Advanced White-Box Heuristics for Search-Based Fuzzing of REST APIs

ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

MAN ZHANG, Beihang University, China

JUAN PABLO GALEOTTI, University of Buenos Aires and CONICET, Argentina

Due to its importance and widespread use in industry, automated testing of REST APIs has attracted major interest from the research community in the last few years. However, most of the work in the literature has been focused on black-box fuzzing. Although existing fuzzers have been used to automatically find many faults in existing APIs, there are still several open research challenges that hinder the achievement of better results (e.g., in terms of code coverage and fault finding). For example, under-specified schemas are a major issue for black-box fuzzers. Currently, EVOMASTER is the only existing tool that supports white-box fuzzing of REST APIs. In this paper, we provide a series of novel white-box heuristics, including for example how to deal with under-specified constrains in API schemas, as well as under-specified schemas in SQL databases. Our novel techniques are implemented as an extension to our open-source, search-based fuzzer EVOMASTER. An empirical study on 14 APIs from the EMB corpus, plus one industrial API, shows clear improvements of the results in some of these APIs.

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

Additional Key Words and Phrases: SBST, fuzzing, REST, Web API, OpenAPI, schema, SQL

1 INTRODUCTION

RESTful APIs are widely used in industry to build services that are available on the internet. At the time of writing, there are several thousands of REST APIs that are either free or commercial [1, 10, 11]. Besides providing functionality over the internet, this kind of APIs are also very common in industry for building enterprise systems with *microservice architectures* [80, 82].

Testing this kind of systems is challenging, due to communications over a network (typically HTTP over TCP), plus all issues in dealing with their environment (e.g., SQL databases and network communications with other APIs). Writing system tests for RESTful APIs can be a tedious, time-consuming, error-prone activity if done manually. In industry, there is a concrete need to automate this task [22].

Due to these challenges and practical importance in industry, in recent years there has been a significant amount of research on this topic [54]. Several *black-box* fuzzers have been implemented and evaluated, such as for example (in alphabetic order): bBoxrt [67], EVOMASTER [21] Restest [77],

Authors' addresses: Andrea Arcuri, andrea.arcuri@kristiania.no, Kristiania University College and Oslo Metropolitan University, Oslo, Norway; Man Zhang, manzhang@buaa.edu.cn, Beihang University, Beijing, China; Juan Pablo Galeotti, jgaleotti@dc.uba.ar, University of Buenos Aires and CONICET, Buenos Aires, Argentina.

RestCT [90], Restler [34], RestTestGen [87] and Schemathesis [61]. This kind of black-box fuzzers has been used to find several real faults in existing APIs. However, there are still many *open problems* in this testing domain [94], which significantly hinders the achievement of better results (e.g., code coverage and fault finding).

One major issue is that often the *schemas* of these APIs are *under-specified* [72, 94]. For example, if one query parameter in a REST endpoint is not specified in the schema, there is no (current) way for a black-box fuzzer to generate test cases using such a query parameter. Creating query parameters with random names would have an extremely low probability of matching any of these existing unspecified parameters. This can reduce the chances of achieving higher code coverage if such parameters have a major impact on the execution control flow of the API.

At the time of writing, our EvoMaster fuzzer is the only tool that supports *white-box* testing of RESTful APIs (and it supports as well RPC [95] and GraphQL [37] APIs). In our previous empirical studies [94], as well as in independent studies [65] from other researchers, it has been shown that white-box testing for REST APIs achieves better results than black-box fuzzing, in terms of achieved code coverage and fault detection. Furthermore, even in black-box mode [25], EvoMaster achieved among the best results (with Schemathesis [61] having similar results) in these tool comparisons [65, 94]. Considering that EvoMaster has been successfully used on large industrial, microservice enterprise systems with millions of lines of code [95], finding thousands of real faults that have been confirmed to be fixed by their developers throughout the years, it can be arguably considered among the state-of-the-art in fuzzing Web APIs (although in the specific context of black-box fuzzing REST APIs, Schemathesis could be considered better, or at least more widely used among practitioners in industry). Still, there are several research challenges to address, as large parts of the code-base of these APIs are not covered with existing techniques [94, 95].

To address these research challenges, in this paper we provide a series of novel *white-box* heuristics to improve the performance of fuzzing in this testing domain. In particular, we aim at addressing three different problems:

(1) *Flag* problem [36] in common library calls [28]. Calls to functions that return boolean values, or throw exceptions on invalid inputs, create "fitness plateaus" in the search space, making the evolutionary process of evolving high code coverage test cases hard.

(2) Under-specified schemas in OpenAPI definitions, in particular when dealing with missing HTTP query parameters and headers information. If some HTTP headers and query parameters are never used in the fuzzing process, then all the code related to their handling and associated functionalities would never be tested.

(3) Under-specified constraints in SQL database schemas, in particular when the tested API uses the Java Persistence API (JPA) to access the databases. Invalid data added to a SQL database could hamper the testing process for code coverage, as the API would simply crash as soon as such data is read. Having invalid data is good for robustness testing, but there is also the need to be able to test the "happy-path" scenarios.

Our novel white-box techniques can be used and applied to any white-box fuzzer. For the analyses in this paper, we have implemented these techniques as part of our EvoMaster fuzzer, in particular focusing on the JVM (e.g., for APIs written in either Java or Kotlin). However, they could be adapted to other programming languages as well, such as for the white-box fuzzing of JavaScript/TypeScript APIs running on NodeJS [98] and C# APIs running on .NET [53].

To validate our novel techniques, in our empirical study in this paper we used all the 14 open-source, JVM REST APIs currently part of the EMB corpus [33]. To better generalize our results, in our empirical study we also included 1 closed-source, industrial API from one of our industrial partners. Our empirical study shows that, in some of these APIs statistically significant improvements were

achieved (both in terms of code coverage and fault detection). On the one hand, this enables us to push forward the boundaries of scientific research in white-box fuzzing of Web APIs. On the other hand, there are still several challenges left, which will require further research to design more advanced white-box heuristics to solve these further issues.

The novel techniques presented in this paper are integrated in our search-based fuzzer called EvoMaster. EvoMaster is a mature research tool that has been under development for more than 7 years, at the time of writing, with many code contributors. Throughout the years, several novel techniques have been evaluated and integrated in EvoMaster, leading to several scientific publications (e.g., for handling of SQL databases [27], adaptive hyper-mutation [93] and testability transformations [28]). With hundreds of thousands of lines of code, at times this can make it difficult to properly distinguish between what already present in EvoMaster, and what novel contribution is presented in a new scientific article (like this one). To make it more clear when we are referring to our novel techniques compared to the previous existing versions of EvoMaster, we use the term EvoMaster^O when referring to the old version, and EvoMaster^N when referring to the new version integrated with the novel techniques presented in this article. To better understand the novel contributions of this article, Section 2 provides background information on EvoMaster^O. All the novel techniques defining EvoMaster^N are presented in Section 4.

This article provides several novel techniques (as we will show in Section 4). However, these novel techniques are independent of EvoMaster. They can be integrated and evaluated in any white-box fuzzer for REST APIs, although currently none exists besides EvoMaster [54]. Using a single research prototype to integrate and evaluate novel techniques throughout the years has major benefits. A mature, maintained tool can be used by practitioners in industry, in contrast to ''throw-away prototypes'' only meant to be used in the ''lab'' for doing a single empirical study in a scientific paper. In the research community, there is a major need of technology transfer from academic results to industrial practice [22, 48--50]. Besides providing novel techniques, we also provide the engineering and technology-transfer contribution of integrating those novel techniques in an existing tool downloaded thousands of times. Still, the challenge of white-box fuzzing web services is far from being solved, and more research needs to be done to design novel techniques to improve performance further. As an example, since conducting the empirical study in this article, based on its results we have been working on improving EvoMaster further, like with the dealing of MongoDB databases (although such work is not completed yet at the time of writing).

The paper is organized as follows. Section 2 provides background information to better understand the rest of paper. Related work is discussed in Section 3. The novel contributions of this article are presented in Section 4, divided into four subsections. In particular, our new handling of the flag problem is discussed in Section 4.1, followed by how we deal with under-specified schemas for OpenAPI in Section 4.2, and for SQL in Section 4.3. Section 4.4 provides our solution to the issue given by timed events. Our empirical study is presented in Section 5. Discussions on the obtained results follow in Section 6. Threats to validity of our study are analyzed in Section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 Terminology and Tools/Libraries

The work presented in this article deal with the white-box fuzzing in the context of REST APIs. In the context of Web and Enterprise applications and technologies, there are several common terms and tools/libraries widely used in industry and that are specific for this domain. To make this article easier to read, here we list (in alphabetic order) some of those terms/tools/libraries with some brief description, as those are referenced often throughout the rest of the article.

DTO: Data Transfer Object. *"... is an object that carries data between processes. The motivation for its use is that communication between processes is usually done resorting to remote interfaces (e.g., web services), where each call is an expensive operation... aggregates the data that would have been transferred by the several calls, but that is served by one call only"*.[1]

GraphQL: Graph Query Language [5]. Language used to define queries over graph-based data, typically implemented as web services over HTTP.

Hibernate: the most used library [7] the implements the JPA specification.

HTTP: Hypertext Transfer Protocol. Main application protocol for web services and web sites, run on top of TCP.

Jakarta: JEE was own by Oracle corporation. When in 2017-2018 Oracle donated JEE to the Eclipse Foundation, it was renamed into *"Jakarta EE"* due to trademark issues. Due to the same issues, the set of EE specifications also got their package renamed from `javax.*` to `jakarta.*`.

Javax: All the specifications in JEE have the same root package `javax.*`.

JEE: Java Platform, Enterprise Edition. Official set of specifications to build enterprise applications in Java. There are many specifications, including how to deal with relational databases (i.e., JPA) and HTTP servers (i.e., Servlet). For each specification of JEE (which at a high level could be seen as a set of interfaces with defined expected semantics), there can be different library/tool implementations.

JPA: Java Persistence API. The set of specification of JEE/Jakarta related to the handling of data in relational databases.

JSON: JavaScript Object Notation. An open standard data interchange format. It *"uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays (or other serializable values)"*.[2]

JVM: Java Virtual Machine. A virtual machines that can execute bytecode, compiled from languages such as Java and others (e.g., Scala and Kotlin).

ORM: Object–relational mapping. *"Object–relational mapping (ORM, O/RM, and O/R mapping tool) in computer science is a programming technique for converting data between a relational database and the heap of an object-oriented programming language"*.[3] Hibernate/JPA is a form of ORM.

REST: Representational State Transfer [43]. A set of architectural guidelines to design accessible resources through HTTP endpoints.

RPC: Remote Procedure Calls. Term used to represent the invocation of procedures on remote host machine. Different technologies exist to implement network and web services using RPC (e.g., gRPC [6]).

SQL: Structured Query Language [14]. Main language used to query and manipulate relational databases, such as for example Postgres and MySQL.

Spring: Java framework for building web and enterprise applications [13]. According to the yearly surveys of JetBrains (developer of the IDE IntelliJ), the Spring Framework is by far the most used in the Java ecosystem, covering more than 70% of the market share.[4]

Tomcat: An HTTP server implementing the JEE/Jakarta Servlet specifications. It is the default server in Spring.

---

[1]https://en.wikipedia.org/wiki/Data_transfer_object
[2]https://en.wikipedia.org/wiki/JSON
[3]https://en.wikipedia.org/wiki/Object-relational_mapping
[4]https://www.jetbrains.com/lp/devecosystem-2023/java/

2.2 REST APIs

Presently, a significant portion of web services are implemented following the REST (REpresentational State Transfer) architectural style [43]. Notable adopters of this approach among organizations and companies are Google[5], Amazon[6], Twitter[7], Reddit[8], LinkedIn[9], among others. Beyond their role in furnishing internet-based functionalities (e.g., see API portals such as *APIs.guru*[10] and *RapidAPI Hub*[11]), the REST architectural style also holds sway within enterprise backends, particularly when microservices architecture are used [80, 82].

The REST architectural style is not a protocol per se, but rather a collection of principles governing and structuring resources accessible over HTTP(S) networks. Resources are pinpointed via URLs and can be manipulated using HTTP semantics, with GET requests for data retrieval, POST for data creation, PUT/PATCH for data modification, and DELETE for deletion. Inputs can be transmitted through path components in URLs, query parameters, HTTP headers, and payload bodies. Various data formats can be employed for transmission, with JSON (JavaScript Object Notation) currently ranking among the most prevalent such data format.

To enhance the understandability and usability of REST APIs, a common strategy involves providing *schemas* outlining the available endpoints and supported input formats (including details like query parameter names and types). Multiple schema standards exist, with OpenAPI/Swagger [9] standing out as the current industry leader. This format harnesses either JSON or YAML (Yet Another Markup Language) to specify these schemas. Schemas can be authored either manually or auto-generated from API source code (exemplified by tools like SpringFox and SpringDoc for the widely adopted enterprise framework Spring [13]).

2.3 EvoMaster

In the context of REST APIs, a system-level test case is a sequence of HTTP calls to different endpoints. The expected result of each HTTP call can be asserted, checking that the value of the status code is within some given set of expected status codes values. As one might expect, writing a specific system-level test case that triggers a specific behaviour in the implementation of the API (such as creating a resource with a specific data format, or retrieving an existing resource against its expected value) is difficult and time consuming. Figure 1 presents a system-level test case (written using the RestAssured [12] library) that creates a new User resource using the endpoint /users. Subsequently, the test case retrieves a User resource by means of a GET HTTP call, and the retrieved data is compared against their expected values. The automation of creating such system-level test cases remains elusive [22].

The open-source tool EvoMaster [21, 29] aims at automatically generating system-level test cases for REST APIs. The existing version EvoMaster^O implements several evolutionary algorithms (such as MIO [23]) to evolve test cases towards maximizing code coverage and fault finding metrics. As API calls might depend on each other (e.g., in Figure 1 a typical GET template is exercised, where a unique PUT call is exercised followed by a GET call), EvoMaster can exploit dependencies among API resources [99, 100] to cover more API behaviour.

EvoMaster^O is divided in two main modules: (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,

---

[5]https://developers.google.com/drive/v2/reference/
[6]http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
[7]https://dev.twitter.com/rest/public
[8]https://www.reddit.com/dev/api/
[9]https://developer.linkedin.com/docs/rest-api
[10]https://apis.guru/
[11]https://rapidapi.com/hub

```
1 @Test
2 public void test() throws Exception {
3  // Prepare the request body for the POST request in JSON format
4  String requestBody = "{\"firstName\": \"Alice\", \"lastName\": \"Smith\"}";
5
6  // Create the new resource (POST request)
7  String newResourceId = given()
8      .contentType(ContentType.JSON)
9      .body(requestBody)
10   .when()
11     .post("/users")
12     .then()
13       .statusCode(201) // Assert the resource was created
14       .extract().path("id"); // Extract the newly created resource's ID
15
16   // Retrieve the newly created resource (GET request)
17   given()
18     .when()
19       .get("/users/" + newResourceId)
20     .then()
21       .statusCode(200) // Assert resource was found
22       .assertThat()
23         .body("firstName", equalTo("Alice"))  // Check the expected first name
24         .body("lastName", equalTo("Smith")); // Check the expected last name
25 }
```

Fig. 1. A system-level test case using the RestAssured library.

writting the 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. This instrumentation allows EvoMaster^O's driver to retrieve several different SBST heuristics like the branch distance. This heuristic is not only retrieved for predicates in the control flow of the SUT, but also for all SQL commands executed over a database (if any) [27].

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 (such as JavaScript [97, 98] and C# [53]). Additionally, this architecture also allows EvoMaster to support both white-box [24] and black-box [25] testing. If black-box testing is chosen, EvoMaster^O's *core* process can directly interact with the deployed API. Therefore, there is no need for engineers to write any driver as EvoMaster could be run on any type of REST API regardless of their programming language.

If white-box testing is chosen, EvoMaster^O currently supports APIs that run on the JVM [23, 24, 28, 94] (e.g., written in Java or Kotlin), JavaScript/TypeScript [97, 98] and C# [53]. For JVM-based APIs, it can output test suites in JUnit format, using the library RestAssured [12] for making the HTTP calls toward the SUT. The generated tests will use the configuration classes for handling the SUT. 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. 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.

In case of black-box testing, the generated test suites would still be output, e.g., in either Java or Kotlin using the RestAssured library, independently of the programming language in which the target APIs were written.

2.4 Method Replacement and Taint Analysis

In [28] we have presented a series of testability transformations [58] to improve the fitness function of EvoMaster^O, which built on top of some basic examples (e.g., for String.equals) from Evo-Suite [44]. In particular, in the case of JDK APIs where method calls end-up providing no gradient to the search (e.g., they return either boolean values or throw an exception on invalid inputs), for several of those APIs we automatically replace those method calls with our own custom versions, at class-loading time via bytecode instrumentation. These custom method versions are semantically equivalent: for the same inputs they give the same outputs. However, they compute heuristic values to determine how far the test data was from returning either a true or false output, and how far they were from not throwing an exception. Such heuristic values are then used in the fitness function to guide SBST search toward generating test data to get the desired output.

Consider this following simple example:

```
if(x.equals("A quite long string that it is unlikely to get at random")){
```

Here, sampling the string variable x at random would have extremely low probability of satisfying the constraint of that if statement. In our bytecode instrumentation, that code would be replaced with:

```
if(StringClassReplacement.equals(x,"A quite long string that it is unlikely to get
    at random",targetID)){
```

Here, String.equals gets replaced with our own StringClassReplacement.equals version. Internally, before returning the same result as String.equals, it computes heuristic values for the 2 possible outcomes: true or false (which are then registered as targets in the search using the label targetID). Such heuristics are based on the distance values defined in [18]. The search is then rewarded in the fitness function to apply modifications to x that lead to have at least 1 test case in which such call returns true, and 1 that returns false.

This approach gives gradient to the search to evolve a desired x value. However, due to the length of the string, it can take several generations in the evolutionary process to evolve the desired value. But what if x is part of the test data? Once it is detected at runtime that such value is compared with an equals method, it would be more efficient to simply use the desired value (i.e., *"A quite long string that it is unlikely to get at random"* in this case) directly instead of evolving it. The problem is that x could be modified during the execution of the SUT (e.g., it is the result of a substring operation, or a concatenation of different strings). Also, in the testing of RESTful APIs, such x could be the value of a URL query parameter, or a nested field in a JSON body payload in a POST request.

To address this issue, in [28] we presented a technique which is a form of taint analysis. When mutating strings in a test case (regardless of where they are, e.g., query parameters and fields in JSON objects), with a certain probability we replace such strings with tainted values in the form _EM_\d+_XYZ_. For example, _EM_0_XYZ_. In our method replacements, every time we detect that a string input is matching such regular expression, we can directly identify which part of the genotype it comes from. Then, the search can automatically replace such string values based on information from the replacement methods, making those constraints trivial to solve. That regular expression is written in that way to reduce the chances that a random string in the SUT would

match it. Also most modifications on the inputs (apart from modifying digits in the middle of the string) would be detected as well.

This approach does not work in all cases (e.g., when strings are modified), but it is very effective for RESTful APIs [28]. The reason is that those APIs often have many string inputs that are not modified by the SUT (i.e., they are just read).

In [28] we provided several method replacements, e.g., for `String.startsWith`, `Integer-.parseInt` and `Collection.contains`. For each one, we defined heuristic distances to provide gradient to the search. However, there are several more methods in the JDK APIs that could be handled this way.

2.5 Genotype Expansion

OpenAPI schemas could be underspecified. For example, some query parameters and headers could be missing from the schema, albeit being handled by the API. A white-box approach could detect some of these cases, as it can analyze the source code of the API.

In the popular Spring framework (which is the most used enterprise framework for the Java language), parameters and headers can be automatically injected via annotations in the REST controllers (e.g., using `@RequestParam` and `@RequestHeader`). In those cases, automated tools that create OpenAPI schemas by analyzing annotations can detect those inputs (e.g., SpringFox and SpringDoc). However, a less common case is to pass as input to those REST controllers an object representing the whole HTTP call, such as `WebRequest`. In those cases, automated schema generators would have no information on the actual expected structure of the incoming HTTP requests. The generated schema definitions would hence have no useful info.

As this problem actually happens in some of the APIs in EMB [3, 33], in [28] we presented some techniques to handle some basic cases of this issue. In particular, we provided method replacements for `getParameter()`, `getParameterValues()`, `getHeader()` and `getHeaders()`. Everytime any of these methods is called, and the presence of a parameter/header not in the schema is detected, our instrumentation will inform the search engine about such value. Then, EVOMASTER will expand the genotype of the evolving individuals, creating test cases that can set and use those newly discovered parameters and headers.

Similarly to parameters and headers, there can be issues when the body payload types are unspecified. For these cases, we provided a method replacement for `getInputStream()` on the object representing the incoming HTTP requests. If the execution of such call is then detected, we then trace the use of the library Gson for marshalling JSON payloads, in particular the calls to `fromJson()`. This way, we check which classes are used as DTO to map the incoming body payloads. This information can then be fed back to the search engine, which will now be able to evolve input objects matching those DTO structures. As there are few technical details at play here, we refer the interested reader to [28] for the full details on how this is achieved in EVOMASTER^O.

3 RELATED WORK

3.1 Fuzzing REST APIs

Fuzzing is one of the most effective approaches to detect software faults, and to achieve higher code coverage [51, 71, 101]. Fuzzers operate using either a white-box or a black-box approach. In a white-box fuzzer, internal details of the system under test, such as its source code, binaries, bytecode, or SQL communication [27], will be accessed. This information can be used to design heuristics to improve the search to produce better test cases with better code coverage and fault detection capabilities. White-box fuzzers have been proven to be highly effective in numerous instances [25, 52, 65, 73, 94].

Fuzzing REST APIs has gained a major amount of attention among researchers in recent years (see for the example the survey [54]). There are multiple fuzzing tools available for RESTful APIs [65, 94]. Some of the aforementioned *black-box* fuzzers for REST APIs include (in alphabetic order): bBOXRT [67], EvoMaster [21], Dredd [8], Fuzz-lightyear [4], Morest [70], Quickrest [63], ResTest [77], RestCT [90], Restler [34], RestTestGen [87] Schemathesis [61], and Tcases [15].

For the testing of REST APIs, different studies having been carried out, including for example robustness testing [42], security testing [41], handling of inter-parameter dependencies [74, 75], generation fo realistic test inputs [85], carving API tests from GUI interactions [91], and defining test coverage criteria [76]. For a more in-depth overview of such work, see our survey on this topic [54].

When dealing with under-specified API schemas, Kim *et al.* [64] provided a Natural Language Processing (NLP) approach for OpenAPI. Although a schema can be underspecified, it might still have natural language descriptions. For example, although a constraint might be not formally defined, the authors of the schema might still have added an informal description of such a constraint in the text comments. NLP techniques can then be used to analyze such text and then to extend the schema with extra inferred constraints (if any is present). One advantage of such a technique is that it is independent of the fuzzer, as it works directly to extend the given OpenAPI schema.

To the best of our knowledge, EvoMaster [21] is the only REST API fuzzer that can support both white-box and black-box testing of REST APIs. With the recent studies conducted for comparing the state-of-the-art REST API fuzzers [65, 94], EvoMaster in white-box mode achieved the best performance in code coverage and fault detection.

REST is not the only approach to define web services. Others are for example GraphQL [5] and different types of RPC frameworks (e.g., gRPC [6]). However, in contrast to REST [54], not much work has been done in the research literature on testing these other kinds of web services (e.g., [37, 62, 86, 92, 95]).

3.2 Search-Based Software Testing

Search-based software testing (SBST) [16, 60] has been shown to be an effective technique to automatically generate test cases. In SBST, the problem of generating adequate test suites and fault-releaving test cases can be reformulated as an optimization problem. Examples include unit testing of Java software with open-source tools like EvoSuite [44], and testing of mobile applications with the Sapienz tool at Facebook [17].

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* [66]. Given a boolean predicate in the code of the system under test (SUT) (e.g., x==y+10), the branch distance provides a heuristic value (e.g., minimize the value of |x-(y+10)|) to guide the search towards satisfying the boolean predicate.

Unfortunately, a common issue is the so-called *flag problem* [36], where the branch distance is not able to provide any gradient. For example, consider a scenario where our target program is dealing with string operations returning booleans [19], like comparisons of two strings for equality. By default, a boolean predicate like x.equals("Hello"), would be just a flag, returning true or false. In such a scenario, the search algorithm has no guidance whether x is getting closer to become "Hello". Inputs such as "ello" and "foo" will be considered indistinguishable. Flags in the code might depend on string operations [19], loop assignments [35, 38], nested predicates [79], calls to boolean functions [68, 69, 89] and non-integer comparisons [68].

An approach to address this issue is to transform the code of the SUT to improve the fitness function, using so-called *Testability Transformations* [59]. A testability transformation modifies the original SUT to produce a new version of the same SUT, but more amenable to the test generation

process. This change might not necessarily preserve its original semantics, but it must *"preserve test sets that are adequate with respect to some chosen test adequacy criterion"* [56]. As an example, if a test generation targets a single goal (e.g., a specific branch in the SUT), a non-semantics preserving transformation might slice (i.e., remove) some lines/branches to alleviate a test generator. In contrast, semantics preserving testability transformation might not (by definition) affect the chosen test adequacy criteria. As an example, a semantic preserving testability transformation could replace the original boolean predicate x.equals("Hello") with a computation of a string distance, like the *edit distance* (number of insertions, deletions or changes to transform a given string into another) [19]. If such a transformation was applied, comparisons to "ello" and "foo" in x.equals("Hello") will no longer be considered indistinguishable by the heuristic.

Different transformations have already been proposed [56, 57], mainly to deal with (but not restricted to) flag conditions [19, 35, 36, 38, 55, 68, 68, 69, 79, 89]. Testability transformations have also been proposed to generate pseudo-oracles [78] (which can be helpful to detect numerical inaccuracies and race conditions). Furthermore, besides search-based test generation, slicing-based testability transformations can also be useful to improve Dynamic Symbolic Execution test generation [40].

EVOMASTER performs whole test suite generation (as it is done by EvoSuite [46]) at the system level. In other words, it simultaneously targets all goals within the SUT. In this scenario, many of the aforementioned non-semantics preserving transformations would not be applicable as they would lead to generating test cases that do behave accordingly in the original SUT. For this reason, a main difference in [28] (i.e., EVOMASTER^O) and in this work (i.e., EVOMASTER^N), is that all of the proposed testability transformations must preserve the semantics of the SUT.

4 NOVEL TECHNIQUES

In this section, we present the set of novel techniques which constitute the core of the scientific contributions of this article, defining EVOMASTER^N. Those are related to new method replacements (Section 4.1), REST APIs schemas (Section 4.2), SQL schemas (Section 4.3) and timed events (Section 4.4).

Note that, in a minority of cases, some of these techniques are simple extensions of what original presented in [28] for EVOMASTER^O. Those cases will be explicitly labeled as such. In particular they are related to the handling of data structures (Section 4.1.1) and HTTP/JSON (Section 4.1.4). For the sake of completeness, we believe it is still important to discuss those cases as well. This is particularly the case for researchers that want to re-use, adapt or build upon what presented in this article. Furthermore, regardless of their *novelty* or levels of *sophistication*, there is scientific value in evaluating their impact on performance with sound empirical studies.

4.1 New Method Replacements

In this paper, one of the scientific contributions is to provide method replacements with SBST heuristics for several more APIs in the JDK and common libraries. This is a direct extension of what first presented in [28] (recall Section 2.4), and currently available in EVOMASTER^O. The details of those new method replacements for EVOMASTER^N will be discussed in the next sections.

When a new method replacement is designed, up to three different tasks need to be considered (depending on the method):
- define a new *branch distance* [66] function d, to guide the search, where d is minimized. The value $d = 0$ would mean that the target is covered. This is done for functions that return booleans, and for functions that throw exception on invalid inputs. To handle some further special cases (e.g., Map.get) we also consider functions that might return null, i.e., for those

cases we create two new different testing targets: returning null and returning a non-null value.

- if the function deals with string inputs, possibly apply taint analysis (recall Section 2.4).
- in case of taint analysis, there is the possibility to define new gene types to evolve strings with specific constraints (e.g., for URL, URI and UUID, as it will be discussed in Section 4.1.3)

In EvoMaster^O, each testing target (e.g., lines and branches) we try to optimize for has a heuristic value $h \in [0, 1]$. The value $h = 1$ means that the target is covered (equivalent to $d = 0$). Note that, internally, EvoMaster^O uses h instead of d to better handle the simultaneous optimization of many targets (e.g., with the MIO algorithm [23]). Given a branch distance value d, h is computed as $h = b + (1 - b)\frac{1}{1+d}$, where b is a small constant, e.g., $b = 0.1$ (see [28] for full details and rationale for these choices).

Note that, for reasons of space, we cannot go here into all the low level technical details of each of these new method replacements designed for EvoMaster^N. Those are thousands of lines of code, to deal with many edge cases (e.g., how to deal with instances of IdentityHashMap which does not use equals for comparisons, or how to detect at runtime maps that have $\Omega(n)$ instead of $O(logn)$ complexity for containsKey, which would result in computational bottlenecks when calculating the branch distances). What we present here is high level descriptions. For full details, we refer the reader to our open-source implementation on GitHub [2], specifically to version 1.6.1, which is stored on Zenodo [30] for long term storage. In particular, most of the implementation is under the package org.evomaster.client.java.instrumentation.coverage.methodreplacement.

4.1.1 Data Structures.

Data structures like Lists, Sets and Maps are widely used. Many of their functions return booleans, e.g., when checking if the collection contains a specific input value, which creates fitness plateaus in the search. In [28] for EvoMaster^O, we already handled most of those boolean methods, by providing method replacements for the class java.util.Collection. One missing case was for Collection.containsAll(). Given an input Y for a collection X, such method would return true if every single element in Y is present inside X.

To deal with this, in EvoMaster^N we rely on the already existing distance d_c defined in [28] for Collection.contains(), which was defined as $d_c(e, X) = min_{x \in X}(d(x, e))$, for input element e on collection X. In other words, compute a branch distance on each element in the collection from the input, based on their type (e.g., string and numbers), and take the minimum (as the input just needs to match one single element, and we can prioritize the closest one). Then, such distance d_c can be converted into a heuristics value h_c, as previously discussed. For the case of Collection.containsAll(), in EvoMaster^N we then use $h_a(Y, X) = \frac{\sum_{y \in Y} h_c(y, X)}{|Y| + log|Y|}$. Here, we compute h_c for each element in Y and sum them, as all of them must match one element in X. As h_c still must be in $[0, 1]$, we scale the result of the sum by the cardinality of the collection Y. However, during the search it might well be that the size of Y changes (if Y is an input from a test case). The less elements in Y, the easier it is to satisfy such constraint. So, an increase in length should be penalized, and that is why we add a further $log|Y|$ to the denominator.

Other missing methods are related to the removal of elements in a collection. For example, Collection.remove() returns a boolean based on whether the element was in the collection. We apply the same distance computations as Collection.contains(). Likewise, Collection.removeAll() is treated similarly to Collection.containsAll(). These are now handled in EvoMaster^N.

Where lists and sets in the JDK do extend from java.util.Collection, maps do not. In other words, the interface java.util.Map is not a subtype of java.util.Collection, although it has several methods with same name and semantics (e.g., size() and isEmpty()). This means that

none of the transformations presented in [28] works on map data structures out of the box. Handling this in EvoMaster^N was only technical effort, as it was just a matter of implementing equivalent replacement methods for those map methods having the same semantics, such as `isEmpty()`, `get()`, `getOrDefault()`, `containsValue()`, `remove()` and `replace()`.

Although technically they are not collections, enumerations (i.e., `Enum` in Java) share some characteristics. For example, `Enum.valueOf(s)` would return an enum instance based on the input string *s*. If *s* is not matching any of the values in the enumeration, it throws an exception. Here, this is the equivalent case of checking `contains` on a list/set of strings. So, in EvoMaster^N we apply the same type of branch distance calculation (together with taint analysis, as the input is a string).

4.1.2 Miscellaneous Functions.

Comparison of two objects for equality is a typical case for which heuristic distances can be defined, depending on their type, e.g., for numerical values [66] and strings [20]. In Java, each object extends from `java.lang.Object`, which defines the method `equals`. Different classes like `String` and numerics such as `Integer` and `Double` have their own overridden implementations. We already provided method replacements for all those cases in [28] for EvoMaster^O. However, an important missing case was when the references of the compared objects are abstracted to the root-type `Object`. Consider the example of `x.equals(y)`, where both `x` and `y` are of type `Object`. At compilation time, as well as at instrumentation time (when classes are first loaded into the JVM), there would be no information on their actual types. They could be two numbers, or a TCP socket compared with a Hash Map, for what we might know. In those cases, we would not be able to apply any of the transformations from [28]. However, at runtime the types of the compared objects would be known. So, a relatively easy solution here is to provide a method replacement for `Object.equals()`. Then, when such method is executed during the evaluation of a test case, we can check the actual, most specific types for `x` and `y` (e.g., using operators such as `instanceof`). If those inputs happen to be of types for which we have defined any heuristic distance (e.g., strings and numbers), then we calculate such distance (as well as applying taint analysis in case the type is string).

Between what already presented in [28] for EvoMaster^O, and what presented newly in this paper for EvoMaster^N, in EvoMaster we have more than 100 methods for which we provide replacements for computing different kinds of heuristics. However, those are applied only when such method usage can be identified in the instrumented bytecode, e.g., like a call to `x.equals(y)`. This unfortunately leaves out the cases when methods are called by *reflection*, like `m.invoke(x,y)`, where `m` is a reference to the `java.lang.reflect.Method` instance for the method `Object.equals()`. In EvoMaster^N, we apply a method replacement for `Method.invoke()`, where, at runtime, we check if the used method referenced by `m` is one for which we have any replacement. If so, inside the replacement for `m.invoke()` we rather call the replacement for the method accessed by reflection. Unfortunately, though, a major issue here is that reflection is used *massively* in Java, especially in popular frameworks such as Spring and Hibernate. A naive replacement for `Method.invoke()` would be (and was) a major performance bottleneck. This was one of the few cases in which several code optimizations were needed to do not drastically reduce performance (e.g., by using different levels of caches to memoize parts of the computation).

Another method that can have drastic effects on performance is `sleep` in the class `java.lang.Thread`. When a test case is evaluated, if a `sleep` is executed on the same thread of the test case, then the duration of the test case would directly increase by the amount of time given as input to `sleep`. If the test case involves executing a network call (e.g., HTTP over TCP when fuzzing web services), such connection could timeout if the `sleep` is too long. This is a major

issue when the input of the `sleep` depends on some variables which are influenced by the data in the test case (and not just a constant in the SUT). Having a fuzzing process in which each single test case timeouts would drastically reduce the number of fitness evaluations during the search, reducing the amount of the search landscape that can be explored.

Furthermore, even if a `sleep` is executed on a background thread, when the evaluation of a test case is completed, we would want to avoid having dangling sleeping threads that might suddenly wake up when evaluating a new test case later on. Test case executions should be independent from each other. As such, when the execution of a test case is completed, before evaluating a new test, the *state* of the SUT should be reset. A typical example is to clean the modifications applied on the connected databases (if any). Dangling threads is yet another case of SUT state that needs to be handled.

In EVOMASTER^N, we provide a new method replacement to handle all the cases of `sleep`, with two main objectives: (1) collect information on all threads that sleep, so they can be ''interrupted'' automatically once a test case execution is completed (also note that EVOMASTER is equipped with a ''kill-switch'' [31] to block the thread execution as soon as the awoken thread executes any instrumented code); (2) put a limit to the amount of time the thread is allowed to sleep (e.g., 1 second).

To avoid possibly messing up with the threads of the HTTP server of the SUT (e.g., Tomcat), sleeps in those threads are not modified. Also, technically speaking, point (2) could change the behavior of the SUT. Doing this is arguably controversial, but the benefits strongly outweigh the negative sides. Leaving the sleeps as they are could just make the fuzzing unfeasible in some cases (due to drastic performance drops), and TCP timeouts together with ''dirty'' state between different test executions would be worse than possibly breaking some ''soft-constraint'' time-related behavior. In other words, in the large majority of the cases when fuzzing REST APIs, we are not dealing with test executions that should last for minutes, where the SUT is still executing business code after it has responded to an incoming HTTP request. In those special, rare cases, more sophisticated techniques would need to be designed.

4.1.3 String Specializations.

Strings are widely used as input in web services. It is one of the most common types, if not the most common one (although being able to claim that would require an analysis of existing APIs on internet). However, strings are problematic for testing purposes. Each string defines a hugely massive search space of possible combinations of characters. Given k possible characters, and length up to n characters, there are $\sum_{i=0}^{n} k^i$ possible strings. During the search, only an extremely tiny subsets of all those possible strings can be evaluated. Therefore, there is the need of smart strategies when dealing with string inputs.

Often, strings should match some specific constraints, like representing a valid email or a valid IP address. Those constraints could be for example represented with regular expressions, which define a subset of valid strings for the given specialized string type. When generating test inputs, it would make sense to generate valid strings based on those constraints. Although, of course, for robustness testing it still makes sense to send some invalid strings any now and then. Regular expressions are widely used, and so this common case was already handled in [28] for EVOMASTER^O (e.g., for classes such as `java.util.regex.Pattern`). However, there are several JDK classes that represent specialized strings, and that can take strings as input when initializing new instances of such objects. Common examples are URIs (constructor for `java.net.URI`, and methods such as `URI.create()` and `URI.resolve()`) and URLs (e.g., constructor for `java.net.URL`). When dealing with databases,
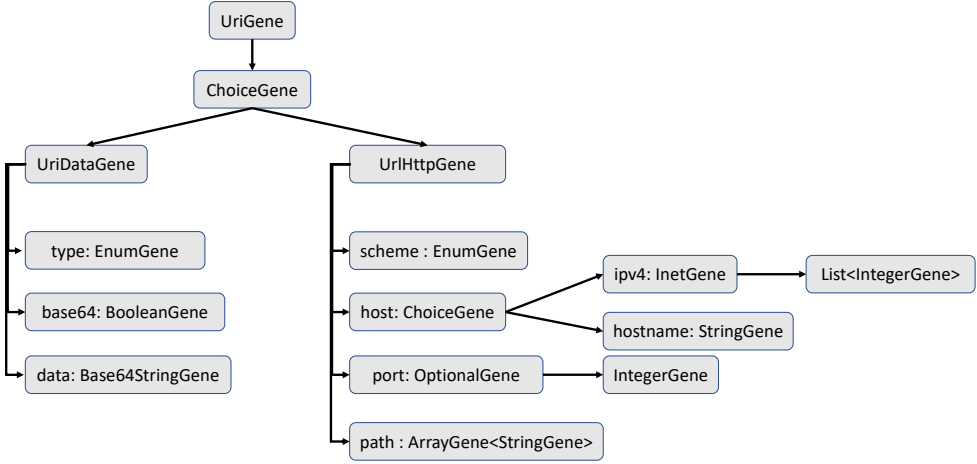
Fig. 2.   Simplified tree-representation for new `UriGene` introduced to handle URI strings.

Universal Unique Identifiers (UUIDs) are also common (e.g., the method `fromString()` in the class `java.util.UUID`).

In EvoMaster^N we provide novel method replacements for all these methods, to enable taint analysis in them. When in our instrumentation we detect that a tainted input is given as input to any of these replaced methods, the search is informed about it. To enable generating valid strings that would not crash those methods (i.e., throw an exception due to invalid inputs), we extended the search engine in EvoMaster^N to provide new specialized *genes* for the evolving test cases. In EvoMaster, each gene not only defines the structure of the data (i.e., the *genotype*) and how it will be represented when evaluating the fitness function (i.e., the *phenotype*), but also it defines the mutation operator on such type (e.g., mutating an integer gene is different from mutating a string gene).

The simplest case is UUIDGene, which is a 128-bit label. Internally, in its genotype it includes two LongGenes, which are then mutated like any LongGene in EvoMaster. The right phenotype is then reconstructed from using the constructor of UUID that takes as input two long inputs.

The cases of URI and URL are similar, but unfortunately much more complex, as there are many rules on how to define valid URI/URL objects (e.g., see RFC 1738[12]). Figure 2 shows a *simplified* example of the tree-representation of the new `UriGene` we introduced for EvoMaster. Each valid URL is a valid URI, so a URI can be represented with a URL, or a URN. The syntax of a URI depends on its scheme, e.g., `data`, `http`, `https`, `file`, `ftp`, and `gopher`. So, in EvoMaster^N first there is the need to make a ''choice'' of which scheme to use, which is done with a ChoiceGene. In a ChoiceGene, only one of its children contributes to the phenotype of the individual, where the mutation operator selects which child to use. For example, the `UriDataGene` used to represent the `data` protocol would have genes to define the type (e.g., an enumeration with entries like `text/plain`), whether it should be in base 64 format (using a BooleanGene), and the data itself. The phenotype for such string specialization could create values such as `data:text/plain;base64,Zm9v`. On the other

----
[12]https://www.rfc-editor.org/rfc/rfc1738

```
1 @Validated
2 @RestController
3 @RequestMapping(path = "/api/valid")
4 public class ValidRest {
5
6     @RequestMapping(method = RequestMethod.POST)
7     public String check(@RequestBody  @Valid ValidDto dto){
8         if(dto == null){
9             return "WRONG";
10        }
11        return "OK";
12    }
13 }
```

Fig. 3.   Example of bean definition for a Spring controller dealing with a REST endpoint.

hand, to represent an HTTP/S URL, we would need to define the scheme (e.g., an EnumGene with the values http and https), the host (which could either be string hostname or a numerical IP address), as well as an optional port, and path component. Note: all these genes also have valid constraints (which are kept satisfied by the sampler and mutation operators), like for example the port gene is constrained in 0 and 65535, and each of the 4 integer genes inside the InetGene are constrained in 0 and 255.

4.1.4   HTTP and JSON.

In Section 2.5 we have explained how we dealt with *genotype expansion* in [28] for handling underspecified schemas. In particular, how to deal with missing query and header parameters, as well as non-specified types for body payloads.

What is implemented in EvoMaster^N compared to EvoMaster^O is a simple, straightforward extension of [28]. Where in [28] we only handled the methods in the Spring interface WebRequest, here we also consider the JEE class HttpServletRequest, which has the same methods with same semantics, i.e., getParameter(), getParameterValues(), getHeader() and getHeaders(). Furthermore, besides supporting Gson to analyze how strings are marshalled into JSON objects, now we do the same as well for Jackson[13] (in particular, all the different variants of readValue() and convertValue()). With this, in EvoMaster^N we now support all the major libraries for parsing JSON objects on the JVM.

4.1.5   Javax/Jakarta Bean Validation.

Web APIs on the JVM are often implemented with enterprise frameworks, like for example Spring and JEE. These frameworks make a large use of *Bean* classes. Those are classes that get *enhanced* at runtime, based on annotations applied on them. When the application starts, these frameworks instantiate *proxy* classes, in which each method invocation is intercepted and possibly modified based on the semantics of the applied annotations (e.g., to automatically handle SQL transactions).

Figure 3 shows such an example. Here, the Spring framework treats the class ValidRest as a bean, as it is marked with the annotation @RestController. The method check() will handle all POST

---

[13]Another Java library for marshalling/unmarshalling JSON payloads

requests for the endpoint /api/valid (this is specified with the annotation @RequestMapping). The body payload of the incoming HTTP POST request is marshalled into an instance called dto of the class ValidDto (based on the annotation @RequestBody), which is then given as input to the method check(). Because the class is marked with @Validated and then input dto is marked with @Valid, then all the constraints in such object are checked. If any is violated, then the Spring framework will automatically return a failing HTTP response with status 400, i.e., user error, without executing check().

To define constraints, JEE/Jakarta provides a rich system of annotations (used by Spring as well), such as @Min, @Max, @Positive, @PositiveOrZero, @Negative, @NegativeOrZero, @Size, @NotEmpty, @NotBlank, @Null, @NotNull, @AssertTrue, @AssertFalse, @Pattern, and many more. These annotations can then be applied on the fields of Java Beans, and validated each time the methods of the beans are called. These annotations are not used only for REST APIs, but for all kinds of beans, including for example the beans used to represent data in databases (i.e., @Entity classes used in JPA, which we will discuss in more details in Section 4.3).

Depending on the constraints in the annotated objects, this can be a major issue for automated testing purposes. A HTTP call toward the endpoint would likely fail with a 400 status without any code of the business logic being executed. If no code of the business logic is executed, then there would be no info on its code coverage and other heuristics such as the branch distance on its predicates (e.g., if statements). The problem is that the check for constraints would be done in a call to validate() in the class javax.validation.Validator, deep inside the internals of the Spring framework. Without any adhoc technique, even white-box fuzzers such as EvoMaster^O would have no way to generate valid data for such endpoints (unless all the constraints are specified as well in the OpenAPI schema). Note that the artificial example in Figure 3 is coming from one of our own end-to-end (i.e., system level) tests [31] for EvoMaster^N, where the class ValidDto has 20 fields with constraint annotations. Without any fitness gradient in the search, it would be very unlikely to sample a random dto instance that satisfies all these constraints. However, with our novel technique presented in this paper for EvoMaster^N, it becomes relatively simple, or at least simple enough that it can be solved consistently with a small search budget, and so it can be used as a end-to-end test for EvoMaster itself [31].

To handle these cases, in EvoMaster^N we provide a novel method replacement for Validator.validate(). Each time it is called, it computes two distances: how far the object is from being evaluated as valid, and how far it is from being evaluated as invalid. However, although there is only few places (typically just one) in which Validator.validate() is called (and where the transformation is applied), there could be hundreds of places in the business logic of the SUT that would trigger that validation check (e.g., all endpoint entry points, all intra-bean calls, and all write operations to the database when using JPA). For the search, we need to be able to distinguish most, if not all, of these cases. To achieve this, in EvoMaster^N we create 2 testing targets (i.e., for the true and false outcome) for each combination of endpoint name with HTTP verb and name of validated object. In this example, those 2 new testing targets would be identified with something like VALIDATE_POST:/api/valid_ValidDto_true and VALIDATE_POST:/api/valid_ValidDto_false.

Regarding the heuristic distance on the validated objects, it can be considered as a conjunction of clauses, where each constrained field is a clause (e.g., $A \wedge B \wedge \ldots$). *All* clauses must be satisfied for the object to be valid. Note that fields could be objects as well, and so this needs to be applied recursively. For the object to be considered invalid, we need only a single constraint to be evaluated as false, and so it can be represented with a disjunction of negated clauses (e.g., $\neg A \vee \neg B \vee \ldots$). Those cases can be handled with standard equations in the SBST literature [47]:

$$d(A \wedge B) = d(A) + d(B)$$

```
1 @RequestMapping(value="paypal/ipn/consumer/{consumerID}",
2                 method=RequestMethod.POST)
3 protected void consumerLoadUpConfirmation(
4       @PathVariable(value = "consumerID") long cid,
5       HttpServletRequest request,
6       HttpServletResponse response
7       ) throws ServletException, IOException {
8   Map<String,String> configurationMap = Configuration.getConfig();
9   IPNMessage ipnlistener = new IPNMessage(request, configurationMap);
10  boolean isIpnVerified = ipnlistener.validate();
11  String transactionType = ipnlistener.getTransactionType();
12  Map<String,String> map = ipnlistener.getIpnMap();
13
14  String payerEmail = map.get("payer_email");
15  Double quantity = Double.valueOf(map.get("mc_gross"));
```

Fig. 4. Snippet of function handler for the endpoint /paypal/ipn/consumer/{consumerID} in *proxyprint*.

$$d(A \lor B) = min(d(A), d(B))$$

As the constraints are based on annotations, and not on executed methods that could have side-effects, here there is no issue when dealing with short-circuit evaluations of boolean predicates (which would require more advanced equations, as done for example in [98]).

Most of these constraints in JEE/Jakarta deal with numbers and strings, which just require a rather straightforward mapping of existing branch distance calculations [18, 66] and use of taint-analysis (especially important for @Pattern). We have handled all of those cases. However, there are two groups of constraints that EvoMaster^N does not handle yet. First, we do not deal with any time-related constraints, like for example @Future, @FutureOrPresent, @Past, and @PastOrPresent. Time related properties are hard to handle in automated testing, and can be a major source of *flakiness*. For example, a time constraint valid during the search might not be longer valid when the generated test cases are executed later on. Without a proper, deterministic handling of time behaviours, trying to handle those constraints would not be particularly useful. Second, we do not handle *custom* constraints. JEE/Jakarta enables users to write their own annotations with customized code to evaluate the validity of the constraints. As such code could do anything, we cannot prepare pre-defined heuristic distances for those cases. Future work would be needed to define on-the-fly heuristics based on the analysis of the source code of these custom methods.

4.2 Underspecified REST API Schemas

In [28] we presented some initial work for EvoMaster^O to handle under-specified OpenAPI schemas (recall Section 2.5), which has been now extended here in EvoMaster^N by dealing with more classes and libraries (Section 4.1.4). However, still such work would only be able to deal with Spring and JEE/Jakarta, and, even for those, not all cases could be handled. Let us consider the snippet in Figure 4, coming from the SUT *proxyprint* used in our empirical analysis (Section 5). This is one of cases of *open problems* that was discussed in details in [94]:

> *"... shows an example in which the line defining the variable* quantity *throws an exception, due to* Double.valueOf *being called on a null input. Here, an HTTP object* request *is passed as input to the constructor of* IPNMessage*, which is part of PayPal*

> *SDK library. Inside such library,* `request.getParameterMap()` *is called to extract all the parameters of the HTTP request, which are used to populate the map object returned by* `ipnlistener.getIpnMap()`. *However, as such parameters are read dynamically at runtime, the OpenAPI/Swagger schema has no knowledge of them (as for this SUT the schema is created automatically with a library when the API starts). Therefore, there is no info to use an HTTP parameter called* `mc_gross` *of type double'' [94].*

Creating a method replacement for `request.getParameterMap()` would not help much here, as, at that point in time when it is called, there is no information yet on the parameter name `mc_gross` that is going to be read later on in the SUT's business logic. There is the need for a more general solution that is able to handle also these cases.

Our novel solution in EvoMaster^N works as follows. First, when we make HTTP calls, we add a ''fake'' HTTP header (e.g., `x-EMextraHeader123`) and a ''fake'' query parameter (e.g., `EMextraParam123`). Then, each time any collection (e.g., lists, sets and maps) is queried with a key/value X, we check if such collection contains our extra header or parameter names. All data structures are already instrumented (e.g., recall Section 4.1.1) to enable heuristic computations and taint analysis, so this is just an extra check done inside those method replacements. Then, if there is a match, we check if X is an already known header/parameter name (e.g., from the OpenAPI schema definition). If not, then there might be a good chance that we are in a case like in Figure 4. The intuition here is that an HTTP server/framework would read all incoming headers/parameters and put them in a data structure, and then check for specific names in such data structure based on what the business logic of the SUT needs (e.g., `mc_gross` in this case). If that happens, then at the next fitness evaluation, we replace the name of the fake header/parameter with X, with a randomly initialized or tainted value. Then, with taint-analysis on the input of `Double.valueOf` we can further infer in the following fitness evaluation that the value of X should be turned into a double. Besides dealing with method calls on collections (e.g., search for a key in a map), we also check for these extra headers/parameters in every string comparison (e.g., in `String.equals()`).

This approach is fully automated, and it does solve the problem like in the example in Figure 4. However, there are two important aspects to consider here. First, it is performance. Sending extra query parameters and headers has a negligible cost, but checking for string matching might not, especially not on large collections. This cost can be justified if the OpenAPI schema is underspecified. However, it would be a clear performance loss if it is not. But we cannot really know if there is any missing header or parameter in the schema before applying our technique. What can be done is to try to minimize its computational cost. We apply two simple heuristics. First, the sending of extra parameters and headers is done only for a short period of time (e.g., currently 10% of the search budget, like for the first 6 minutes if the search budget is 1 hour). Second, if the analyzed collection is too large (e.g., more than 16 elements), then it would be unlikely that it represents the storing of HTTP headers or query parameters. If so, in these cases we simply skip any string matching checks in the instrumented method replacements.

Another rather peculiar issue is that, as we found out during some preliminary experiments, there might be some special headers automatically handled by the enterprise framework used by the API. For example, in older versions of Spring the HTTP filter `HiddenHttpMethodFilter` was active by default. This would handle a special query parameter named `_method`, to enable to change the HTTP verb of incoming requests. For example, a user could make a POST HTTP request with a query part `?_method=PUT` to tell Spring to consider the request as a PUT. This feature comes from old time requirements, before the widespread use of JavaScript and AJAX, when it was possible to do only GET and POST requests from a HTML page (e.g., by clicking on a `<a>` link or submitting a `<form>`). What happened here is that our technique would automatically discover such hidden

parameter _method, and use it in the evolving test cases. This provides no benefit to the search, and actually just lead to plenty of useless tests that return a 405 HTTP status code (''Method Not Allowed''), e.g., when sending random strings such as ?_method=Pfre5. The solution here is to simply ignore any newly discovered parameter that is named _method.

4.3 Underspecified SQL Constraint Schemas

Data in a SQL database can have constraints, which are checked each time a new entry is added. Constraints could be as simple as stating that the entries of a column are unique, or that they might or might not be nullable. However, there might be more sophisticated cases, in which custom constraints can be written with CHECK directives in the SQL table schema.

For a white-box fuzzer, it is important to be able to create data directly into the SQL databases as part of the fuzzing process. This is needed to be able to test scenarios when the same SQL database is used by more than one application (e.g., in producer/consumer scenarios, or when data is populated by external batch jobs and the tested API is read-only). Also, it might improve the readability of the generated tests if some specific needed data in the SQL database could only be obtained with a long sequence of HTTP calls to the API. For this reason, EvoMaster^O is able to automatically generate data into SQL databases as part of the search [27]. During the search, it analyzes all executed SELECT commands, to see which ones return no data. In those cases, it will automatically insert data in the queried tables, with the fitness function aiming at solving the constraints in the WHERE clauses of those empty SELECTs [27]. Each test case will then be enhanced with extra initializing actions for the database, whose inputs (i.e., the data to insert) will be evolved during the search in the same way as any other element of the HTTP requests [27].

As adding invalid data would be pointless, as the SQL database would just straight-up reject them, when we insert SQL data in EvoMaster we make sure that all (linear) constraints are satisfied. However, there might be cases in which this is not enough. Figure 5 shows a snippet from the class VerificationAppSession in the SUT *cwa-verification-server* in our case study (Section 5). This class is marked with the JPA @Entity notation, and it is used to map data from the SQL table app_session. One problem here is that the numeric variable tan_counter in that table is nullable, i.e., NULL is a valid value for it. However, that entity class declares it with the primitive Java type int, instead of the nullable Java type Integer. If there is a null value for that variable, the parsing of that entity would crash, throwing an exception. Regardless of any JEE/Jakarta constrain annotation (e.g., @Min and @Max), primitive vs. nullable types do implicitly define further constraints. In this particular case, there is a mismatch of constraints between the SQL database and its ORM representation in the API. This is a fault, but whether the fault is in the API (i.e., wrong constraint mapping) or in the SQL database definition (i.e., underspecified constraints) is something that only the authors of that API can really comment on.

Another issue can be seen for columns such as teletan_type. In the database, it is declared as a varchar(10), i.e., a string of at most 10 characters. However, the JPA entity defines a further constraint that such string is actually an enumeration, where only a limited set of values is acceptable. In particular, the enumeration TeleTanType contains only the values TEST and EVENT. A random string in that column would crash the JPA parsing of entries from such table.

Our novel solution in EvoMaster^N for this problem is as follows. First, every single time a class is loaded into the JVM, we check if it is a JPA entity (e.g., by checking if the class has the @Entity annotation). If so, we analyze all JEE/Jakarta constraints defined on it (recall Section 4.1.5), including any implicit nullability check based on primitive types, and any enum declarations. Then we compare these sets of constraints with the constraints derived directly from querying the SQL database [27]. For this, we need to identify the corresponding table in the SQL database. We follow

```
1  @Entity
2  @Table(name = "app_session")
3  public class VerificationAppSession implements Serializable {
4
5    private static final long serialVersionUID = 1L;
6
7    @Id
8    @GeneratedValue(strategy = GenerationType.IDENTITY)
9    @Column(name = "id")
10   private Long id;
11
12   @Version
13   @Column(name = "version")
14   private long version;
15
16   @Column(name = "created_at")
17   private LocalDateTime createdAt;
18
19   @Column(name = "updated_at")
20   private LocalDateTime updatedAt;
21
22   @Column(name = "hashed_guid")
23   private String hashedGuid;
24
25   @Column(name = "hashed_guid_dob")
26   private String hashedGuidDob;
27
28   @Column(name = "registration_token_hash")
29   private String registrationTokenHash;
30
31   @Column(name = "tele_tan_hash")
32   private String teleTanHash;
33
34   @Column(name = "tan_counter")
35   private int tanCounter;
36
37   @Column(name = "sot")
38   @Enumerated(EnumType.STRING)
39   private AppSessionSourceOfTrust sourceOfTrust;
40
41   @Column(name = "teletan_type")
42   @Enumerated(EnumType.STRING)
43   private TeleTanType teleTanType;
```

Fig. 5.   Snippet of the JPA entity class VerificationAppSession in the SUT *cwa-verification-server*.

the same algorithms as done in JPA implementations such as Hibernate to resolve table and column names, which might be the same as the Java class/field names, or overridden with annotations, like VerificationAppSession vs. app_session (see Figure 5). If the JPA entity provides more constraints, then we use those constraints when inserting data into the database. For example,

we would not insert a NULL into the column `tan_counter`. However, with a small probability, for robustness testing any now and then we also insert data into the relational database that does not satisfy these extra constraints.

4.4 Timed Events

In a backend system, it is possible to schedule tasks at precise time intervals. For example, in the API *catwatch* used in our case study (Section 5) a task is executed at 8:01am each day, to fetch project data from GitHub. In the API *ocvn-rest* data is imported each day at 3am and 9pm. These background tasks are set with Spring annotations such as `@Scheduled`, and are executed independently from the HTTP requests to the API.

These background tasks introduce few complications for testing. First, generated tests could become *flaky* if the background tasks have side-effects (e.g., adding or deleting records from a database). Second, it might invalidate empirical comparisons of fuzzers (and their different settings) when measuring code coverage metrics. It is not uncommon that experiments for this kind of systems take days to run. For example, when running experiments with a fuzzer X at 2:59am on *ocvn-rest*, we might get better results than a fuzzer Y at 4am simply because the code executed by the background task would contribute to the measured code coverage.

This is a particular problem for black-box fuzzers, as there is no programmatic way (at least in Spring) to disable all the scheduled tasks when starting the API for testing. The API would need to be *manually modified* to enable disabling background tasks during fuzzing experiments. To handle this threat to the validity of the experiments, with a white-box approach using bytecode instrumentation in EvoMaster^N we simply remove any `@Scheduled` annotations when classes are loaded into the JVM (and so the tasks are not executed).

5 EMPIRICAL STUDY

In this paper, we have carried out an empirical study to answer the following research questions:

RQ1: *What is the impact on line coverage and fault detection of our novel white-box techniques?*
RQ2: *How do the heuristic computation overhead and search budget correlate?*

With **RQ1**, we aim to study if our novel techniques provide practical benefits. Experiments are run with a fixed search budget (i.e., 1 hour). But our novel techniques have a computational cost. Even if these novel techniques provide a benefit, such a computational cost might hinder the final results, e.g., by reducing the number of fitness evaluations (e.g., HTTP calls) that can be done within the same time budget. Given a large time budget might not be beneficial, as at a certain point the progress of the fuzzers will stagnate (i.e., they reach a so called ''local optimum''). So, in case in which we get worse results for fixed budget, with **RQ2** we analyze what would happen if a larger search budget (i.e., 10 hours) is used to balance the extra computational cost of more sophisticated techniques. Will the novel techniques lead the fuzzer to stagnate on the same local optimum? Or will they lead to find a better local optimum?

5.1 Case Study

To evaluate our novel techniques presented in Section 4, we used all the 14 RESTful APIs present in the EMB corpus [33] at the time of these experiments, in particular version 1.6.1 [32]. EMB is a corpus of Web APIs (including GraphQL and RPC APIs), which we have collected and extended each year with new APIs since 2017. EMB includes the EvoMaster drivers for enabling white-box

Table 1. Statistics of the used SUTs in the empirical study.

SUT	#SourceFiles	#LOCs	#Enbdpoints
catwatch	106	9636	14
cwa-verification	47	3955	5
features-service	39	2275	18
genome-nexus	405	30004	23
gestaohospital	33	3506	20
ind0	103	17039	20
languagetool	1385	174781	2
market	124	9861	13
ocvn	526	45521	258
proxyprint	73	8338	74
rest-ncs	9	605	6
rest-news	11	857	7
rest-scs	13	862	11
restcountries	24	1977	22
scout-api	93	9736	49
Total 15	2991	318953	542

fuzzing on all of these APIs. To obtain more generable results, besides using open-source APIs, we also included in our experiments one API from one of our industrial partners. For this paper, to refer to this API we use the fictional name *ind0*. Table 1 shows some statistics on these 15 RESTful APIs, including number of source files, lines of code and number of REST endpoints in each of these APIs. Note these code statistics count only what present in business logic of those APIs. Statistics on code in third-party libraries (e.g., HTTP servers) are not included here.

EMB provides APIs of different size and complexity, coming from different domains, covering a variegated set of APIs needed for scientific experimentation [33]. There are two artificial APIs aimed at studying how to deal with numeric (*rest-ncs*) and string (*rest-scs*) constraints. The other 12 APIs are all taken from GitHub: some APIs come from public administrations (e.g., *ocvn*), or are widely popular tools that provide a REST interface (e.g., *languagetool*). A full description of these APIs can be found at [3, 33], including all the source code, build scripts, and links to the original repositories these APIs were collected from throughout the years.

5.2 Experiment Settings

In this paper, we have carried out two different sets of experiments. In the first set, we considered and compared 6 different configurations of $EvoMaster^N$, namely:

Base : default version of EvoMaster, without any of our novel techniques (i.e., $EvoMaster^O$), besides what presented in Section 4.4. This latter is on in the *Base* version as well because it can impact the fairness and soundness of the comparisons.

TAOS : short for "<u>T</u>aint <u>A</u>nalysis <u>O</u>n <u>S</u>ampling", in which in $EvoMaster^N$ we use tainted values (with a certain probability, e.g., 90%) when test cases are sampled, and not just when

test cases are mutated as done in EvoMaster^O [28]. This is a rather minor modification to the MIO algorithm [23], but turned out that can have quite an impact on results.

TT : all the <u>T</u>estability <u>T</u>ransformations presented in Section 4.1 are activated.

TT+OpenAPI : configuration TT plus the handling of underspecified OpenAPI schemas presented in Section 4.2. Note that the techniques presented in Section 4.2 relying on instrumentation for JVM collections, and so that is why it requires TT on. Disabling all the other TT heuristics just for the sake of these experiments would had required significant engineering effort, which we did not consider worthy to invest.

JPA : handling of underspecified SQL schemas using the Java Persistence API (JPA) presented in Section 4.3.

All : all new techniques presented in this paper are activated at the same time.

Fuzzing sessions were run for 1 hour each. To take into account the randomness of search-based fuzzing, each experiment was repeated 10 times. In total, considering 6 configurations and 15 SUTs, this required $6 \times 15 \times 10 = 900$ hours, i.e., 37.5 days of computation. To be able to run all these experiments in reasonable time, they were run in parallel (15 at a time) on a HP Z6 G4 Workstation with Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz, 192 GB RAM, with 64-bit Windows 10 OS.

Ideally, the impact of each single new technique (e.g., each single testability transformation presented in Section 4.1) should be studied in isolation. However, that would have a non-trivial engineering cost (e.g., to support enabling only specific subsets of testability transformations in EvoMaster^N's code instrumentator), as well as making the running of all the experiments unfeasible in reasonable time. Using 6 configurations for the experiments was a viable compromise.

Still, it is important to check that each single technique is useful in its own right. EvoMaster is an industry-ready tool (e.g., used daily in large companies such as Meituan on hundreds of microservices [95, 96]), and particular care is taken to verify the correctness of its components. For this goal, EvoMaster has a sophisticated system of end-to-end tests, where it is run on a set of artificial API examples, carefully crafted to study and verify each single of its features [31]. These tests are automatically run in CI (e.g., GitHub Actions). To verify the correctness of our novel techniques, in EvoMaster^N we created artificial APIs and new end-to-end tests for all of them. An example is `ValidEMTest`,[14] used to verify the handling of bean validation presented in Section 4.1.5. All these new system level tests pass, and are currently part of the daily CI testing of EvoMaster. As discussing all these end-to-end tests would take considerable amount of space, we refer the interested reader to the code repository of EvoMaster (e.g., the module `e2e-tests`).

Based on the results of these experiments, after analyzing its results, to get more insight on a potential issue, we carried out a second set of experiments using a single SUT, namely *ocvn*. In this case, fuzzing sessions were run for 10 hours instead of just 1 hour. But only two settings were considered, *Base* and *All*, with 10 repetitions. This took $10 * 10 * 2 = 200$ hours, i.e., 8.3 days. Experiments were run on the same hardware and configurations of the first set of experiments.

In total, the computation cost of our experiments was $37.5 + 8.3 = 45.8$ days. Note that, although experiments can be run in parallel, there is a limit on how many can be parallelized, considering the specification of the employed hardware. For example, each single experiment requires to run few processes, like EvoMaster itself, the tested API and potentially its databases (e.g., Postgres through Docker) if any is in use. This can take significant amount of OS resources, such as RAM and OS-level threads. Overloading the target machine with too many experiments in parallel could have negative impact on time-based comparisons.

---

[14]https://github.com/EMResearch/EvoMaster/blob/master/e2e-tests/spring-rest-openapi-v2/src/test/java/org/evomaster/e2etests/spring/examples/valid/ValidEMTest.java

Table 2. Performance comparisons between the *Base* and *All* configurations, in terms of average (i.e., arithmetic mean) line coverage and average number of detected faults. Results of statistical tests are reported, including p-values and $\hat{A}_{12}$ effect sizes. For p-values lower than the threshold $\alpha = 0.05$, the effect sizes $\hat{A}_{12}$ are shown in bold. We also report the average number of HTTP calls done during the search, and their scaled difference compared to the *Base* configuration, i.e., $\frac{avg(Base) - avg(All)}{avg(Base)}$.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	All	$\hat{A}_{12}$	p-value	Base	All	$\hat{A}_{12}$	p-value	Base	All	Difference
catwatch	42.5	47.1	**0.97**	< 0.001	19.4	25.2	**1.00**	< 0.001	3120	14352	-360.04
cwa-verification	47.4	57.6	**1.00**	0.010	7.4	12.0	**1.00**	0.010	151478	122480	+19.14
features-service	81.5	81.5	0.39	0.396	33.2	34.1	0.61	0.424	182534	184688	-1.18
genome-nexus	36.7	36.5	0.44	0.705	20.0	21.0	0.66	0.216	59304	41742	+29.61
gestaohospital-rest	39.6	39.6	0.50	1.000	22.0	22.0	0.50	1.000	240280	229379	+4.54
ind0	12.4	23.1	**1.00**	< 0.001	44.1	59.1	**1.00**	< 0.001	232335	177159	+23.75
languagetool	41.7	40.0	0.29	0.146	8.3	11.2	0.75	0.080	13713	21742	-58.55
market	48.6	47.5	0.28	0.117	20.0	20.2	0.54	0.781	27731	27270	+1.66
ocvn-rest	37.1	37.1	**0.84**	0.008	553.8	500.0	**0.00**	< 0.001	136751	103057	+24.64
proxyprint	53.2	53.7	0.57	0.633	83.6	84.7	0.56	0.688	40095	38105	+4.96
rest-ncs	93.0	93.0	0.50	1.000	6.0	6.0	0.50	1.000	275741	262458	+4.82
rest-news	66.9	67.7	**0.82**	0.008	8.0	7.8	0.40	0.167	300261	288004	+4.08
rest-scs	85.7	86.0	0.61	0.437	12.0	11.9	0.45	0.368	283604	261644	+7.74
restcountries	77.0	77.1	**0.75**	0.014	2.0	2.0	0.50	1.000	233799	224224	+4.10
scout-api	52.9	53.4	0.58	0.567	89.5	88.1	0.39	0.437	157931	135622	+14.13
Average	54.4	56.1	0.64		62.0	60.4	0.59		155912	142128	-18.44
Median	48.6	53.4	0.58		20.0	21.0	0.54		157931	135622	+4.82

5.3 Results for RQ1

Table 2 shows the results in details for *All* configuration compared to *Base*. We follow the statistical guidelines from [26], reporting p-values of Mann-Whitney-Wilcoxon U tests and Vargha-Delaney standarized $\hat{A}_{12}$ effect sizes. Results are compared in term of line coverage and detected faults. Detected faults are based on 500 HTTP status codes (server errors) and mismatches of the responses from the OpenAPI schemas for each distinct endpoint (more details on what kind of faults can be detected this way can be found in [72]). Note that there exist also other kinds of metrics that could be used for comparisons, such as for example branch and path coverage. For sake of simplicity and space, we just report the two most common metrics used by practitioners in industry.

As our novel techniques in EVOMASTER^N could introduce some non-trivial computation overhead in the fitness function, in Table 2 we also report the number of HTTP calls done during the search. This could give some insight on the computational cost of our techniques. The less efficient (i.e., more computationally expensive) a technique is, the fewer number of HTTP calls can be done during the search budget (e.g., 1 hour in our case). However, this needs to be analyzed with care, as the cost of each HTTP call is not the same. For example, a call that returns immediately a 400 status code (user error) due to an invalid input would be faster than a successful 200 code call that executes large parts of the API's code, including interactions with external databases. Better heuristics that lead to cover more code might result in executing HTTP calls that are more expensive to run, regardless of computation cost of the heuristics themselves.

Table 3. For each of the six analyzed configuration, we report average line coverage and average number of detected faults. Results that are statistically different from *Base* (at $\alpha = 0.05$ level) are reported in bold.

SUT	Base		TAOS		TT		TT+OpenAPI		JPA		All	
catwatch	42.5	19.4	41.9	18.7	**45.7**	**25.3**	**47.9**	**26.2**	**45.1**	19.5	**47.1**	**25.2**
cwa-verification	47.4	7.4	47.5	7.4	47.6	8.2	**47.6**	8.0	**57.0**	**10.2**	**57.6**	**12.0**
features-service	81.5	33.2	81.5	33.9	81.5	34.8	81.4	33.2	81.8	**35.5**	81.5	34.1
genome-nexus	36.7	20.0	37.3	20.6	36.9	20.9	36.4	20.5	36.3	20.1	36.5	21.0
gestaohospital-rest	39.6	22.0	39.5	22.0	39.6	22.0	39.6	22.0	**39.4**	22.0	39.6	22.0
ind0	12.4	44.1	**24.1**	**54.7**	15.6	**50.6**	13.5	**49.1**	13.9	45.0	**23.1**	**59.1**
languagetool	41.7	8.3	41.0	**5.4**	41.1	7.4	41.9	11.2	38.3	7.9	40.0	11.2
market	48.6	20.0	47.6	19.6	47.8	19.3	47.2	19.9	47.2	19.4	47.5	20.2
ocvn-rest	37.1	553.8	37.1	**515.3**	37.1	539.9	37.1	547.0	37.1	550.7	37.1	**500.0**
proxyprint	53.2	83.6	52.9	86.9	51.5	82.6	54.1	86.2	54.0	82.1	53.7	84.7
rest-ncs	93.0	6.0	93.0	6.0	93.0	6.0	93.0	6.0	93.0	6.0	93.0	6.0
rest-news	66.9	8.0	67.4	8.0	**67.6**	7.8	67.3	7.8	66.8	**7.6**	**67.7**	7.8
rest-scs	85.7	12.0	85.7	11.9	86.0	12.0	86.2	12.0	86.3	12.0	86.0	11.9
restcountries	77.0	2.0	**77.1**	2.0	77.1	2.0	77.0	2.0	77.0	2.0	**77.1**	2.0
scout-api	52.9	89.5	52.7	89.3	53.0	90.6	53.2	89.4	54.4	90.1	53.4	88.1

To see the results of each of the 6 configurations in isolation, average results are summarized in Table 3. Detailed statistical comparisons of each configuration with *Base* are reported in Appendix.

From these results, for line coverage we can see an average improvement of +1.7% (median +4.8%) over the 15 APIs. Results are statistical significant for only 6 APIs, with no statistically worse results. On these APIs improvement are either ''small'' (e.g., less than 1% for *ocvn*, *rest-news* and *restcountries*), ''medium'' (+4.6% for *catwatch*), or ''large'' (i.e., +10.2% for *cwa-verification* and +10.7% for *ind0*). Note that the terms used here small, medium and large are subjective, and so technically arbitrary.

Regarding fault detection, statistically better results are found in 3 APIs. However, there are statistically *worse* results for 1 API, namely *ocvn*. There is a large number of detected faults in *ocvn* (e.g., more than 500), which is related to the large number of endpoints in this API (i.e., 258, recall Table 1). This outlier significantly impacts the statistics over the whole 15 APIs. Although the average decreases from 62.0 to 60.4, the median increases from 20.0 to 21.0. This is a special case in which the average value is roughly three times the median one.

We can make few hypotheses on why code coverage was significantly improved in only 6 out of 15 APIs.

- The case of *rest-ncs* is easy to explain, as all maximum achievable coverage is already obtained [94], and so no further improvement is technically possible. Running experiments on a ''solved'' SUT such as *rest-ncs* is still valuable, for example to check if a novel technique does not make performance worse.
- New code heuristics do have an impact only if the code in which they are applied is executed by the tests. If that code is never reached, those new heuristics have no way to contribute to better performance. A possible example of this phenomenon is *gestaohospital*, where

achieved line coverage is 39.6%. All of its missed branches seem related to interactions with a MongoDB database [94], for which currently EVOMASTER has no heuristics. Once new techniques and heuristics to handle MongoDB databases are designed and implemented, more code of this API would be covered. This new reached code might contain structures (e.g., branch statements) for which our novel heuristics could be helpful, and so improve performance even further.

- Even if a novel heuristic provides better gradient for the search, its computational cost might be non negligible, which might lead to fewer fitness evaluations. Those 2 contrasting phenomena might balance themselves out. A possible example here is *genome-nexus*, where although code coverage stays very similar (with no statistically significant difference), the number of HTTP calls is drastically reduced, from 59k to 41k.
- Improvements could be applicable only on small part of the API's code. For example, an heuristic to better handle URL strings might have only a small impact on coverage if there is only a single call to new URL(x) in a code base of thousands of lines of code. Small improvements could be masked by the variance of the randomized process, and so be hard to detect. There are still many open problems in white-box fuzzing of Web APIs [94], and not all of them have same impact among all APIs. However, you could have a single branch that, once its constraints are solved, could lead to the execution of further thousands of lines of code (e.g., if related to input validation done at the beginning of HTTP call evaluation). This is hard to determine before running experiments.
- When presenting a set of new techniques, e.g., X and Y, they might have conflicting side effects among them. For example X could improve performance whereas Y could reduce it, depending on the API. Studying both at same time might mask out the benefits provided by X. An example of this is *catwatch* in Table 3, where each single technique improves performance but TAOS (although this latter is not statistically significant). When all are combined together, performance seems worse than just using TT+OpenAPI. The case of TAOS is quite peculiar. It gives drastic improvements to *ind0*, nearly doubling the achieved code coverage, from 12.4% to 24.1%. However, it also significantly reduces the number of detected faults in *ocvn*. Applying a new technique Y only based on properties of a SUT could lead to address this issue, although how to determine which properties to use to estimate the impact of Y does not seem trivial.

> **RQ1**: *Statistically better results were achieved on 6 out of 15 APIs, with average line coverage improvements up to 10.7%. With a search budget of 1 hour, statistically worse results were obtained for fault detection in 1 API, namely **ocvn**.*

5.4 Results for RQ2

In the first set of experiments, our novel techniques achieved better results but in one case, namely *ocvn*. The computation cost of our novel techniques is not negligible on this API, as the number of HTTP calls is reduced from 136k to 103k. The question here is what would happen if the fuzzing would be run for longer. The choice of 1 hour for the experiments is technically arbitrary, as arbitrary as 42 minutes or 24 hours.

After running for a certain amount of time, a search algorithm will be stuck on an *optimum* (either *global* or more likely a *local* one). Running the search for longer would not help much if the algorithm cannot escape from the local optima. The search performance would reach a so called *plateau*, and stagnates. There is decades of research effort on addressing this problem, e.g., for example with techniques such as *fitness sharing* [84] to increase population diversity in population-based evolutionary algorithms. The tradeoffs between *exploration* and *exploitation* of
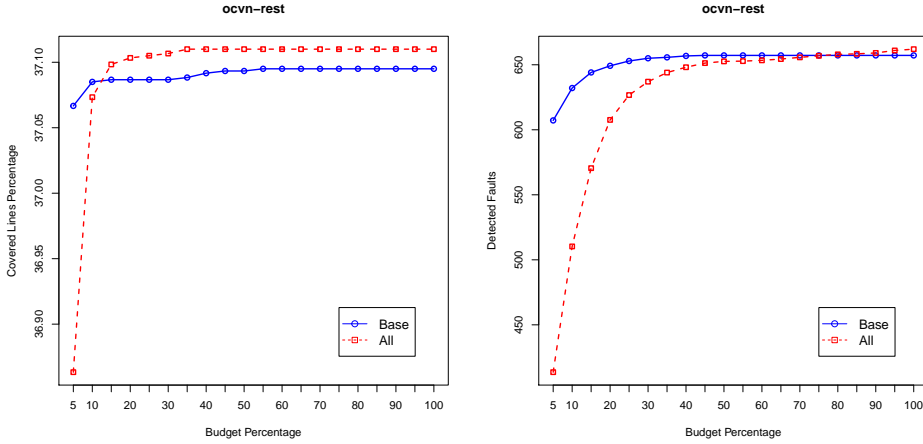
Fig. 6. Performance of *Base* and *All* configurations on *ocvn*, with search budget of 10 hours. Results are displayed for each 5% intervals of the search (i.e., every 30 minutes).

the search landscape applied by a search algorithm do impact how, when and what type of local optimum would be reached. In this regard, the MIO [23] algorithm used in EVOMASTER applies few of these techniques. Still, without a fitness function that can provide gradient to the search, such search would degenerate in a so called random walk on ''fitness plateau'' once a local optimum is reached, preventing further improvement. For all these reasons, the comparisons of 2 algorithms (or algorithm variants) might give very different results based on the used search budget, especially if using different fitness functions. A more sophisticated and expensive fitness function could give worse results for ''low'' time budgets, and better for ''higher'' budgets.

Figure 6 shows the results of the second set of experiments. Those focus only on the API *ocvn*, with a search budget of 10 hours. For low search budgets, the *All* configuration gives worse results. With increasing budget, both configurations *Base* and *All* improve in performance, although *Base* does plateau to lower values. For line coverage, *All* takes over after 1 hour. For number of detected faults, it takes over after 9 hours.

So, based on these results, which of the 2 configurations is ''better''? In a software engineering context, it all depends on how common those time settings are in practice among practitioners. Unfortunately, we do not have such information. Most of the work done in the literature on automated testing of REST APIs has not considered its use among practitioners [54]. This is possibly because there is currently no popular REST API fuzzer widely used in industry. Even when considering mature fuzzers in other popular domains, such as data parsers, there is no much research work aimed at studying how practitioners use those fuzzers [81].

It can be speculated that, considering the life-time of the development of a SUT which can be in years, fuzzing sessions could be relatively *long*. This is particularly the case if the fuzzing can be integrated in remote CI servers (e.g., like done in [39] for unit test generation), especially if the outcome of a previous fuzzing session can be reused for following sessions (e.g., using different kinds of test seeding strategies [83]), even if parts of the tested code have been modified. Easy-to-use integration of fuzzers into CI servers is a common request among users of fuzzers [81]. In those scenarios, more expensive techniques could pay off better in the end. Still, hybrid approaches could be designed: e.g., simple, cheap techniques at the ''beginning'' (e.g., first few sessions on CI), followed by more expensive techniques afterwards.

Another point to consider is that, as we got better results with 10 hour budget for *ocvn*, this might (or might not) happen as well for the other SUTs in our case study. Without empirical experiments, this is not possible to tell. Furthermore, even if there is no difference at 10 hours, perhaps there could be a difference at 24 hours, or more. Testing all different kinds of large test budgets is unfortunately not a viable option for academic experimentation, e.g., our experiments already took more than 45 days if run in sequence.

> **RQ2**: *More sophisticated, expensive techniques can require to use longer time budgets before they can pay off.*

6 DISCUSSION

Our analyses show that our novel techniques presented in this paper can improve performance significantly, in some cases. There are several open-issues in fuzzing REST APIs [94], including for example how to deal with MongoDB databases and interactions with external services. In this paper we addressed some of them, including underspecified schemas and some flag problems in existing JDK functions. Where those problems occur, improvement can be significant.

A clear example of this is the handling of SQL schemas (Section 4.3). Based on the data in Table 3, we can see it has improvements in 2 APIs, a moderate +2.6% average line coverage for *catwatch*, and a more substantial +9.6% coverage for *cwa-verification*. How common is such issue in practice? Having such issue showing in 2 out of 15 APIs would mean 13% of the employed case study. Even if improvements can be obtained on only 13% of APIs, there are possibly millions of APIs developed in enterprises around the world. And so, there is practical value in these new presented techniques. Still, EMB is not (and nor can it be) a statistically valid representative [45] of APIs built in industry, so we have no way to state how common this problem actually is in industry. It might be less common than 13%, or even more common.

Cherry-picking for experimentation only APIs in which improvements are obtained (in our case, 6 specific APIs) would be scientifically invalid, and ethically questionable. To reduce bias, given a corpus like EMB, it is important to use all of it. Even if a technique dealing with SQL databases would have for sure no benefits on databaseless APIs such as *rest-ncs* and *rest-scs*, it is still important to study if such a technique has no negative side-effects. If some important API features are missing in the corpus, the corpus can be extended, as we do each year [3, 33]. This also helps reducing risks of designing techniques that overfit for a specific corpus of SUTs, and so would likely not generalize to other SUTs.

Running fuzzing sessions for longer (e.g., 10 hours *vs.* 1 hour) could lead to better results. However, if the search is stuck in a local optimum with a large fitness plateau, even significantly longer time budgets would be of little help, as the search would simply degenerate into a random walk. New techniques would be required to improve the fitness function.

Let us consider the case of *ocvn*, for which we ran a second set of experiments for 10 hour budget. Still, achieved average line coverage does not go much over 37.1%. How come? Many of the inputs in the API calls refer to *ObjectID*s for the database MongoDB. Those input strings are marked with javax annotation constraints such as `@EachPattern(regexp = "^[a-zA-Z0-9]*$")`. In laymen terms, such regular expression allows for strings containing only basic letters and digits, with any size (including empty). This is an easy to sample regex that, even without our novel techniques presented in this paper, there is no issue for EVOMASTER^O to sample valid values such as `1nZQfS5q`. However, that constraint is wrong. When generating test cases, we find many crashes (i.e., 500 status code) with error messages such as ``invalid hexadecimal representation of an ObjectId: [1nZQfS5q]''. In the source code of *ocvn*, such IDs are instantiated with instructions like ``new ObjectId(s)'', where s is a field in the incoming HTTP calls to the API. The class

```
1  public static boolean isValid(final String hexString) {
2     if (hexString == null) {
3         throw new IllegalArgumentException();
4     }
5
6     int len = hexString.length();
7     if (len != 24) {
8         return false;
9     }
10
11    for (int i = 0; i < len; i++) {
12        char c = hexString.charAt(i);
13        if (c >= '0' && c <= '9') {
14            continue;
15        }
16        if (c >= 'a' && c <= 'f') {
17            continue;
18        }
19        if (c >= 'A' && c <= 'F') {
20            continue;
21        }
22
23        return false;
24    }
25
26    return true;
27 }
```

Fig. 7. Code of the function isValid() in the class org.bson.types.ObjectId from the library org.mongodb:bson:3.4.2. Licensed under the Apache License, Version 2.0.

org.bson.types.ObjectId from the library org.mongodb:bson:3.4.2 does a check on the input string, as shown in Figure 7. A missing constraint from the regular expression is that the input string must be of the exact length 24 (see Line 7). For example, the faulty regular expression could be fixed by writing it as ^[a-zA-Z0-9]{24}$.

Still, even with a faulty regular expression, a white-box fuzzer should be able to maximize code coverage. In the case of the if statement with constraint len != 24 on Line 7, already something basic as standard branch distance on numeric values [66] should be enough to reward the search to sample strings of lengths that get closer to the value 24. This does not happen, unfortunately, as the class ObjectId is not part of the business logic of the SUT, and so it is not instrumented for branch distance computations. Indiscriminately instrumenting third-party libraries as well could have huge computational costs, as those could be millions of lines of code, even for small APIs [33]. Smart strategies would need to be designed (e.g., to avoid instrumenting classes that have no impact on the execution flow of the business logic of the API). Furthermore, we would not want to cover the true output (i.e., Line 26) just once, but in all places from which ''new ObjectId(s)'' is called from. This most likely would require inter-method call heuristics, e.g., by adapting techniques presented for unit testing such as [69, 88].

An alternative approach could be to provide method replacements for ObjectId, like done for JDK classes in Section 4.1. On the one hand, providing adhoc transformations for each single method

for the vast amount of third-party libraries out there in the world would be an infeasible task. A more general solution would be preferred (e.g., as experimented for unit testing in [69, 88]). On the other hand, for widely used libraries, and widely used classes/functions in them, it could make sense to provide adhoc solutions. An adhoc solution would be more efficient than a generic one. For example, a method replacement for `ObjectId` could check for tainted values, and treat them like they were matched with the regular expression `^[a-zA-Z0-9]{24}$`. This could significantly boost the search.

7 THREATS TO VALIDITY

To address threats to internal validity, our code implementation for EVOMASTER^N has been carefully tested, with several unit and end-to-end tests [31]. Furthermore, EVOMASTER is open-source, with all the new releases automatically stored on Zenodo for long term storage, like for example version 1.6.1 [30]. Anyone can review its source code. Furthermore, most of our case study is from the open-source corpus EMB [3, 33], which is as well stored on Zenodo. However, we are not allowed to share the API of our industrial partner.

To handle the randomness of the employed algorithms, all experiments were repeated 10 times, and analyzed with the appropriate statistical tests, following common guidelines in the literature [26].

In this paper, we have not compared our novel white-box techniques with any other white-box fuzzer, as none exists for RESTful APIs besides EVOMASTER [54]. Comparing with existing black-box techniques would add only little to the discussion, as there are already few studies that show that EVOMASTER even in its black-box mode gives the best results, and white-box testing, when applicable, significantly improves over black-box testing. The reader interested in tool comparisons is referred to our previous work in [94], and the independent study carried out by Kim *et al.* [65].

Regarding threats to external validity, we used all the REST APIs in the EMB corpus, plus one API from one of our industrial partners. This provides a wide range of different APIs in terms of complexity and type. Still, as for most empirical studies, we cannot guarantee that our results would generalize to other APIs as well.

Our empirical study focused on REST APIs. However, some of our techniques could be applicable and successful in other domains as well. For example, the handling of underspecified SQL schemas could be applicable to other types of enterprise systems that use databases as well, like for example GraphQL and RPC-based APIs. Method replacements to address the flag problem could be used also in other white-box testing contexts, such as for example unit test generation. However, without proper empirical studies, whether our novel techniques would be successful in those cases as well is not something that can be given for granted.

8 CONCLUSIONS

There is a large body of research in software test generation, with successful applications in many different contexts. When it comes to test enterprise systems such as REST APIs, besides our own work with EVOMASTER [29], all work in the literature has been aimed at black-box testing [54]. White-box testing can significantly improve performance on achieved code coverage and fault detection, especially for REST APIs [25, 94]. Considering the widespread use of web services in industry, more research on this topic is warranted.

In this paper we provided novel search-based heuristics to push forward the start-of-the-art in this software testing domain. Experiments on 14 open-source and 1 industrial APIs show the effectiveness of our novel techniques. For example, on a service from the German Covid's app

backend (i.e., *cwa-verification*), our novel handling of underspecified SQL schemas increased average line coverage by 9.6%.

Still, considering an average line coverage of 56.1% with a 1-hour search budget, there are several research challenges that need to be overcome in the future. Several of these issues have been already identified in the literature [94], including how to deal with interactions with external services and databases such as MongoDB. As all of our work is open-source [30], it can be used to bootstrap new research effort in this domain.

What presented in this paper (i.e., EvoMaster^N) is active by default since EvoMaster version 1.6.1. At the time of writing, based on download statistics and direct contact with large companies such as Meituan [96], hundreds of engineers in industry are already benefiting from the scientific research presented in this paper. EvoMaster is open-source on GitHub [2] with releases automatically stored on Zenodo [30]. To enable the replicability of our scientific studies, the code repository of EvoMaster also contains the scripts used to run experiments, and documentation on how to use them.

ACKNOWLEDGMENTS

REFERENCES

[1] [n.d.]. APIs.guru. https://apis.guru/. Online, Accessed November 21, 2023.

[2] [n.d.]. EvoMaster. https://github.com/EMResearch/EvoMaster.

[3] [n.d.]. EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Online, Accessed November 21, 2023.

[4] [n.d.]. Fuzz-lightyear: Stateful fuzzing framework. https://github.com/Yelp/fuzz-lightyear.

[5] [n.d.]. GraphQL Foundation. https://graphql.org/foundation/.

[6] [n.d.]. gRPC. https://grpc.io/.

[7] [n.d.]. Hibernate. http://hibernate.org. Online, Accessed November 21, 2023.

[8] [n.d.]. Language-agnostic HTTP API Testing Tool. https://github.com/apiaryio/dredd.

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

[10] [n.d.]. Programmable Web. https://www.programmableweb.com/. Online, Accessed November 21, 2023.

[11] [n.d.]. RapidAPI. https://rapidapi.com/. Online, Accessed November 21, 2023.

[12] [n.d.]. RestAssured. https://github.com/rest-assured/rest-assured. Online, Accessed November 21, 2023.

[13] [n.d.]. Spring Framework. https://spring.io. Online, Accessed November 21, 2023.

[14] [n.d.]. SQL. https://www.iso.org/standard/63555.html. Online, Accessed November 21, 2023.

[15] [n.d.]. Tcases for OpenAPI: From REST-ful to Test-ful. https://github.com/Cornutum/tcases/tree/master/tcases-openapi.

[16] 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.

[17] 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.

[18] 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

[19] 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.

[20] M. Alshraideh and L. 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.

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

[22] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959--1981.

[23] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195--206.

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

[25] Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72--78.

[26] 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.

[27] 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.

[28] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1--34.

[29] 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.

[30] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan, Juan Pablo Galeotti, Seran, Amid Gol, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2023. *EMResearch/EvoMaster: 1.6.1*. https://doi.org/10.5281/zenodo.7821550

[31] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. 2023. Building an open-source system test generation tool: lessons learned and empirical analyses with EvoMaster. *Software Quality Journal* (2023), 1--44.

[32] Andrea Arcuri, Man Zhang, Amid Gol, Asma Belhadi, Juan Pablo Galeotti, and Seran. 2023. *EMResearch/EMB: v1.6.1*. https://doi.org/10.5281/zenodo.7821607

[33] Andera Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan Pablo Galeotti, Bogdan Marculescu, and Seran Susruthan. 2023. EMB: A Curated Corpus of Web/Enterprise Applications And Library Support for Software Testing Research. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[34] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 748--758.

[35] 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.

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

[37] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2023. Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Transactions on the Web* (2023).

[38] 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

[39] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 55--66.

[40] 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

[41] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-box Testing of Mass Assignment Vulnerabilities in RESTful APIs. *arXiv preprint arXiv:2301.01261* (2023).

[42] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability* (2022), e1808.

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

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

[45] G. Fraser and A. Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 178--188.

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

[47] 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.

[48] Vahid Garousi and Michael Felderer. 2017. Worlds apart: a comparison of industry and academic focus areas in software testing. *IEEE Software* 34, 5 (2017), 38--45.

[49] Vahid Garousi, Kai Petersen, and Baris Ozkan. 2016. Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology (IST)* 79 (2016), 106--127.

[50] Vahid Garousi, Dietmar Pfahl, João M Fernandes, Michael Felderer, Mika V Mäntylä, David Shepherd, Andrea Arcuri, Ahmet Coşkunçay, and Bedir Tekinerdogan. 2019. Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. *Empirical Software Engineering* 24, 4 (2019), 2540--2602.

[51] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70--76.

[52] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40--44.

[53] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. .NET/C# instrumentation for search-based software testing. *Software Quality Journal* (2023), 1--27.

[54] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* (aug 2023). https://doi.org/10.1145/3617175 Just Accepted.

[55] 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

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

[57] 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

[58] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. 2004. Testability Transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3--16.

[59] 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.

[60] 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.

[61] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 345--346.

[62] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. Automatic Property-based Testing of GraphQL APIs. *arXiv preprint arXiv:2012.07380* (2020).

[63] 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.

[64] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1232--1243.

[65] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289--301. https://doi.org/10.1145/3533767.3534401

[66] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870--879.

[67] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738--24754.

[68] 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

[69] 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.

[70] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[71] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312--2331.

[72] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1--43.

[73] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231--241.

[74] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* (2021).

[75] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In *International Conference on Service-Oriented Computing*. Springer, 399--414.

[76] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 15--21.

[77] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 682--685.

[78] 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

[79] 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

[80] Sam Newman. 2021. *Building microservices*. " O'Reilly Media, Inc.".

[81] Olivier Nourry, GABRIELE BAVOTA, MICHELE LANZA, and YASUTAKA KAMEI. 2023. The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities. *ACM Transactions on Software Engineering and Methodology* (2023).

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

[83] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366--401.

[84] Bruno Sareni and Laurent Krahenbuhl. 1998. Fitness sharing and niching methods revisited. *IEEE transactions on Evolutionary Computation* 2, 3 (1998), 97--106.

[85] Juan Carlos Alonso Valenzuela, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* (2022).

[86] LS Veldkamp, Mitchell Olsthoorn, and A Panichella. 2023. Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs. In *The 16th International Workshop on Search-Based and Fuzz Testing*. IEEE/ACM.

[87] 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.

[88] Sebastian Vogl, Sebastian Schweikl, and Gordon Fraser. 2021. Encoding the certainty of boolean variables to improve the guidance for search-based test generation. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1088--1096.

[89] 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

[90] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[91] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[92] Louise Zetterlund, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2022. Harvesting Production GraphQL Queries to Detect Schema Faults. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 365--376.

[93] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).

Table 4. Same kind of analysis done in Table 2, but for the configuration *TAOS*.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	TAOS	$\hat{A}_{12}$	p-value	Base	TAOS	$\hat{A}_{12}$	p-value	Base	TAOS	Difference
catwatch	42.5	41.9	0.40	0.479	19.4	18.7	0.33	0.241	3120	2669	+14.43
cwa-verification	47.4	47.5	0.70	0.177	7.4	7.4	0.50	1.000	151478	150240	+0.82
features-service	81.5	81.5	0.57	0.604	33.2	33.9	0.49	0.967	182534	182441	+0.05
genome-nexus	36.7	37.3	0.71	0.130	20.0	20.6	0.60	0.460	59304	52567	+11.36
gestaohospital-rest	39.6	39.5	0.35	0.204	22.0	22.0	0.50	1.000	240280	239161	+0.47
ind0	12.4	24.1	**0.99**	< 0.001	44.1	54.7	**1.00**	< 0.001	232335	184305	+20.67
languagetool	41.7	41.0	0.42	0.579	8.3	5.4	**0.23**	0.035	13713	11052	+19.41
market	48.6	47.6	0.29	0.118	20.0	19.6	0.40	0.461	27731	27520	+0.76
ocvn-rest	37.1	37.1	0.49	0.968	553.8	515.3	**0.10**	0.003	136751	143464	-4.91
proxyprint	53.2	52.9	0.53	0.847	83.6	86.9	0.65	0.310	40095	38109	+4.95
rest-ncs	93.0	93.0	0.50	1.000	6.0	6.0	0.50	1.000	275741	276332	-0.21
rest-news	66.9	67.4	0.73	0.063	8.0	8.0	0.50	1.000	300261	297393	+0.96
rest-scs	85.7	85.7	0.51	0.968	12.0	11.9	0.45	0.368	283604	275094	+3.00
restcountries	77.0	77.1	**0.75**	0.014	2.0	2.0	0.50	1.000	233799	233477	+0.14
scout-api	52.9	52.7	0.41	0.496	89.5	89.3	0.49	0.970	157931	157212	+0.46
Average	54.4	55.1	0.56		62.0	60.1	0.48		155912	151402	+4.82
Median	48.6	47.6	0.51		20.0	19.6	0.50		157931	157212	+0.82

[94] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023). https://doi.org/10.1145/3597205 Just Accepted.

[95] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-box fuzzing rpc-based apis with evomaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1--38.

[96] Man Zhang, Andrea Arcuri, Yonggang Li, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2022. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. https://doi.org/10.48550/ARXIV.2208.03988

[97] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[98] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2023. JavaScript SBST Heuristics To Enable Effective Fuzzing of NodeJS Web APIs. *ACM Transactions on Software Engineering and Methodology* (2023).

[99] 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.

[100] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 1--61.

[101] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

APPENDIX

In this appendix, we provide extra tables with more details on the comparisons of our novel techniques.

Table 5. Same kind of analysis done in Table 2, but for the configuration *TT*.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	TT	$\hat{A}_{12}$	p-value	Base	TT	$\hat{A}_{12}$	p-value	Base	TT	Difference
catwatch	42.5	45.7	**0.89**	0.005	19.4	25.3	**1.00**	< 0.001	3120	12948	-315.04
cwa-verification	47.4	47.6	0.85	0.098	7.4	8.2	0.85	0.075	151478	134539	+11.18
features-service	81.5	81.5	0.46	0.795	33.2	34.8	0.68	0.188	182534	198295	-8.63
genome-nexus	36.7	36.9	0.57	0.650	20.0	20.9	0.66	0.241	59304	44243	+25.40
gestaohospital-rest	39.6	39.6	0.40	0.398	22.0	22.0	0.50	1.000	240280	228894	+4.74
ind0	12.4	15.6	0.74	0.094	44.1	50.6	**0.99**	< 0.001	232335	204282	+12.07
languagetool	41.7	41.1	0.32	0.190	8.3	7.4	0.42	0.540	13713	14234	-3.79
market	48.6	47.8	0.47	0.855	20.0	19.3	0.33	0.207	27731	28260	-1.91
ocvn-rest	37.1	37.1	0.59	0.499	553.8	539.9	**0.05**	0.001	136751	118203	+13.56
proxyprint	53.2	51.5	0.34	0.336	83.6	82.6	0.41	0.600	40095	35266	+12.04
rest-ncs	93.0	93.0	0.50	1.000	6.0	6.0	0.50	1.000	275741	272240	+1.27
rest-news	66.9	67.6	**0.78**	0.025	8.0	7.8	0.40	0.167	300261	299284	+0.33
rest-scs	85.7	86.0	0.63	0.308	12.0	12.0	0.50	1.000	283604	268197	+5.43
restcountries	77.0	77.1	0.65	0.210	2.0	2.0	0.50	1.000	233799	221756	+5.15
scout-api	52.9	53.0	0.38	0.404	89.5	90.6	0.55	0.733	157931	154334	+2.28
Average	54.4	54.7	0.57		62.0	62.0	0.56		155912	148998	-15.73
Median	48.6	47.8	0.57		20.0	20.9	0.50		157931	154334	+4.74

Table 6. Same kind of analysis done in Table 2, but for the configuration *TT+OpenAPI*.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	TT+OpenAPI	$\hat{A}_{12}$	p-value	Base	TT+OpenAPI	$\hat{A}_{12}$	p-value	Base	TT+OpenAPI	Difference
catwatch	42.5	47.9	**0.96**	< 0.001	19.4	26.2	**1.00**	< 0.001	3120	14493	-364.56
cwa-verification	47.4	47.6	**0.94**	0.019	7.4	8.0	0.80	0.067	151478	127285	+15.97
features-service	81.5	81.4	0.44	0.636	33.2	33.2	0.42	0.582	182534	190030	-4.11
genome-nexus	36.7	36.4	0.43	0.623	20.0	20.5	0.56	0.696	59304	43972	+25.85
gestaohospital-rest	39.6	39.6	0.45	0.681	22.0	22.0	0.50	1.000	240280	230234	+4.18
ind0	12.4	13.5	0.48	0.930	44.1	49.1	**0.84**	0.016	232335	208888	+10.09
languagetool	41.7	41.9	0.49	0.971	8.3	11.2	0.72	0.101	13713	28754	-109.68
market	48.6	47.2	0.31	0.186	20.0	19.9	0.47	0.874	27731	27516	+0.78
ocvn-rest	37.1	37.1	**0.95**	< 0.001	553.8	547.0	0.31	0.177	136751	115378	+15.63
proxyprint	53.2	54.1	0.62	0.491	83.6	86.2	0.57	0.696	40095	40491	-0.99
rest-ncs	93.0	93.0	0.50	1.000	6.0	6.0	0.50	1.000	275741	258372	+6.30
rest-news	66.9	67.3	0.66	0.210	8.0	7.8	0.40	0.167	300261	289447	+3.60
rest-scs	85.7	86.2	0.70	0.119	12.0	12.0	0.50	1.000	283604	269067	+5.13
restcountries	77.0	77.0	0.52	0.899	2.0	2.0	0.50	1.000	233799	223623	+4.35
scout-api	52.9	53.2	0.43	0.623	89.5	89.4	0.46	0.791	157931	139254	+11.83
Average	54.4	54.9	0.59		62.0	62.7	0.57		155912	147120	-25.04
Median	48.6	47.9	0.50		20.0	20.5	0.50		157931	139254	+4.35

Table 7. Same kind of analysis done in Table 2, but for the configuration *JPA*.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	JPA	$\hat{A}_{12}$	p-value	Base	JPA	$\hat{A}_{12}$	p-value	Base	JPA	Difference
catwatch	42.5	45.1	**0.84**	0.014	19.4	19.5	0.48	0.932	3120	3574	-14.55
cwa-verification	47.4	57.0	**1.00**	0.017	7.4	10.2	**0.95**	0.031	151478	137342	+9.33
features-service	81.5	81.8	0.62	0.375	33.2	35.5	**0.78**	0.033	182534	179841	+1.47
genome-nexus	36.7	36.3	0.44	0.677	20.0	20.1	0.51	1.000	59304	54399	+8.27
gestaohospital-rest	39.6	39.4	**0.25**	0.032	22.0	22.0	0.50	1.000	240280	234277	+2.50
ind0	12.4	13.9	0.59	0.513	44.1	45.0	0.50	1.000	232335	219452	+5.55
languagetool	41.7	38.3	0.39	0.436	8.3	7.9	0.38	0.356	13713	11321	+17.45
market	48.6	47.2	0.26	0.080	20.0	19.4	0.37	0.347	27731	28192	-1.66
ocvn-rest	37.1	37.1	0.54	0.746	553.8	550.7	0.32	0.197	136751	138513	-1.29
proxyprint	53.2	54.0	0.65	0.321	83.6	82.1	0.41	0.561	40095	40918	-2.05
rest-ncs	93.0	93.0	0.50	1.000	6.0	6.0	0.50	1.000	275741	271886	+1.40
rest-news	66.9	66.8	0.47	0.824	8.0	7.6	**0.30**	0.034	300261	290365	+3.30
rest-scs	85.7	86.3	0.74	0.064	12.0	12.0	0.50	1.000	283604	277050	+2.31
restcountries	77.0	77.0	0.60	0.425	2.0	2.0	0.50	1.000	233799	230260	+1.51
scout-api	52.9	54.4	0.62	0.384	89.5	90.1	0.48	0.910	157931	148864	+5.74
Average	54.4	55.2	0.57		62.0	62.0	0.50		155912	151083	+2.62
Median	48.6	54.0	0.59		20.0	19.5	0.50		157931	148864	+2.31