 (1)**
Average Rank	3.3	2.8	2.6	2.9	3.4

considering the complexity of handling HTTP requests (e.g., using Tomcat) and accessing databases (e.g., using Hibernate), with beans autowired by Spring (which is used by seven of the SUTs).

Regarding our *Genotype Expansion* technique (Section 6), it was applicable 21 times, but only on the *proxyprint* case study. Seven SUTs use Tomcat (as they are SpringBoot applications), which leads to 3 call replacements each for handling the TPC connections (Section 7).

> **RQ2**: *our testability transformations could be applied 3134 times on the SUT classes, and more than 30,000 times on the classes of the third-party libraries.*

EvoMaster optimizes for several criteria, like line coverage, branch coverage, coverage of HTTP status codes per endpoint, detected faults, etc. In previous work (e.g., [14]), we usually reported such total number of covered testing targets to measure and compare the effectiveness of our techniques. However, our testability transformations presented in Section 4 introduce new testing

targets. Looking at the total of covered targets would hence be unsound when doing comparisons with the default *Base* technique that does not do such instrumentation (and this was a mistake done in the previous version of this study [16]). Due to space limitations, we do not report each metric individually, but rather show the results for the two most important ones: line coverage and fault detection.

Regarding line coverage, EvoMaster computes it at bytecode level, counting the number of `LINENUMBER` instructions that are covered. However, for system testing, there is a challenge in computing the total percentage values, as the total number of `LINENUMBER` instructions per class can be derived only for the classes that are actually loaded into the JVM during the search. If some classes are never loaded (e.g., because their execution depends on code that was not reached), then we cannot reliably compute the percentage of achieved line coverage. Notice that 7 out of 9 SUTs use SpringBoot, whose custom class-loader does a scan of the whole classpath to analyze all the classes that need to be instantiated as proxy-beans. For those, all the SUTs classes will be loaded into the JVM, so for them this is not a problem. However, for the other two SUTs (i.e., *scout-api* and *languagetool*), some classes might have not been loaded during the experiments. This is the reason why we are also going to report the actual total number of covered `LINENUMBER` instructions. When we report percentages, those are calculated by taking into account the highest number of loaded classes considering each of the 150 experiments per SUT.

Regarding fault detection, EvoMaster counts each returned 500 status code per different endpoint, distinguished based on the last executed statement in the business logic of the SUT. This allows to detect possible different faults in the SUT that can lead to a crash in the same endpoint (which is among one of the main advantages of white-box testing compared to black-box techniques). Furthermore, EvoMaster can detect when a returned response is not matching what specified in the OpenAPI/Swagger schemas.

Table 4 shows the results of comparing the five different configurations of our experiments on each of the nine open-source SUTs in terms of line coverage. For most SUTs, the configuration $M + T$ with $P = 0.9$ gives the best results (as it has the lowest rank 2.0). But, for *proxyprint*, it seems like our input tracking has some negative side effects, as best results are for $P = 0.1$, and, the higher the P the lower the performance. However, it still gives better results than M. But this is not the case for *features-service*, where although M is better than *Base*, the use of input tracking worsens the results. This could be explained if indeed our novel heuristics give better gradient to the search, but, at the same time, they involve inputs that are transformed before used in the replacement methods. In those cases, using string values like `evomaster_x_input` would be just a waste of resources.

The default version of EvoMaster, i.e., *Base*, gives the worst results on average (worst rank 4.3). However, out of nine SUTs, there is one where *Base* gives the best results, which is *rest-news*. But, the difference in performance is minimal, i.e., only 2 lines out of 94 covered. On the other hand, for example for *languagetool* the coverage more than double with $M + T$, i.e., 10,156 vs. 4,864 covered lines.

Table 5 shows the results regarding the found faults in these SUTs. Even for such metric, $M + T$ is the configuration that gives the best results. On the one hand, interestingly there is a negative effect regarding P, where low values give better results. On the other hand, the difference in actual number of found faults is quite minimal among the SUTs, for all but *languagetool* and, to a lesser degree, *proxyprint*.

From the results of Table 4 and Table 5 it is clear that $M + T$ gives better results than M and *Base*. But the choice of P is not straightforward. We suggest $P = 0.9$, but the other two evaluated values would be reasonable as well. Future work will be needed to prevent this kind of side-effects. One

Table 6. Statistical comparisons for line coverage percentage of the two configurations *Base* and $M + T$ with $P = 0.9$. We report the resulting $\hat{A}_{12}$ effect sizes, and the p-values of the statistical tests (in bold when lower or equal to 0.05).

SUT	Base	$M + T$	$\hat{A}_{12}$	p-value
catwatch	31.0%	31.4%	0.59	0.239
features-service	39.9%	42.5%	0.72	**0.003**
languagetool	18.6%	38.7%	0.99	**≤ 0.001**
proxyprint	27.1%	26.9%	0.50	0.984
rest-ncs	87.7%	87.9%	0.62	0.073
rest-news	51.2%	50.5%	0.37	0.081
rest-scs	60.0%	79.9%	1.00	**≤ 0.001**
restcountries	71.7%	74.3%	0.97	**≤ 0.001**
scout-api	38.5%	38.7%	0.54	0.594

Table 7. Statistical comparisons for fault findings of the two configurations *Base* and $M + T$ with $P = 0.9$. We report the resulting $\hat{A}_{12}$ effect sizes, and the p-values of the statistical tests (in bold when lower or equal to 0.05).

SUT	Base	$M + T$	$\hat{A}_{12}$	p-value
catwatch	20.9	21.4	0.62	0.105
features-service	33.9	33.8	0.43	0.223
languagetool	8.1	56.0	0.98	**≤ 0.001**
proxyprint	95.4	99.4	0.68	**0.004**
rest-ncs	5.1	5.0	0.43	**0.042**
rest-news	7.0	6.9	0.45	0.231
rest-scs	11.0	10.4	0.25	**≤ 0.001**
restcountries	23.0	23.0	0.50	1.000
scout-api	102.9	105.9	0.61	0.144

possible approach to mitigate side-effects would be to use an adaptive system to modify the value P based on fitness feedback during the search.

Table 6 and Table 7 show a more in depth comparison between *Base* and $M + T$ with $P = 0.9$. Regarding line coverage, our novel techniques improve performance significantly in four SUTs, with strong effect size, e.g., the maximum possible 1.0 for *rest-scs*, very high $0.97 - 0.99$ for *restcountries* and *languagetool*, and good 0.72 for *features-service*.

Regarding fault detection, we have statistically significant improvement for *languagetool* (+47.9 new found faults) and *proxyprint* (+4 new found faults). However, we get statistically worse results for *rest-ncs* and *rest-scs*, albeit the difference in results is minimal, i.e., -0.1 and -0.6 faults, respectively.

RQ3: *our novel techniques with configuration $M + T$ and $P = 0.9$ provides significant performance improvements, especially for large and complex SUTs like* ***languagetool****.*

8.5 Results on Industrial Software

Table 8 shows the results of the experiments on the industrial case study (which is a SpringBoot application). On such SUT, the improvements are very large and substantial. Target coverage more

Table 8. Average results (over 30 runs) on the industrial web service, where the *Base* configuration is compared against $M+T$. Besides reporting the average number of covered targets (which is a total of several different metrics), we report as well some of those metrics, like bytecode line coverage, percentage of endpoints for which a successful 2xx status code was returned, a faulty 5xx was returned, and the number of detected faults in such a web service. We also report the resulting $\hat{A}_{12}$ effect sizes, and the p-values of the statistical tests.

	Base	$M+T$	$\hat{A}_{12}$	p-value
Targets	142.0	512.3	1.00	≤ 0.001
Lines	0.8%	11.5%	1.00	≤ 0.001
HTTP 2xx	5.0%	26.0%	1.00	≤ 0.001
HTTP 5xx	95.0%	95.0%	0.50	1.000
Faults	38.0	44.6	1.00	≤ 0.001

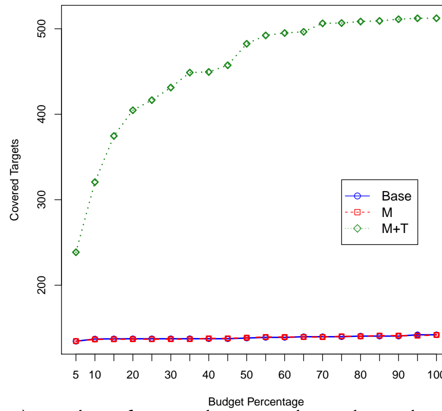

Fig. 11. Average (out of 30 runs) number of covered targets throughout the search, for the three configurations *Base*, M and $M+T$.

than tripled (however recall the presence of extra targets for the testability transformations), with line coverage going from not even 1% to 11.5%. It was possible to automatically detect 7 new faults in such web service. On each single metric but HTTP 5xx, we have very low p-values, with the strongest possible $\hat{A}_{12} = 1$ effect size.

This web service is composed of 20 endpoints, most of them using input validation based on several regular expressions and date parsing. Without our novel techniques, a tool like EvoMaster (or any black-box technique) would have extremely low probability of sampling strings that would pass those input validations.

Based on the results of Table 8, we can see that there is 1 out of 20 endpoints for which it is easy to obtain a 2xx code (so likely no input validation on it). However, on the other 19 endpoints there are several regular expression checks made by Spring before the SUT main code is even executed. This would explain the very low statement coverage 0.8% for *Base*. However, even with random inputs, it was possible to generate invalid inputs for which the SUT wrongly returns a 5xx status (server error) code instead of a 4xx one (which is used in HTTP to represent user errors). On the other hand, with our novel testability transformations, it was possible to create the right data to at least pass this first layer of input validation.

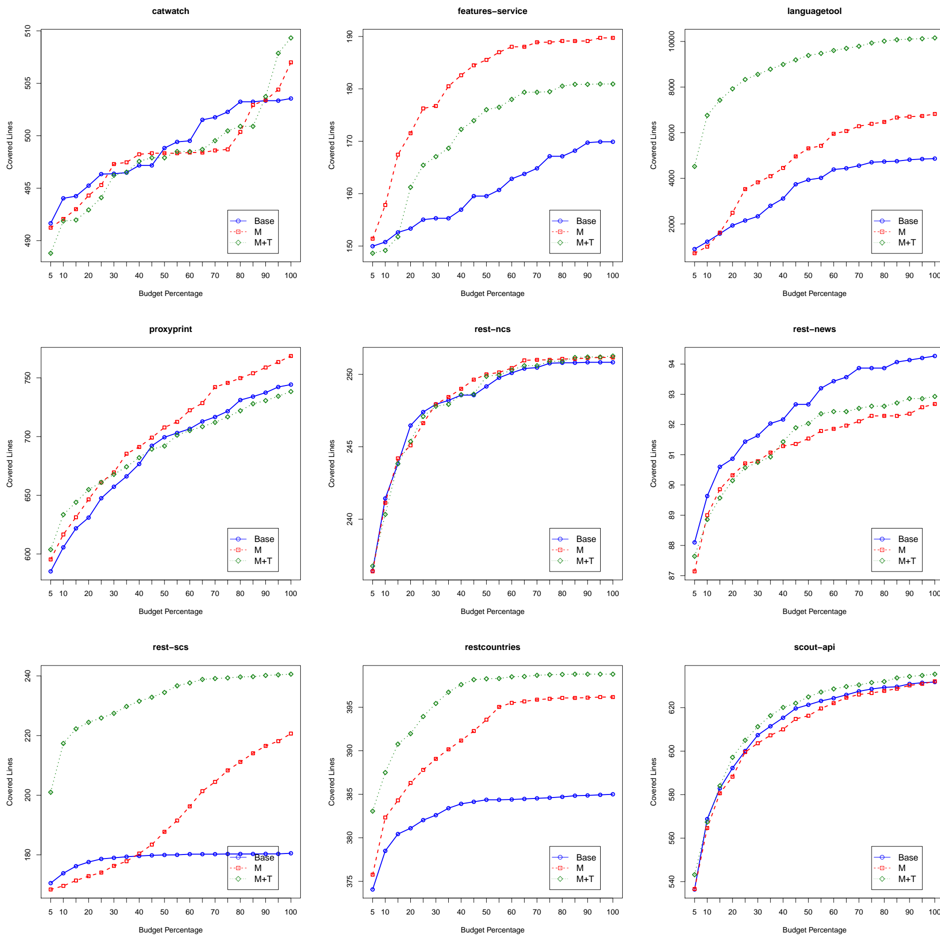

Fig. 12. Average (out of 30 runs) number of covered bytecode-level lines throughout the search, for the three configurations *Base*, M and $M + T$ with $P = 0.9$.

The performance of a search algorithm strongly depends on for how long it is left running. However, how engineers will use such test generation tools in practice is not yet clear. Would 4 hours be a too long time? Or would engineers be happy to keep the tool running overnight, and collect the results the day after (e.g., after 16 hours)? For fair comparisons, algorithms and their configurations should be compared with different search budgets, possibly representing realistic budgets that engineers would use in practice.

Figure 11 shows the performance of the three compared configurations at each 5% intervals of the search budget (i.e., every 50,000 HTTP calls). There is no performance difference between *Base* and M. This is expected, as M has no effect on third-party libraries (e.g., the regular expression checks in Spring). On the other hand, already at 5% of the budget, $M + T$ has performance nearly twice as high as the other configurations at the end of the search (i.e., at 100%). For completeness, Figure 12 shows how bytecode-level line coverage varies throughout the search for the open-source SUTs.

Even considering the $M + T$ configuration, a line coverage of 11.5% could be considered low. However, this is system testing of a complex industrial web service, where fitness evaluations are expensive, because they can involve several HTTP calls, with access to external databases like Postgres. At any rate, even with the current settings, our results are already of practical importance for our industrial partners, as 7 new faults were automatically found.

RQ4: *with our novel testability transformations, line coverage increased more than 10 times, and 7 new faults were automatically found in the industrial SUT.*

8.6 Results Discussion

Our empirical study shows the effectiveness of our novel techniques, in regards to code coverage and fault finding, when applied to white-box testing of RESTful APIs written for the JVM (e.g., using Java and Kotlin). But some of our techniques could be used in other testing contexts as well.

The method replacement techniques discussed in Section 4 could likely be used in any white-box testing context, like unit test generation. For example, tools such as EvoSuite do optimize for several different testing criteria [51]. Therefore, new testing targets could be considered for each transformed method. For example, a new testing target for the transformed method throwing an exception, and another testing target for the same method not throwing an exception. Due to the design of the MIO [10] algorithm, our heuristic values h are scaled in the range [0,1], where 1 means the target is covered. Mapping such values into a more traditional branch distance d (where the value 0 is used to represent a covered target) would be as trivial as applying a transformation function like $d = 1 - h$. However, how to best integrating these values h in other tools will depend on how the fitness functions are defined.

Adapting the use of input tracking (Section 5) would require more work, as it depends on the implementation of the search operators. For example, EvoMaster has a rich gene-system, where we already had specialized mutation operators for strings representing dates and regular expressions. This was done to handle such type of constraints in OpenAPI schemas [11] and SQL databases [15].

Although the techniques presented in Section 6 are specific for RESTful APIs (and so they are unlikely to be useful in other contexts), what presented in Section 7 could likely be applied in any other testing context where TCP connections are involved. These would include the testing of other kinds of web services (e.g., SOAP, GraphQL and gRPC), as well as web applications.

We have implemented our techniques for the JVM. However, they could be applied to other programming languages as well. Most languages do provide APIs for common operations, such as dealing with strings and date objects. As an example, .Net bytecode has many similarities with Java bytecode, as both VMs are stack-based. So, adapting these techniques for C# will likely be just a technical effort (but of course, we cannot be sure until such endeavor has been undertaken). On the other hand, dynamically and weakly typed languages such as JavaScript would be more complex to support.

The architectural design of EvoMaster (with a clear distinction between the *core* and *driver* processes) was made from the start to support different programming languages [8] (as those require just to write a new driver). We are in the process of supporting other programming languages, besides the ones that compile to JVM bytecode. Once such initial support is completed, we will adapt the techniques presented in this paper and investigate their performance for these other programming languages as well.

Although our experiments show an improvement in both code coverage and fault finding, this was not the case for all the SUTs in our case study. For some, there was no improvement. The techniques presented in this paper address one type of flag problem, specifically the case of flags in the methods/functions of existing standard APIs (like the JDK). If an SUT is negatively affected

by other types of flags as well, then not obtaining better results can be expected. For example, currently we do not handle flags in interprocedural boolean method calls [43]. To achieve better results, these other kinds of flags would need to be handled as well.

Besides search-based techniques, another successful approach for test generation is Dynamic Symbolic Execution (DSE) [19, 23, 40]. Applying DSE directly to testing of web services would be challenging (e.g., due to the dealing of TCP connections and database accesses), and that might be one reason why there is no current DSE approach for this problem domain. However, for constraints that can be handled with a constraint solver, DSE can be faster than SBST approaches. So, hybrid approaches that combine the strengths of SBST and DSE could lead to better results. This has been shown to be promising for unit test generation [30]. And so, this could hold as well for system testing of web services. But, several research and technical challenges would need to be addressed to successfully combine these two techniques for this specific problem domain.

9 THREATS TO VALIDITY

Threats to *internal* validity comes from the fact that that our experiments are based on an extension of the EvoMaster tool. Bugs in such extension could undermine the validity of our results. Although our extension was rigorously tested, we cannot guarantee it is fault-free. To address this problem, and to also enable the replicability of this study and third-party independent validation, our EvoMaster tool is published as open-source on *GitHub*.

Evolutionary algorithms are based on randomized algorithms. To analyze the effects of randomness on the results, we followed the guidelines in [13]. In particular, we used the Wilcoxon-Mann-Whitney U-test, and the Vargha-Delaney $\hat{A}_{12}$ effect size. All experiments were repeated 30 times, using different random seeds.

Threats to *external* validity comes to the fact that only 10 web services were used in the case study, due to the high cost of running experiments on system testing. We used open-source services to enable replicability of our study. We also used an industrial web service to study relevance and effectiveness in practice. However, we cannot claim that our results would generalize to other web services as well.

Our novel testability transformations were evaluated in the context of system testing of web services. As discussed in more details in Section 8.6, some of those transformations could be used also in other white-box testing contexts (e.g., unit testing). But, without empirical validation, we cannot affirm that our novel testability transformations would be as effective in those other contexts.

10 CONCLUSION

In this paper, we have presented a series of novel *testability transformations* to enhance search-based software testing. Experiments on nine open-source and one industrial REST web services show that our novel techniques improve performance significantly. For example, it was possible to automatically detect seven new faults in the industrial web service.

Future work will aim at providing more transformations for other common API methods. Furthermore, techniques and ideas from taint analysis, seeding (e.g., [52]) and Dynamic Symbolic Execution could be integrated in the search algorithms to improve performance even further. Although our novel techniques show significant improvements, there is still much more that need to be done to get high code coverage results and higher fault detection. System test generation is very complex, and the techniques presented in this paper provide an important contribution toward solving such complex problem.

To enable the replicability of our study, we implemented our techniques as an extension to our EvoMaster open-source tool. To learn more about EvoMaster, visit its website at:

http://www.evomaster.org

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 864972), and partially by UBACYT-2018 20020170200249BA, PICT-2015-2741.

REFERENCES

[1] [n.d.]. OpenAPI/Swagger. https://swagger.io/.

[2] [n.d.]. RestAssured. https://github.com/rest-assured/rest-assured.

[3] [n.d.]. Spring Framework. https://spring.io.

[4] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36, 6 (2010), 742--762.

[5] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 3--45.

[6] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability (STVR)* 16, 3 (2006), 175--203.

[7] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175--203. https://doi.org/10.1002/stvr.v16:3

[8] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[9] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering (EMSE)* (2018), 1--23.

[10] A. Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology (IST)* (2018).

[11] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.

[12] Andrea Arcuri. 2020. Automated Blackbox and Whitebox Testing of RESTful APIs With EvoMaster. *IEEE Software* (2020).

[13] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.

[14] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1--31.

[15] Andrea Arcuri and Juan P. Galeotti. 2020. Handling SQL Databases in Automated System Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1--31.

[16] Andrea Arcuri and Juan P. Galeotti. 2020. Testability Transformations For Existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 153--163.

[17] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.

[18] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1--39.

[20] A. Baresel, D. Binkley, M. Harman, , and B. Korel. 2004. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 108--118.

[21] A. Baresel and H. Sthamer. 2003. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO)*. 2442--2454.

[22] D. W. Binkley, M. Harman, and K. Lakhotia. 2011. FlagRemover: A testability transformation for transforming loop-assigned flags. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 12:1--12:33. https://doi.org/10.1145/2000791.2000796

[23] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82--90.

[24] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196--206.

[25] H. Converse, O. Olivo, and S. Khurshid. 2017. Non-Semantics-Preserving Transformations for Higher-Coverage Test Generation Using Symbolic Execution. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 241--252. https://doi.org/10.1109/ICST.2017.29

[26] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. 181--190.

[27] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures.* Ph.D. Dissertation. University of California, Irvine.

[28] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE).* 416--419.

[29] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276--291.

[30] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2014. Extending a search-based test generator with adaptive dynamic symbolic execution. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA).* ACM, 421--424.

[31] Matthew J. Gallagher and V Lakshmi Narasimhan. 1997. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering (TSE)* 23, 8 (1997), 473--484.

[32] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* Association for Computing Machinery, New York, NY, USA, 725–736. https://doi.org/10.1145/3368089.3409719

[33] D. Gong and X. Yao. 2012. Testability transformation based on equivalence of target statements. *Neural Computing and Applications* 21, 8 (2012), 1871--1882. https://doi.org/10.1007/s00521-011-0568-8

[34] Mark Harman. 2018. We need a testability transformation semantics. In *International Conference on Software Engineering and Formal Methods.* Springer, 3--17.

[35] M. Harman, A. Baresel, D. W. Binkley, R. M. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. 2008. Testability Transformation - Program Transformation to Improve Testability. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers.* 320--344. https://doi.org/10.1007/978-3-540-78917-8_11

[36] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering (TSE)* 30, 1 (2004), 3--16.

[37] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.

[38] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* 106--117.

[39] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE.

[40] J. C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385--394.

[41] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* (1990), 870--879.

[42] Y. Li and G. Fraser. 2011. Bytecode Testability Transformation. In *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings.* 237--251. https://doi.org/10.1007/978-3-642-23716-4_21

[43] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 440--451.

[44] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. A qualitative analysis of Android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 102--114.

[45] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing.*

[46] Phil McMinn. 2009. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009.* 1689--1696. https://doi.org/10.1145/1569901.1570127

[47] P. McMinn, D. Binkley, and M. Harman. 2009. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.* 18, 3 (2009), 11:1--11:27. https://doi.org/10.1145/1525880.1525884

[48] Sam Newman. 2015. *Building Microservices.* " O'Reilly Media, Inc.".

[49] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do android taint analysis tools keep their promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 331--341.

[50] RV Rajesh. 2016. *Spring Microservices.* Packt Publishing Ltd.

[51] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering.* Springer, 93--108.

[52] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability (STVR)* (2016).

[53] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317--331.

[54] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87--97.

[55] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[56] Stefan Wappler, Joachim Wegener, and André Baresel. 2009. Evolutionary testing of software with function-assigned flags. *Journal of Systems and Software* 82, 11 (2009), 1767--1779. https://doi.org/10.1016/j.jss.2009.06.037

[57] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1426--1434.