JavaScript SBST Heuristics to Enable Effective Fuzzing of NodeJS Web APIs

MAN ZHANG and ASMA BELHADI, Kristiania University College, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

JavaScript is one of the most popular programming languages. However, its dynamic nature poses several challenges to automated testing techniques. In this paper, we propose an approach and open-source tool support to enable white-box testing of JavaScript applications using **Search-Based Software Testing (SBST)** techniques. We provide an automated approach to collect search-based heuristics like the common *Branch Distance* and to enable *Testability Transformations*. To empirically evaluate our results, we integrated our technique into the EvoMaster test generation tool, and carried out analyses on the automated *system testing* of RESTful and GraphQL APIs. Experiments on eight Web APIs running on NodeJS show that our technique leads to significantly better results than existing black-box and grey-box testing tools, in terms of code coverage and fault detection.

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

Additional Key Words and Phrases: JavaScript instrumentation, NodeJS, white-box test generation, SBST, fuzzer, Babel

ACM Reference format:
Man Zhang, Asma Belhadi, and Andrea Arcuri. 2023. JavaScript SBST Heuristics to Enable Effective Fuzzing of NodeJS Web APIs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 139 (September 2023), 29 pages.
https://doi.org/10.1145/3593801

1 INTRODUCTION

As of 2020, according to the official statistics of GitHub [23] (the most popular hosting solution for open-source software), JavaScript has been for several years the most common programming language for its hosted repositories. JavaScript has been mainly known for running code in the browser, but it can also be used for server-side applications using NodeJS [15], as well as desktop applications (e.g., using Electron [7]) and mobile apps (e.g., using Ionic [9]).

JavaScript (more formally, ECMAScript [6]) is a *weakly* and *dynamically* typed language. These language properties differentiate JavaScript from other popular programming languages, such as

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 864972).
Authors' addresses: M. Zhang and A. Belhadi, Kristiania University College, Kirkegata 24-26, 0153, Oslo, Norway; emails: {man.zhang, asma.belhadi}@kristiania.no; A. Arcuri, Kristiania University College and Oslo Metropolitan University, Kirkegata 24-26, 0153, Oslo, Norway; email: andrea.arcuri@kristiania.no.

1049-331X/2023/09-ART139 $15.00
https://doi.org/10.1145/3593801

Java, C#, and C/C++. Unfortunately, the lack of *strong typing* significantly complicates static and dynamic analyses [27], including automated test generation. An approach to add types to JavaScript is TypeScript, which is a programming language that is a superset of JavaScript (i.e., any JavaScript code is valid in TypeScript). However, TypeScript does not run on either the browser or the server (e.g., NodeJS runtime), and has to be transpiled to JavaScript (e.g., similarly to how Java programs need to be compiled into bytecode to run on the JVM).

In this paper, we show how **Search-Based Software Testing (SBST)** [25, 65, 66, 79] techniques can be used for JavaScript applications (and applications in other languages like TypeScript that get transpiled into JavaScript). We describe how common techniques in the literature like the *Branch Distance* [26, 55, 69] can be used for JavaScript source code. In particular, we deal with the issues of dynamic types and exception handling in composed predicates and short-circuit operations. To the best of our knowledge, this is the first full solution for the handling of SBST heuristics for JavaScript source code. We implemented our technique in an open-source tool, as a fully automated plugin for Babel [1] (which is a popular tool for JavaScript code transformations).

Our instrumentation would enable the use of SBST techniques in different testing contexts, for example, *unit testing*. To empirically evaluate the effectiveness of our JavaScript instrumentation, we integrated it into the EvoMaster [28] tool, which does automated *system test* generation for RESTful and GraphQL APIs running on the JVM (e.g., compiled from programming languages such as Java, Kotlin, and Scala). This can be considered as a form of *fuzzing*,[1] as we generate system level test cases to automatically find faults (e.g., due to software crashes that lead to responses with HTTP 500 status code). EvoMaster uses evolutionary algorithms like MIO [29] enhanced with adaptive hypermutation [90], based on fitness functions that exploit SBST heuristics.

The empirical study was conducted on eight different **systems under test (SUTs)** running on NodeJS, in which we compared our white-box SBST approach with grey-box random testing (in which only coverage metrics are employed, and no SBST heuristics are used), and three black-box approaches: the one provided by EvoMaster, RESTler [40], and RestTestGen [86]. The results of our experiments show that EvoMaster integrated with our SBST heuristics for JavaScript achieves the best results, both in terms of code coverage and fault finding.

This paper is an extension of [93]. Here, we provide further novel research contributions by designing techniques to enable *testability transformation* with *taint analysis* [35] in JavaScript programs (Section 3.5), as well as improving heuristics on short-circuit operations with purity analysis (Section 3.4) and accesses to arrays/objects (Section 3.6) and handling of the ternary operator (Section 3.7). Furthermore, besides adding one more RESTful API for the experiments, we also extended the experiments to include two GraphQL APIs, as now EvoMaster can generate tests for this kind of Web API as well [43] (Section 4.1). We added new analyses to evaluate performance of the compared algorithms through time (Section 4.3.2). Furthermore, we also repeated all the experiments with the other black-box fuzzers (i.e., RESTler and RestTestGen), to use their most recent versions, running them for a longer amount of time, 1 hour (Section 4.3.3).

This paper provides the following contributions to the state of the art:

- A full working approach to enable SBST for JavaScript applications;
- A novel algorithmic approach to handle SBST heuristics in JavaScript applications;

[1]The term "*fuzz*" originated from Miller *et al.* [81] that refers to a program which does facilitate "generating random input strings" in order to test UNIX utilities [81]. "*Fuzzing*" (or '*'Fuzz testing*") now has been widely used to refer to an automated software testing technique for detecting faults by automatically generating tests or test inputs [57, 75, 89, 96] that has been applied in various contexts of software testing, e.g., security testing [56, 58, 59], GUI testing [61, 85], testing of Android system services [49], compiler testing [50, 72], and Web API testing [40, 44, 74, 91, 92]. An implementation of "*fuzzing*" can be called as "*fuzzer*" [91, 96].

- An empirical study on the testing of RESTful and GraphQL APIs showing the improvement of our approach compared to three state-of-the-art tools;
- An open-source implementation, together with all the scripts used to run the experiments.

The rest of the paper is organized as follows: related work is discussed in Section 2. Our proposed approach is explained in Section 3 followed by its empirical study (Section 4). We discuss threats to validity in Section 5, and conclude the paper in Section 6.

2 RELATED WORK

SBST [25, 65, 66, 79] techniques have been successfully used for more than three decades, in several different testing contexts. Popular SBST examples are EvoSuite [53] for unit testing, and Sapienz [76] for testing of mobile apps. Evolutionary algorithms are driven by the provided fitness function. To achieve better results, different heuristics are employed to "smooth" the search landscape, providing "gradient" to the search algorithm to reach an optimal solution. In the context of white-box testing, a popular technique introduced in the 90s by Korel [69] is the so-called *Branch Distance*. It was first introduced to handle predicates involving numerical comparisons (e.g., $a < b$), and then later refined for logical operators [55] (e.g., `AND` and `OR`) and string comparisons [26].

To the best of our knowledge, the only existing SBST work that deals with JavaScript instrumentation is the JEDI tool [52], which aims at unit testing of JavaScript code that interacts with the DOM in the browser. However, this research does not specify how the instrumentation was done, nor does it mention many of the issues we solve in this paper. It was claimed that JEDI is open-source [52], but its repository [10] on GitHub is empty at the time of this writing, so it is not possible to verify if such challenges were addressed.

SBST instrumentation poses quite a few challenges, e.g., when dealing with logic operators such as `&&` and `||`, and exceptions during test evaluation. To compute the branch distance, the code needs to be manipulated, but still we need to make sure that the semantics of the SUT are not changed. Tools that work on the JVM (e.g., EvoSuite) have an advantage here, as the bytecode instructions are significantly simplified compared to their original source code. For example, in JVM bytecode there is no logical `AND`/`OR` operators, as those are compiled into a series of base predicates with jump instructions. In the past, in the literature of SBST, there has been work on instrumentation of source code, in particular for the C programming language [70, 88]. However, no full details were given on how the instrumentation was done. At any rate, some of our techniques rely on specific properties of JavaScript (e.g., closures), which would not be applicable to other programming languages such as C.

Regarding dynamic analyses on JavaScript programs with non-SBST techniques, we refer to the survey of Andreasen et al. [27]. Nevertheless, no technique has been proposed that aimed at system test generation for web services (e.g., RESTful and GraphQL APIs) written in JavaScript. Most of the research has been conducted on unit testing (e.g., the more recent [82, 83]). One of the main challenges here is that the entry point of the SUTs are TCP sockets, and this kind of application often uses databases. This is a very different scenario compared to other kinds of applications like parsers where there are no external dependencies. Other testing techniques that are not able to handle networking and databases cannot, therefore, be used in this testing context.

In the past, tools like Jalangi [84] were proposed to enable different kinds of code analyses on JavaScript programs. However, they do not support any of the SBST heuristics that we discuss in this paper.

As far as we know, EvoMaster [28] is currently the only tool that does white-box system test generation for Web APIs, where being able to deal with networking and databases is a major requirement. All other techniques presented in the literature are black-box using different variants of random testing (e.g., [40, 51, 59, 68, 78, 86]), and so would not be able to exploit any source code instrumentation.

3 JAVASCRIPT INSTRUMENTATION

3.1 Tool Support

JavaScript code is not compiled directly by users, although for performance reasons it can get compiled on-the-fly by execution runtime like browsers and NodeJS. When downloading a third-party library with package managers such as NPM, the source code would be downloaded, as there is no intermediate compiled representation (like bytecode in environments such as JVM and .NET). However, it is typical to apply transformations (referred to as *transpilation*) to JavaScript source code. Examples are for minimizing the code size (e.g., by removing unnecessary empty lines, line-return characters and code comments) to make those source files faster to download on the browser, and to support old browsers (e.g., new features of JavaScript can be transpiled into the equivalent code in older versions of JavaScript). Another example is to support different languages, e.g., TypeScript and React JSX, to run on the browser (which only supports JavaScript, and more recently WebAssembly).

At the time of this writing, the most used JavaScript transpiler is Babel [1], which can be easily integrated in package managers such as NPM/YARN and bundlers like WebPack. Babel provides a plugin system, in which different transformations can be applied in sequence.

Our instrumentation for SBST has been implemented as a plugin for Babel, written in TypeScript. When an SBST technique is applied, Babel needs to be called (e.g., from NPM/YARN) to create an instrumented version, which is the one that is going to be used as SUT. Probes are added to the source code of the SUT, where the instrumentation runtime library is automatically added as a dependency.

All the code instrumentation and transformations discussed in this paper are designed to preserve the *semantics* of the SUT. For the same inputs, the instrumented SUT should produce the same outputs as the original non-instrumented version. The motivation here is that test cases generated for the instrumented version should still behave the same when run by the user for debugging and regression testing. However, instrumentation can introduce time delays, as more code is executed. If the SUT strongly depends on time behavior of the runtime of its executing code (in contrast to waits on I/O operations), then such delays might break some time constraints. Fortunately, although possible, we have not experienced this type of issue yet in the context of white-box testing of Web APIs. However, it could happen if the techniques presented in this paper are applied in other domains.

Instrumentation itself is not enough though, as it needs to be integrated with an SBST test generation tool. For our experiments in this paper, we employed EvoMaster [28, 30, 36], which generates system-level test cases for RESTful [31] and GraphQL [43] APIs. EvoMaster has two main components: a *core* process and a *driver* process that is responsible to start/stop/reset the SUT, plus applying code instrumentation with SBST heuristics. The two processes communicate via HTTP, where the *driver* process exposes a series of functionalities as a RESTful API (e.g., having HTTP endpoints to collect coverage information after a test case has been executed by the *core* process as part of the fitness function evaluation). To use EvoMaster, we simply implemented a new *driver* program written in JavaScript, implementing the same APIs of the original JVM driver in EvoMaster. The only modification needed in the *core* program (which is written in Kotlin) was to add a new kind of test output to support JavaScript. Currently, EvoMaster can output test cases (which are sequences of HTTP calls) as JUnit [12] test suite files, in either Java or Kotlin, using the library RestAssured [19] for making the HTTP calls. We simply implemented a further option to output the test cases in JavaScript, as Jest [11] test suite files, using the library SuperAgent [24] for making the HTTP calls.

3.2 Code Coverage

A JavaScript program is composed of a series of source files, each one having code lines with code statements. Each of these will become a testing target, and we keep track in our instrumentation when they get covered by any test case execution. The goal of test generation tools like EvoMaster is to generate test cases that maximize the number of covered targets.

For each statement in the program, we add a probe that, once executed, tells our instrumentation runtime that the statement has been covered. Consider this simple example of a variable assignment in a `test.ts` example file:

```
let x = 0;
```

then, its instrumented version would be:

```
const __EM__ = require("evomaster-client-js").InjectedFunctions;
__EM__.registerTargets(["File_test.ts", "Line_test.ts_00001", "Statement_test.
    ts_00001_0"]);
__EM__.enteringStatement("test.ts", 1, 0);
let x = 0;
__EM__.completedStatement("test.ts", 1, 0);
```

In the first line, we import the declaration of our runtime probes, with a unique name (e.g., `__EM__`) to avoid clashes with the existing variables of the SUT. Secondly, we mark all the existing testing targets in this source file. This is needed to know what have *not* been covered after a test execution (as which probes are executed depends on the control flow of the SUT). Each target gets a unique *id*, with a meaningful name (which helps when debugging). In this case, there are three targets: one for the file, one for the line, and one for the statement in that line.

In this case, two probes are added to the code: before the statement (i.e., `enteringStatement`), and after it (i.e., `completedStatement`). Those functions take as input info to create the unique *id*s for the targets (e.g., the file name, source line, and a unique counter value for each statement on the same line).

For handling the SBST heuristic values, we use the same approach currently used in EvoMaster. Each target will have a heuristic value in the range $h \in [0,1]$, where 1 represents that the target is fully covered, and 0 represents that the target is not even reached during a test case evaluation. Values in between represent how heuristically close a test case was to covering the target.

When `enteringStatement` is executed, the targets for file and line are marked as covered, i.e., $h = 1$. The one for the statement is marked as 0.5, though. The idea here is that statements might throw exceptions, and only once a statement is fully completed we can know that no exception was thrown. This means that in `completedStatement` the heuristic value is then increased to $h = 1$. It is important here to stress the importance of having two separated testing targets for the line and the statement. Assume, for example, a test case in which an exception is thrown in the statement (e.g., inside a method call), and so $h_s = 0.5$. If there was no target for the line, then the test case would not end up in the final output test suite of EvoMaster, as it only outputs test cases for targets that are fully covered [30]. On the other hand, if we only reported the line target with $h_l = 1$, then the search would have no way to know that an exception was thrown, and that there is still the need to do mutations to try to find input data for which an exception is not thrown. Search algorithms like MIO [29] keep sampling and mutating test cases for targets that are not fully covered (e.g., $h_s = 0.5$), whereas, for the targets that are covered, the test cases are saved in an archive, and no longer used in the search (unless there exist copies in the other populations for the non-covered targets [29]). Note, though, that a value like $h_s = 0.5$ does not really give gradient to find input data that leads to no exception. It is used to mainly tell the search to keep trying to mutate that test case.

One problem, though, is that there are some cases in which `completedStatement` cannot be used after a statement. This happens for statements that exit the execution flow, like `return`, `throw`, and `break/continue`. In this case, we replace `enteringStatement` with `completedStatement` before the statement. For example, consider the following code snippet:

```
const x = function(){
   return;
};
```

which would be instrumented into:

```
__EM__.enteringStatement("test.ts", 1, 0);
const x = function () {
        __EM__.markStatementForCompletion("test.ts", 2, 1);
    return;
};
__EM__.completedStatement("test.ts", 1, 0);
```

However, there can still be cases in which a `return` statement could throw an exception, for example when returning the result of a function call, e.g., `return foo(x);`. We still want the search to evolve at least one test case for which no exception is thrown. In this case, the instrumentation would look like:

```
__EM__.enteringStatement("test.ts", 1, 0);
return __EM__.completingStatement(foo(x), "test.ts", 1, 0);
```

Here, `completingStatement` would mark the statement as covered (i.e., h_s=1), and then return the value of its first input (recall that the instrumentation should not change the semantics of the SUT), which is the expression `foo(x)`. Note that, if `foo(x)` throws an exception, then `completingStatement` would not be called, and the heuristic value would remain the same as the one set in `enteringStatement`, i.e., $h_s = 0.5$.

Another simple transformation needed here is that, for `if`, `while`, and `for` statements, we need to add a code block (i.e., curly braces {}) if they have only one inside statement. For example, something like `if(x)foo(y)` needs to be rewritten into `if(x){foo(y)}`. Otherwise, adding probes (e.g., between the `if` and `foo`) would change the control flow execution.

3.3 Branch Distance

The control flow of the SUT can depend on complex predicates, e.g., conditions in `if` statements. The branch distance [69] was introduced in the purpose of providing gradients that evolve test inputs in order to solve those constraints. For example, consider a statement like `if(x===42)`. Here, if the input `x` is taken at random, there would be only 1 out of 2^{64} possibilities to make that predicate true (note that JavaScript has no integer type, but rather it has `number` type, which is a 64 bit double-precision floating-point number). However, a value like $x = 50$ would be heuristically closer to solve that constraint than something like $x = 900$. Here, a branch distance would be defined as $d(x) = |x - 42|$, for any given input x.

To compute those distances, we replace all occurrences of these binary expression operators: `==`, `===`, `! =`, `! ==`, `<`, `<=`, `>`, `>=`. Given a binary expression `A op B`, we replace it with: `cmp(A,"op",B,id)`. The function `cmp` will return the same result of `A op B`. However, internally it will create two new testing targets (based on the unique input `id`): one for when the expression is evaluated as `true`, and one for when it is evaluated as `false`. Note that this transformation is applied anywhere in the code, and not just in the `if` statements. For example, `const x = y > 42` would be instrumented into `const x = cmp(y,">",42 ,0)` (assuming `id` = 0). This also helps to deal with the so called *flag* problem [64].

For each of these new targets, a heuristic $h \in [0,1]$ is computed, which is based on the branch distance (after a remapping transformation). Note that, when `cmp` is called, necessarily one of two conditions will hold: e.g., the predicate is either true or false. So, one of the two targets will be necessarily covered (i.e., $h = 1$). If either the evaluation of `A` or `B` (which could be functions) throws an exception, then `cmp` would not be called anyway.

One challenge here is that JavaScript is *weakly* typed. It is perfectly valid JavaScript code to compare an array to an object. For example, something like `[] > {}` does return `false`, i.e., an empty array is not larger/bigger than an empty object. However, a comparison like `[] == 0` returns `true`, i.e., an empty array and the numeric constant 0 are the same for JavaScript (there are plenty of these "oddities" in the JavaScript language, besides these simple examples). Therefore, the function `cmp` can be called with any type of inputs for `A` and `B`, and their types cannot be guaranteed to be known at instrumentation time. This poses a challenge to determine which branch distance to use (if any).

The solution here is to check the types at runtime, using the JavaScript `typeof` operator. If both `A` and `B` are of type `number`, then Korel's branch distance [69] is used to compute the h heuristics. If they are both of type `string`, then we use the string distances defined in [26]. For all the other input types, we simply use a binary flag: $h = 1$ for the target that is covered (i.e., either the `true` or `false` condition), and a value $h < 1$ for the other (e.g., $h = 0.01$). Note that `cmp` will never assign a value $h = 0$ for any non-covered target. The idea here is that the search should be able to distinguish between a testing target that is not even reached (e.g., inside a code block that is not executed) with a value $h = 0$ which is always lower (e.g., compared to $h = 0.01$) than the heuristics for a target that has been evaluated, even if not covered, regardless of which kind of branch distance we can use. This helps the search to keep on trying to solve that predicate, possibly keeping mutating the current test case that led to the execution of that binary expression operator.

3.4 Logical Operators

A major challenge here is how to deal with `AND`/`OR` operators. In the 90s, the branch distance for those operators was defined [55] as:

$$d(A\&\&B) = d(A) + d(B) \tag{1}$$

$$d(A||B) = min(d(A),d(B)) \tag{2}$$

However, when dealing with source code like JavaScript, those equations cannot be directly applied, due to short-circuited evaluations. For example, in $A\&\&B$, if A is false, then B is not computed (recall it could be a function call that returns a boolean). Computing B to derive its branch distance can lead to breaking the SUT's semantics if B has side-effects (e.g., changing the value of some variables). Unfortunately, there are further problems: how to deal with the branch distance values when there are chained logical operators (e.g., $A||B||C||D$) without breaking the SUT's semantics, and how to deal with exceptions. For example, if in $A||B$ the second clause B throws an exception, we still want to compute and track the branch distance for A when it is false.

To address these issues, in this paper we provide a novel algorithm that relies on the use JavaScript's *closure* feature [6], which allows access to an outer function's scope from an inner function. Each use of || and && gets replaced by a function call (i.e., `or()` and `and()`). However, the two operands A and B get replaced with function call declarations, e.g., `() => A` (note that the arrow operator => is used in JavaScript to define new functions, with the left side being inputs, and right side being the code of the function). Consider the following code:

```
const x = y == 42 || foo.bar();
```

it will be replaced by:

```
const x = __EM__.or(
  () => __EM__.cmp(y, "==", 42, "test.ts", 1, 1),
  () => foo.bar(),
  "test.ts", 1, 0);
```

Writing something like `() => foo.bar()` declares a new function with no inputs, which, once called, executes the function `foo.bar()` and returns its value. When the `or()` function is called, `foo.bar()` is not executed yet. Note the importance of closures here, as inside `or()` we would still need to be able to access the variables `y` and `foo` (which might be local to the function where `const x` is declared).

The importance of using function declarations is that, inside the functions `or()` and `and()`, we can create two new testing targets (for `true` and `false` results of the predicate), before either A or B is executed (which could lead to exceptions). Their execution can be done inside a try/catch, in which, if an exception is thrown, all heuristics can be computed and registered before re-throwing the exception (recall that the semantics of the SUT have to be preserved).

Figure 1 shows our actual implementation of the `or()` function (written in TypeScript). We omit the implementation of `and()`, as it is quite similar to `or()`. At any rate, as our EvoMaster extension is released open-source on GitHub, it is available online, with long term storage on Zenodo [38] (the interested reader can look at the script file `HeuristicsForBooleans.ts`).

There are a few problems that need to be taken care here. First, there is the need to be able to access the branch distance (if any) of the `left` and `right` operands. This is done using a global variable `lastEvaluation`, which needs to be updated inside the `cmp` function before it returns. Note that, in JavaScript, there is no need to worry about concurrent access to such a variable from different threads, as JavaScript has no multi-threading with shared memory.

The class `Truthness` is just an utility, to store the two heuristic values for the two new testing targets (i.e., both possible boolean outcomes). Then, when `left()` is evaluated at line 8, there are three possible outcomes that affect the heuristics: an exception was thrown, the operand had a computed heuristics, or none was computed (e.g., when it was simply a boolean variable). In the first case, *both* the `true` and `false` targets get the worst fitness (i.e., 0.005), but still greater than 0 (line 14), to reward evaluating the `or()` compared to not even executing it. If the operand had no existing heuristics, then the two heuristic values will be 1 (for the covered target), and 0.01 (i.e., `base`) for the other. This depends on the outcome `x` of the `left()` evaluation. If there was an existing heuristic value, before we can use it, it needs to be re-scaled to be not smaller than `base` (line 12), which can be done with a simple function like $d' = b + (1 - b)d$.

The `right` operand can be evaluated only if the `left` one was evaluated as false, without throwing any exception, or if `right` is a *pure* (i.e., with no side-effects) function (line 19). Otherwise, we cannot guarantee the behavior of the SUT would not be changed by our instrumentation if we invoke `right()`. The heuristic computation of `right` is the same as for `left` (line 23).

For determining if the right operand is pure, we use a *conservative* approach, done at instrumentation time. In other words, if there is any possibility that the operand is non-pure (e.g., any kind of function call or access on an object field that could throw an exception if the object is null), then `isRightPure` is marked as false. For example, `x==42` would be marked as pure, whereas `y.x==42` and `x()==42` would not. In other words, we need to make sure that the semantics of the SUT are not changed, even though we might lose gradient in some cases. In future work, this purity analysis could be extended with runtime information, to make a more precise evaluation to better determine when operands are pure.

```
1  or(left: () => any, right: () => any, isRightPure: boolean, fileName: string,
     line: number, branchId: number): any {
2    HeuristicsForBooleans.lastEvaluation = null;
3    const base = 0.01;
4    const exception = 0.005;
5    let xT: Truthness;
6    let x: any;  let xE: any = null;
7    try {
8      x = left();
9      xT = HeuristicsForBooleans.lastEvaluation;
10     if (!xT) {
11       xT = new Truthness(x?1:base, x?base:1);
12     } else {xT = xT.rescaleFromMin(base);}
13   } catch (e) {
14     xT = new Truthness(exception, exception);
15     xE = e; }
16   const leftIsFalse = (!x && xE === null);
17   let h: Truthness;
18   let y: any; let yE: any = null;
19   if (leftIsFalse || isRightPure) {
20     HeuristicsForBooleans.lastEvaluation = null;
21     let yT: Truthness;
22     try {
23       y = right();
24       yT = HeuristicsForBooleans.lastEvaluation;
25       if (!yT) {
26         yT = new Truthness(y?1:base, y?base:1);
27       } else {yT = yT.rescaleFromMin(base);}
28     } catch (e) {
29       yT = new Truthness(exception, exception);
30       yE = e;}
31     h = new Truthness(
32     Math.max(xT.getOfTrue(), yT.getOfTrue()),
33     (xT.getOfFalse()/2) + (yT.getOfFalse()/2));
34   } else {
35     h = new Truthness(xT.getOfTrue(),xT.getOfFalse()/2); }
36   ExecutionTracer.updateBranch(fileName,line,branchId,h);
37   HeuristicsForBooleans.lastEvaluation = h;
38   if(xE){throw xE;}
39   if(leftIsFalse && yE){throw yE;}
40   return x || y;
41 }
```

Fig. 1. Implementation for `or()`.

The computation of the combination of the heuristics for `left` and `right` (line 31) is inspired by Equations (1) and (2), but adapted from branch distances to $h \in [0,1]$. On the one hand, the outcome `true` is taken when either of the two operands is true, so we can take the maximum between these two values (line 32). On the other hand, the outcome `false` happens when both operands are false, i.e., $\neg(A \vee B) = \neg A \wedge \neg B$. Therefore, their heuristic values can be summed together (like in Equation (1)), but then need to be divided by 2 to still remain in the [0,1] range (line 33).

When the `left` operand throws an exception, the heuristics for the `or` is based only on the value of the `left` (line 35). However, an important detail here is that the heuristic value for the outcome `false` still needs to be divided by 2. Otherwise, if after a mutation the `left` operand is no longer true (or no longer throwing an exception), then the computation at line 33 could lead to a worse heuristic value when adding the value for `right`, instead of being rewarded for making `left` evaluated to false.

After the heuristics h are computed for both targets, their values are saved (line 36), and `lastEvaluation` is updated (line 37). Then, we need to make sure that we do not change the behavior of the SUT. If any exception was thrown, we re-throw it (lines 38–39). Otherwise, the original output of the `||` is returned.

The last piece of the puzzle is how to deal with the negation !, for example in expressions like `!(x==42)`. As we already create two testing targets for each boolean expression, there is no need to define any new heuristics for the ! operator, as no new gradient information can be provided to help the search. However, we still instrument it, by replacing any `!A` with a function call `not(A)`. The reason is that the `lastEvaluation` variable needs to be updated (if any), by swapping the heuristic values of the `true` and `false` outcomes.

3.5 Testability Transformations

To be able to effectively test an application, an SBST approach needs a fitness function that can smooth the search landscape, providing easy-to-follow gradients for the algorithm to reach the global optima. One major issue is when there is no gradient, and the heuristics for covering a target is just a *flag*, i.e., either the target is covered or not covered. This is an issue that has been studied in the past in the literature of SBST, as there are several sources for these kind of flags [26, 41, 42, 45, 60, 62, 63, 71, 73, 80, 87]. Consider the following example:

```
if(x.includes("a"))
```

If the variable `x` is a string, then the predicate would be true if such variable contains the letter `'a'` inside it. This is a flag, as the condition is either true or false, and the branch distance (Section 3.3) gives no help here. To address this issue in function calls that return booleans, or that throw an exception if the input is invalid, in [35] we proposed a novel *testability transformation* technique. For a set of specified methods in the existing APIs of the supported programming languages (e.g., the JDK in [35]), we provide *replacement methods*. During instrumentation, each of these methods gets replaced with a new version, which is semantically equivalent. In other words, the execution of the SUT is not changed. However, internally, such replaced methods will create two new testing targets, one for when the function returns `true` (or throws an exception) and one for when it returns `false` (or does not throw an exception). These extra targets will then be added to the search, and optimized for like any other testing target.

For these replacement methods, the gradient for the two outcomes will depend on the semantics of the replaced method. For example, for something like `x.includes(‘‘a’’)`, for the `true` outcome it would take the minimum distance (as numeric representation of a char) of `‘‘a’’` from each character in the string (i.e., something like $d = min(|x[i] - “a"|)$.

For the most used API methods in the JDK, in [35] we defined custom heuristics. For JavaScript, we provide the same kind of method replacements, e.g., for handling API functions like `includes` and `startsWith` on string objects, as well as operations like `includes` on collections. For instance, a heuristic of a `true` target for the collection is calculated as Equation (3) where C is the collection and k is the value or variable to search if C `includes` it. Note, `null` and `undefined` are allowed in a collection in JavaScript. The `includes` method is first executed. If the result is true, $h = 1$ indicates that the target is *covered*. If C is empty or k is `null`/`undefined`, low heuristics (i.e., 0.05

and 0.1, respectively) are provided for indicating that they are far from covering the target. As an empty C always leads the result to be false with any value of k, we provide a lower value than k with `null`/`undefined`. For other cases, we calculate all heuristics between k and every value in C (with the '`of`' operator[2]) using branch distance (Section 3.3), then take the maximum value as the heuristic.

$$h_c(k, C) = \begin{cases} 1 & \texttt{C.include(k)} \text{ is true} \\ 0.05 & |C| = 0 \text{ reached but empty} \\ 0.1 & k \text{ is null or undefined} \\ max_{1 \le i \le |C|} h^{'}(k, C[i]) & \text{otherwise, note } h^{'} \text{ is a re-scaled value} \end{cases} \quad (3)$$

The heuristics are the same as defined in [35], although some of the method names are different (e.g., the equivalent of JavaScript `includes` on strings is called `contains` in Java). However, a research challenge is how to decide when a method replacement should be applied in weakly typed dynamic languages such as JavaScript. Recall the example of `x.includes("a")`, where `x` could be anything. A substring pattern-matching (i.e., `x` is a string) is completely different in semantics than an element comparison in a container (e.g., `x` is an array, and elements in the array are not restricted to the same type). Furthermore, `x` could be a custom object defined in the business logic of SUT with a method called `includes`, and not a pre-existing type of the JavaScript APIs (and so we cannot prepare pre-defined heuristics for those custom objects). The type of `x` is not known at instrumentation time, which poses the challenge of how to determine if `x.includes("a")` should be subject to a testability transformation.

The solution here that we present in this paper is that we instrument ALL method calls in the SUT. For example, the previous `if` statement would be rewritten into:

```
if(__EM__.callTracked("test.js", 42, 0, x, "includes", "a"))
```

Such method replacement `__EM__.callTracked` would take as input several parameters. The first three are the name of the JavaScript file where the transformation is applied, together with the line number and index of the call in that line (in case more than one function call is made on the same line). This is needed to create unique *ids* for the two new testing targets that are created. Then, as the semantics of the SUT must not be changed, internally `callTracked` is still going to execute the method `includes` on the object `x` with input `"a"`, and return its output. However, internally, the type of `x` is analyzed at runtime, and, if we have defined any method replacement for the function `x["includes"]`, then we compute the custom heuristics for the two new testing targets.

A further optimization that can be done here is a form of basic *taint analysis*, which in [35] we called *input tracking*. Assume for example that `x` is a string, which is given as input in the test case (e.g., a field in a JSON body payload of a `POST` request). And assume that such `x` is not modified before used in an `if` statement. This is actually quite a common occurrence in Web APIs, as often inputs are validated before being used (e.g., checking if strings are matching some regular expression). Then, although the fitness function with the branch distance and our heuristics in the method replacement would provide gradient to solve those constraints, it would be much more efficient to use the needed values directly instead of mutating random strings until the right value is evolved after a few generations. For example, consider this condition:

```
if(x === "Hello World!")
```

[2]Note that an `array` in JavaScript could have properties, e.g., `x = [ 1, 2, bar: 3 ]`, but the property (such as `bar:3`) is not counted by `includes` operator, e.g., `x.includes(3)` and `x.includes('bar')` would be evaluated as `false`. Then, we employ the '`of`' operator to iterate over every value of an `array` for the heuristic calculations.

The branch distance defined in Section 3.3 would give a gradient to evolve such an input x. However, it would be more efficient to use "Hello World!" directly as the value for x in the test case (as long as x is not modified before the execution of that if statement). In [35], we applied this simple technique, by using tainted values for strings in the generated test cases in the form of evomaster_k_input (where k is a number to distinguish between the different tracked inputs). Every time in a method replacement (or string comparison) we detect that an input is tainted (e.g., by checking the regular expression evomaster_\d+_input), the search gets feedback, and the genotype of the evolving test gets the tainted input evomaster_k_input with the actual needed value (e.g., "Hello World!" in this case). Although quite easy to implement, such basic technique can significantly improve coverage [35] when testing RESTful APIs, especially when dealing with strings that should match a given regular expression (as then the search can directly sample valid strings for it). In this paper, we implemented the same technique for JavaScript.

3.6 Array/Object Access

In JavaScript, object is a datatype that could store a collection of data (e.g., numbers, strings and functions) as properties, and array is considered as a special object representing an ordered collection. Unlike strongly typed languages (e.g., Java), the properties/fields of object in JavaScript are *mutable*. For example, you can remove a property named bar with delete data["bar"], and add a property with data[{}] = "bar" (or replace it if the key {} is already an existing field). In addition, the properties can be accessed dynamically using a variable within square brackets notation (i.e., "[]"), as the example shown next.

```
const y = data[x];
data[z] = 42;
```

Writing data["foo"] is equivalent to writing data.foo. In other words, the square brackets "[]" have different meaning based on what they are applied on (e.g., generic objects vs. arrays) and what they take as input (e.g., numeric indices or not).

As array and object are the main data structures to manipulate data in JavaScript, whether the variable accesses an existing property would often result in different flows of execution (e.g., y with undefined or not). To enable better code coverage, we provide a gradient for such an access to objects at runtime, by replacing "[]" accesses with a pre-defined function, i.e., squareBrackets. However, in JavaScript, the access with square brackets operator (identified as MemberFunction) is not only applied for getting the value but also for assignment. Calculating such a gradient for the assignment is arguably not meaningful, e.g., the assignment could be used for adding a new property. Therefore, at instrumentation, we replace square brackets operators that are used to get values with squareBrackets, and skip cases performing assignments, such as *update expression*, left hand side of *assignment expression*, and left hand side of *assignment pattern*. For example, the previous statements with the squareBrackets would be rewritten into:

```
const y = __EM__.squareBrackets("test.ts", 1, 0, data, x);
data[z] = 42; // skip replacement of square brackets operator for assignments
```

In the squareBrackets function shown in the example, the first three arguments refer to descriptive information for locating the square brackets operator, i.e., the script name ("test.ts"), a line number (1), and an index of a target on the same line (0). The information is needed to define a unique *id* for two testing targets to optimize, i.e., accessing (true) or not accessing (false) for any existing property. The fourth argument is the caller (e.g., data variable), and the last is the given key (e.g., x variable). The function would return the accessed value or throw an error if the caller is null or undefined, to maintain the same code semantics. To provide the gradient for the targets, a heuristic h for the accessed target with the given key is computed as:

- $h = 1$ if the key exists in the caller using '`in`' operator;
- $h = h_c(k, K)$ if the key does not exists, we employ the heuristic for collection by comparing the key with all existing keys as Equation (3).

Note that in JavaScript, the given key could be any type (e.g., `object`), and `null` and `undefined` are also allowed as the key. Thus, we enable the above gradients only if the type of the given key is either `string` or `number`. In addition, the property name is coerced to a string in JavaScript. For instance with an object `data = {42:0}`, the value 0 could be accessed with either `data[42]` or `data['42']`. Therefore, when calculating the distance between the given key and the existing property, the given key would be coerced to a string if it is a number.

Besides the gradients, to be more efficient to cover the accessed target, any existing key could be directly used as a value for `x`. Then, all existing properties are considered as potential candidates with *taint analysis* (Section 3.5).

3.7 Ternary Operator

In JavaScript, a ternary operator (also known as conditional operator) takes three operands for deciding which expression to execute with a condition. Its syntax is `test ? consequent : alternate`. `consequent` would be executed only if the `test` is *truthy* , otherwise `alternate` is executed. An example is shown below:

```
const y = x < 42 ? foo(x): bar(x);
```

To better guide the search for covering all of the three operands, we explicitly handle the three operands at instrumentation and enable four testing targets with gradients. For `test`, it could be handled as two branch targets using branch distance (Section 3.3) for covering *truthy* and *falsy*. For `consequent` and `alternate`, to further represent their executions, we identify them as statement targets and handle their heuristics as $h = 0.5$ for any exception thrown and $h = 1$ for statement completion. To enable the heuristics, we instrument ALL `consequent` and `alternate` with a predefined function `ternary`. For example, the previous statement with the ternary operator would be rewritten into:

```
const y = __EM__.cmp(x, "<", 42, "test.ts", 1, 0)
  ? __EM__.ternary(() => __EM__.callBase(() => foo(x)), "test.ts", 1, 1)
  : __EM__.ternary(() => __EM__.callBase(() => bar(x)), "test.ts", 1, 2);
```

4 EMPIRICAL STUDY

In this paper, we conducted an empirical study to answer the following research questions:

RQ1: Are our white-box SBST heuristics effective at guiding the search for maximizing code coverage and fault finding?

RQ2: How does the search budget impact the achieved results?

RQ3: Compared with the selected baseline black-box techniques, how does our novel approach fare?

4.1 Case Studies

In this empirical study, we employed eight Web APIs running on NodeJS as case studies. Their detailed descriptive statistics are reported in Table 1, where we specify the programming language in which these SUTs are written (i.e., either JavaScript or TypeScript), their type (i.e., REST or GraphQL), the number of script files (i.e., files with suffix either `.js` or `.ts`), **lines of codes (LOCs)**, the number of endpoints, and type of database (if any is used in the SUT). For the REST APIs, the number of endpoints includes all the different actions (i.e., HTTP verbs like `GET` and `POST`) for each

Table 1. Descriptive Statistics of the Eight Web APIs used in our Experiments

SUT	Language	Type	#Files	LOCs	#Endpoints	Database
cyclotron	JavaScript	REST	25	5803	50	MongoDB
disease-sh-api	JavaScript	REST	57	3343	34	Redis
ecommerce-server	TypeScript	GraphQL	48	1791	17	Posrgres
react-finland	TypeScript	GraphQL	461	16206	13	–
realworld-api	TypeScript	REST	37	1229	12	MySQL
js-rest-ncs	JavaScript	REST	8	775	6	–
js-rest-scs	JavaScript	REST	13	1046	11	–
spacex-api	JavaScript	REST	63	4966	94	MongoDB
Total			712	35159	237	

resource identified with a URL. For the GraphQL APIs, this represents the number of `Query` and `Mutation` operations in their schema (i.e., all the entry-points for the API).

Two of the case studies (i.e., *js-rest-ncs* and *js-rest-scs*) are artificial APIs from an existing benchmark [8], used in our previous work when evaluating EvoMaster [30, 31, 94]. They are based on numerical and string-based functions previously used in the literature of unit test generation [26, 32]. We simply re-implemented them in JavaScript, and put them behind a RESTful API (with one distinct endpoint for each different function, where inputs are passed as URL path parameters). The other six APIs were selected from GitHub (a popular open-source repository). We use the APIs of GitHub to search for REST and GraphQL APIs running on NodeJS, sorted by number of *stars* (which is one indicator for the popularity of these open-source projects). It was a long process, as not all projects have clear documentation, and we were not able to run many projects due to several technical issues (e.g., due to broken third-party dependencies). Also, we excluded projects that were clearly too trivial. We stopped the process of finding suitable APIs once we had found enough for an adequate empirical study (e.g., compared to other work in the literature on the subject). Also, due to the high cost of running experiments for system test generation, using a larger case study would require a significant amount of time.

Regarding the case studies, *js-rest-ncs* and *js-rest-scs* were previously implemented with Java, based on code that was designed for studying unit testing approaches on solving numerical [32] and string [26] problems. For this study, we re-implemented them with JavaScript. *cyclotron* [3] is an application (with currently 1.6k stars on the GitHub) for creating dashboards using a REST API that interacts with MongoDB [13] for persisting data. *disease-sh-api* [4] (currently with 2.4k stars on GitHub) provides a set of APIs to get detailed statistics of various viruses (e.g., infected cases with a specified country, and status of vaccine), especially for COVID-19. The statistics information is collected from various online sources (e.g., Worldometers, The New York Times), updated periodically by a separate service, and then stored in Redis [18]. In the context of RESTful API testing, this separated service is not part of the testing. Thus, to test its REST API for retrieving data, we initialized a fixed data sample into the Redis database. *realworld-api* [14] is a NestJS application (2.0k stars on GitHub) that follows a real-world API specification [17] which has been implemented by over 100 different solutions. The application also connects with a MySQL database built with TypeORM/Prisma. *spacex-api* [22] is an unofficial API (9.3k stars on GitHub) to retrieve information for launch, rocket, core, capsule, starlink, launchpad, and landing pad data for the company SpaceX. It uses a MongoDB database. *react-finland* [16] is a GraphQL backend (52 stars on GitHub) for the TOSKA conference in Finland about React (a web frontend framework made by Facebook). Finally, *ecommerce-server* [5] is a GraphQL backend (127 stars on GitHub) for an e-commerce service. It uses a Postgres database.

4.2 Experiment Setup

For this paper, our experiments are divided into three groups, with different settings, as explained in more detail in the remainder of this section. Each experiment was repeated 30 times to take into account the randomness of the used algorithms [33]. All settings of the experiments were executed on an HP Z6 G4 Workstation with the following specs: processor Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz; RAM: 192G; Operating System 64-bit Windows 10.

Group A (for RQ1 and RQ2). To study the effectiveness of our white-box SBST heuristics, we integrated them into EvoMaster, then conducted a comparison between MIO [29] (i.e., the default evolutionary algorithm with SBST heuristics) and the Random-Search algorithm (without SBST heuristics, denoted as Random). This latter can be considered as a grey-box testing technique (as the SUT gets instrumented, and it still keeps track of which targets are covered, like line coverage, when outputting the final test suites at the end of the search). For both techniques, experiments were run on all the eight Web APIs. Regarding the search budget settings, we used $100k$ HTTP calls as the stopping criterion. Using the number of HTTP calls as stopping criterion makes the experiments easier to replicate, as they are independent of the employed hardware. Furthermore, they are less dependent on implementation details that might negatively impact computational performance. In addition, it is expected that a naive random search with no fitness computation would be faster than a complex evolutionary algorithm. Given the same amount of time, the former would likely make more HTTP calls than the latter. Technically, the choice of using $100k$ as stopping criterion is arbitrary: could had been higher, or lower. It was chosen here for consistency, as it is the same value we used in all our previous work on testing RESTful APIs with EvoMaster [35, 77, 90, 94]. Originally, the value $100k$ was chosen because it is a round number, and roughly translates to 1 hour of computation time on average (although this strongly depends on the selected SUTs and on the hardware used for experimentation). The amount of 1 hour could represent a reasonable amount of time practitioners would use a tool like EvoMaster in practice, without making the experiments too prohibitive to run in feasible time. For the given SUTs used in this study, on the given hardware, running the search for 100k HTTP calls takes on average around 4 minutes for *js-rest-ncs* and *js-rest-scs*, 13-20 minutes for *react-finland*, *disease-sh-api*, and *cyclotron*, 1 hour for *realworld-app*, 2 hours for *ecommerce-server* and 3 hours for *spacex-api*. So, around 50 minutes on average across these eight APIs. This also means that this set of experiments with two compared configurations would take roughly $2 \times (50m \times 8) * 30 = 16.6$ days if run in sequence.

Regarding the evaluation metrics, we selected covered testing targets (#Targets), line coverage (%Lines), and the number of detected faults. The testing target (#Targets) is the default coverage criterion in EvoMaster. It comprises and aggregates all metrics used in EvoMaster, such as code coverage, HTTP status code coverage, and fault finding. For the experiment evaluations, these metrics can be used only by the techniques which integrate with our code instrumentation, i.e., MIO and Random in this group of experiments. Currently, the line coverage collected by EvoMaster only applies to the generated tests, and not on the bootstrapping of the SUT (i.e., when it starts, before any test is executed). This means that the reported line coverage is currently an under-estimation (i.e., a lower-bound). This will be fixed in future versions of EvoMaster (we realized this only after all the experiments were completed). Regarding the found faults, this is based on 500 status codes and mismatches of the received responses with the API schema.

Group B (for RQ3). In the context of white-box testing, to the best of our knowledge, there does not exist any other available test generation approach for Web APIs running on NodeJS. Therefore, no comparisons can be made there. Still, it is important to put these novel results in context with the current state of the art in fuzzing Web APIs. For this goal, we chose a set of state-of-the-art open-source black-box techniques as baseline techniques in our experiments. Based on a recent empirical study on existing black-box testing generation tools for REST APIs [46], RESTler [20, 40] and RestTestGen [21, 86] were evaluated as the most robust and the most effective black-box techniques, respectively. We used their most recent versions at the time of this writing. For RESTler, this was version 8.5.0. For RestTestGen, we used the version from its most recent replication package [47, 48]. Besides these two tools, we also selected the black-box technique provided by EvoMaster [31], which was not evaluated in [46]. Note that, for GraphQL APIs, we are aware of no other existing fuzzer that is available. This means that this group of experiments was run only on the six RESTful APIs. All of the selected techniques were applied on the selected case studies using their default settings.

Each of the three tools was run for one hour on each SUT (and this was repeated 30 times), as stopping criterion. In total, this needed $3 * 6 * 30 * 1h = 22.5$ days of computational effort. To avoid bias in the results, the test code coverage of the black-box techniques are collected based on their whole execution process using an external JavaScript instrumentation tool, i.e., C8 [2]. For instance, line coverage results are based on all executed HTTP calls during the whole search, and not just by the generated tests. As the black-box tools have no information on the achieved coverage, the output tests would be based only on black-box metrics (e.g., coverage of different HTTP status code for each endpoint). As hundreds of thousands (if not millions) of HTTP calls can be made during a fuzzing process, the output test suites are usually minimized. Minimizing without code coverage information might lead to the loss of important test cases, and makes tool comparisons more difficult. However, this also means that, if a tool crashes without making any HTTP call, C8 would still report the coverage achieved during the bootstrapping of the SUT.

For this group of experiments, we report only code coverage with C8, and no information on detected faults. The reason is that *there is no clear, easy, and unbiased way to detect faults for each tool*. Each tool can self-report the number of faults it finds, but how those are calculated might vary greatly from tool to tool, making any comparison likely meaningless. Some heuristics could be derived, e.g., based on the analysis of the logs of the SUTs, but we found it unnecessary for answering the research questions in this paper. Furthermore, not all SUTs output exception info into logs when exceptions are thrown during the test executions. Checking for executed endpoints returning HTTP status 500 is not simple either, as each tool generates output test cases in different formats. Trying to compute such values by analyzing network traffic (e.g., using a tool like Wireshark to capture all HTTP traffic) could be doable. However, there would be challenges when using loopback addresses and when running experiments in parallel on the same machine. But, even in such a case, this would represent only one type of fault that this kind of fuzzer can find [77]. Faults related to the schema (e.g., a response that is not matching the expected structure given in the API schema) or to the expected behavior of the API (e.g., it should not be possible to retrieve a resource X with a `GET` after a successful `DELETE` operation on it) would be harder to compute and collect in the same way for each compared tool.

An alternative approach would be to use *mutation testing* [67], but we are aware of no tools that support it for system testing of Web APIs. Creating different versions of the API with only one known, manually seeded fault per version is possible, although it would require quite an engineering effort. Whether faults are found depends on whether there exist any faults. And there are many faults in these kinds of APIs on GitHub [77], as they are not particularly robust to wrong inputs (which often results in a 500 crash instead of returning a more proper 4xx "user error" response).

Regardless, there is a positive correlation between code coverage and fault detection. For example, a fault cannot be detected if its source in the code is never executed. Even if it is a possibility that a tool with lower code coverage might achieve higher fault detection, techniques can be combined. For example, if a tool has a better handling and wider scope of automated oracles to detect different kinds of faults, improving its code coverage results would likely be further beneficial.

Group C (for RQ3). To compare our novel white-box techniques with the current state of the art in black-box fuzzing of Web APIs, we run MIO for 1 hour on each SUT, i.e., the same duration as for the black-box experiments. Based on the previous discussions, comparing the line coverage reported by EvoMaster with the one reported by C8 would be quite pointless. To compare with the black-box techniques, we need to use the same code coverage tool. However, we cannot use C8 directly when the SUT starts, as our novel instrumentation changes the number of lines in the files of the SUT. To make fair comparisons, for white-box EvoMaster we measure the coverage of the final test suite outputted by EvoMaster at the end of the search. This test suite is run with C8 without any instrumentation by EvoMaster. We also make sure to filter out from the coverage reports any extra files, like the tests themselves and the EvoMaster driver.

One issue we faced is with the SUT *realworld-api*, which is written in TypeScript. When running experiments with black-box tools, C8 uses the `.ts` files to report code coverage (note NodeJS runs JavaScript, the mapping from JavaScript to source TypeScript files is done via `.js.map` files outputted by the TypeScript compiler). For some reason (possibly a bug in C8), when running the tests with C8, those map files are ignored, and coverage is measured on the JavaScript files. C8 does not seem to provide any option to setup/configure this automated handling of TypeScript. Although the number of lines between a source `foo.ts` file and its transpiled version `foo.js` would be different (the latter is usually bigger/longer), the ratio of covered lines could likely be similar. It is likely that this is not a major issue, but still needs to be kept in mind when analyzing the results.

Experiment Cost. Running experiments with system testing on Web APIs is computationally expensive. In our case, our experiment setup for the three groups takes roughly $16.6 + 22.5 + 7.5 = 46.6$ days of computation, if run in sequence.

Even when running experiments in parallel, it can still take some days. A further complication is that, regardless of the number of cores the employed machines have, there are limits on how many experiments can be run in parallel on the same machine. For example, each experiment not only requires to run a process instance of EvoMaster and one of the SUT, but also possibly a database via Docker, and these can take a significant amount of RAM, with possible bottlenecks on I/O driver buffers if too many experiments are run in parallel (this latter issue can be partially addressed by mapping the filesystem accessed by the databases directly into RAM). Furthermore, most Web APIs do communications via TCP. Web servers can close down connections on errors, or after a certain number of HTTP requests (e.g., to mitigate DoS attacks). Continuously opening and closing TCP connections can be problematic, as operating systems can take up to 2 minutes before the resources (e.g., a local port) of a closed connection can be re-used for a new connection. If too many experiments are run in parallel, it can happen that the operating systems can run out of local ephemeral ports needed to establish new TCP connections, messing up with the results of the experiments (e.g., EvoMaster would fail to make calls to the APIs if it cannot establish a new TCP connection because there is no local port available). For the JVM, we have addressed this problem using *testability transformations* [35] (in particular, "Testability Transformations for TCP Performance"), by forcing the web servers to never shut down an open TCP connection established with EvoMaster (more details can be found in [35]). But this is not something we do yet for NodeJS, as it requires a non-trivial amount of work.

One consequence of the high cost of running this kind of experiment is that we have not evaluated each different technique presented in this paper in isolation, one at a time.Some techniques might have more impacts than others on the final results. Still, for each single technique, to make sure that it works as intended, we have created artificial examples. On such artificial examples, it

```
const bodyParser = require("body-parser");
const express = require("express");

const app = express();
app.use(bodyParser.json());

app.get("/constantEqual", (req, res) => {

    if(req.query["value"] === "Hello world!!! Even if this is a long string, it
    will be trivial to cover with taint analysis"){
        res.status(200);
        res.json("OK_hello")
    } else {
        res.status(400);
        res.json("FAILED")
    }
});
```

Fig. 2. API example, used to verify the taint analysis technique presented in Section 3.5.

Table 2. Average #Target and Pairwise Comparison between MIO and Random using #Targets

SUT	MIO	Random	$\hat{A}_{12}$	*p*-value	relative
cyclotron	**947.9**	919.5	**0.93**	**≤0.001**	**3.09%**
disease-sh-api	**1247.1**	1109.4	**1.00**	**≤0.001**	**12.41%**
ecommerce-server	**359.2**	340.5	**1.00**	**≤0.001**	**5.49%**
js-rest-ncs	**839.9**	582.3	**1.00**	**≤0.001**	**44.25%**
js-rest-scs	**743.3**	509.8	**1.00**	**≤0.001**	**45.80%**
react-finland	**528.1**	294.2	**1.00**	**≤0.001**	**79.52%**
realworld-app	**737.3**	711.1	**1.00**	**≤0.001**	**3.68%**
spacex-api	1285.1	**1291.6**	0.09	≤0.001	−0.50%

Values in bold or red mean that there is a statistically significant difference, i.e., $p\text{-}value < 0.05$ and $\hat{A}_{12} \neq 0.5$. Values in bold are for the cases in which MIO gets better results than Random, while values in red are for the cases in which MIO gets worse results than Random.

should be trivial for EvoMaster to solve them with the new techniques presented in this paper. So trivial, that we can use these artificial examples as end-to-end tests to verify the correctness of EvoMaster itself.

Figure 2 shows one such example we used. Here, to cover the branch returning status code 200, a query parameter named `value` needs to be provided, matching a specific string. Such a string is long, and unlikely to be matched by sampling strings at random. Branch distance computations defined on string comparisons can provide gradient for the search to evolve the right matching string via mutation operators. However, considering the length of the string, it can take many generations in the evolutionary process. On the other hand, the use of *taint analysis* (recall Section 3.5) makes this trivial to cover in just a couple of iterations.

4.3 Experiment Results

4.3.1 Results for RQ1. To answer RQ1, Table 2 reports the average #Targets achieved by MIO and Random for each of the case studies. Results show that MIO consistently achieves the best performance on average #Target for all the case studies, but one (i.e., *spacex-api*). However, in this

Table 3. Average %Lines and Pairwise Comparison between MIO and Random using %Lines

SUT	MIO	Random	$\hat{A}_{12}$	p-value	relative
cyclotron	**31.1**	30.7	**0.89**	**≤0.001**	**1.42%**
disease-sh-api	**19.6**	18.2	**1.00**	**≤0.001**	**7.49%**
ecommerce-server	8.1	**8.2**	0.14	≤0.001	−2.23%
js-rest-ncs	**82.6**	52.8	**1.00**	**≤0.001**	**56.25%**
js-rest-scs	**62.2**	47.7	**1.00**	**≤0.001**	**30.31%**
react-finland	**1.8**	0.9	**1.00**	**≤0.001**	**88.91%**
realworld-app	24.6	**24.6**	0.22	≤0.001	-0.30%
spacex-api	41.8	**42.0**	0.20	≤0.001	-0.39%

Values in bold or red mean that there is a statistically significant difference, i.e., $p\text{-}value < 0.05$ and $\hat{A}_{12} \neq 0.5$. Values in bold are for the cases in which MIO gets better results than Random, while values in red are for the cases in which MIO gets worse results than Random.

Table 4. Average #Faults and Pairwise Comparison between MIO and Random using #Faults

SUT	MIO	Random	$\hat{A}_{12}$	p-value	relative
cyclotron	**34.0**	29.5	**1.00**	**≤0.001**	**15.37%**
disease-sh-api	**34.2**	29.0	**1.00**	**≤0.001**	**18.18%**
ecommerce-server	**30.3**	27.9	**0.97**	**≤0.001**	**8.59%**
js-rest-ncs	**5.0**	**5.0**	0.50	1.000	0.00%
js-rest-scs	**4.0**	**4.0**	0.50	1.000	0.00%
react-finland	**30.5**	24.5	**1.00**	**≤0.001**	**24.66%**
realworld-app	**35.4**	24.0	**1.00**	**≤0.001**	**47.64%**
spacex-api	51.7	**54.3**	0.03	≤0.001	−4.79%

Values in bold or red mean that there is a statistically significant difference, i.e., $p\text{-}value < 0.05$ and $\hat{A}_{12} \neq 0.5$. Values in bold are for the cases in which MIO gets better results than Random, while values in red are for the cases in which MIO gets worse results than Random.

case the difference is minimal (just six targets). In addition, based on an analysis of the pairwise comparisons using Mann-Whitney-Wilcoxon U-tests (*p-value*) and Vargha-Delaney effect sizes ($\hat{A}_{12}$), MIO significantly outperforms Random, with a high effect size (i.e., $\hat{A}_{12} \geq 0.93$) and a low *p*-value (i.e., $p \leq 0.001$). Regarding the relative improvement, MIO achieves the most on *react-finland* (i.e., +79.52%) and the least on *cyclotron* (i.e., +3.09%).

To get better insight in these results, we report the average line coverage in Table 3 (however, recall these values do not include the coverage achieved by just bootstrapping the SUT before running any test). For *js-rest-ncs* and *js-rest-scs*, we have a large improvement of MIO over Random (up to nearly +30%). Those are SUTs with many numeric and string constraints, which become easy to solve with the SBST techniques we introduced in this paper for JavaScript applications. However, relatively, the largest improvement is for *react-finland*. It might seems that coverage is low on this SUT, but this is a very peculiar SUT where most of its code is executed at bootstrapping (the SUT does not use a database, and all the info of the conference is stored in JavaScript files that are executed when the API starts). Running those generated tests in an IDE demonstrates more than 80% code coverage. For the other SUTs, there is not much difference in achieved code coverage (up to 1.5% difference). An analysis of these SUTs and the generated test cases shows this being mainly due to constraints based on interactions with databases. This is an important research topic that will need to be addressed to achieve better results.

In Table 4, we report the number of potential faults identified by MIO, which are due to 5xx status code and unexpected responses based on the API schemas. After a manual analysis of those found faults, most of those are related to *invalid* input handling, i.e., lack of checks or constraint specifications on the parameters of the requests. There also exist some faults due to improper service configuration. For instance, there is a potential fault (with 500 status) detected by the following generated test for testing GET request on `/ldap/search`. By debugging the SUT with the test, we found out that the problem can refer to an empty check on a configuration file of the service. Thus, the test would help set up a proper configuration for the SUT.

```
test("test_7_with500", async () => {
    await superagent.get(baseUrlOfSut + "/ldap/search?q=LHmM6EfXJDcSiyf").set('
    Accept', "*/*") // src/middleware/cors.js_32_10;
  });
```

On average, 225 faults were found in these eight APIs. This is consistent with previous studies on APIs running on the JVM [77], where it was shown that often these kinds of APIs do not properly do input validation, which leads to crashes (resulting in HTTP 500 status code).

Based on the above analysis, we can conclude that:

RQ1*: In terms of target coverage, our white-box approach demonstrates consistent and significant improvements (up to relatively 79.52%) compared with the grey-box random testing, for 7 out of the 8 case studies. On average, 225 faults were found in these APIs.*

4.3.2 Results for RQ2. In Figure 3, we show plot-lines for demonstrating the performance of MIO and Random in detail with the number of covered targets (including code coverage and fault detection) over the course of the search. This data was collected at each 5% interval of the search (i.e., at every 5k HTTP calls). As shown in the figures, for most of the SUTs, MIO consistently outperforms Random by a clear large margin throughout the whole search process. This further shows that our white-box SBST heuristics provide an effective guide to the search for maximizing code coverage and fault finding.

One exception is *spacex-api*, where the performance of MIO is lower than Random by the end of the search. However, for the first 35% of the search, MIO gives better results. This can be explained if our heuristics provide gradients for some of the constraints in the SUT (so better results for MIO), but those constraints are not too complex. Therefore, given enough time, even a naive random search does catch up. After this point, if there is no gradient in the fitness landscape for MIO to follow, the evolutionary search would stagnate in specific areas of the search landscape, i.e., local optima. On the other hand, Random would keep sampling tests from the whole search space, which might get to cover new targets by chance outside those local optima.

To shed light on this peculiar behavior, we have run more experiments and in-depth analyses of the source-code of *spacex-api*, to try to understand the culprit of this phenomenon. Two issues seem at play here: relations between `POST-GET` operations on a resource, and lack of gradient on the queries to the database. For example, to successfully test the `GET` on endpoint `/v4/launches/next`, a `POST` operation on `/v4/launches` needs to be executed to create such resource. Such resource needs to be created with some specific values (e.g., some booleans set to `true`), as the `GET` operation does a database query searching for entries using those values. However, our heuristics on resource-operation relations do not handle such cases [95], so test cases with this `POST-GET` sequence get sampled less often by MIO in contrast to random search. As there is no gradient in the `GET` operation when dealing with the database query, MIO is actually rewarded by *removing* the `POST` operation from the test case, as the test becomes shorter while keeping the same fitness

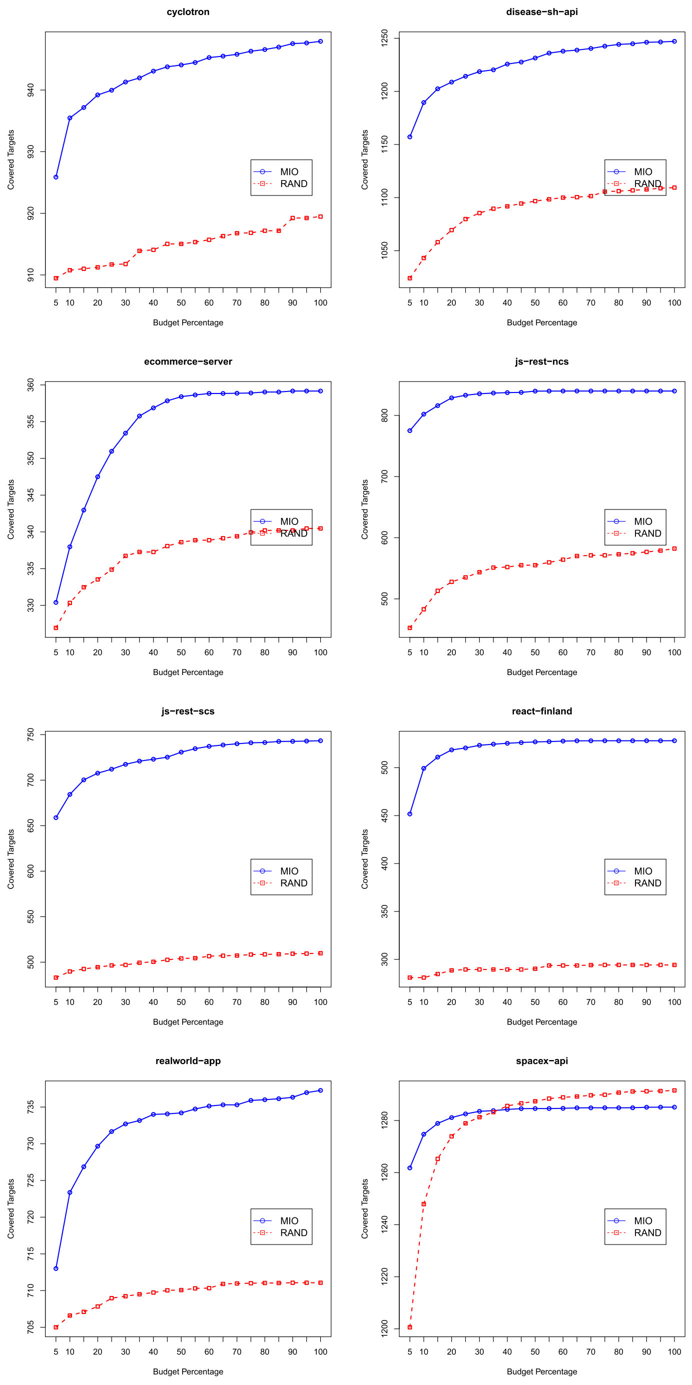

Fig. 3. Average covered targets (y-axis) of MIO and Random throughout the search, reported at 5% intervals of the used budget allocated for the search (x-axis).

value regarding the coverage of the `GET`. This problem would be easily fixed by adopting here the database handling techniques presented in [34].

This phenomenon that we see in *spacex-api* could in theory happen in any of the other SUTs as well, given a bigger search budget (i.e., more than 100k HTTP calls). Also, for *spacex-api*, we would have drawn different conclusions if we used a smaller search budget like 10k. In other words, the choice of search budget does play a major role when comparing search algorithms (and fuzzers in general).

Ideally, the choice of the search budget should be based on how practitioners would use these fuzzers in practice. This likely would be based on time, e.g., 10 minutes, 2 hours or 1 day. However it is unlikely that a practitioner would run these fuzzers for long periods of time, e.g., more than a week. For example, it would make little sense to run a fuzzer for 5 years, as, by the time the test cases are generated, the tested version of the SUT would be long deprecated. Therefore, even if random search could eventually achieve the same coverage or higher, it does not mean it can do it within a viable amount of time.

At the time of this writing, in our documentation we currently recommend practitioners that download EvoMaster to run it somewhere between 1 and 24 hours per API. But this is just an educated guess, as how much time is needed depends on the API. The output of EvoMaster also provides some information to help deciding on how long to run EvoMaster after the first time. From our current documentation [37]:

> "For how long should the search be left running? The info `Needed budget` helps with this question, as it keeps track of the last time there was an improvement in the search (e.g., a new line is covered, or a new fault is discovered). If the `Search budget` (i.e., for how long the search is left running) is 100 minutes, and the `Needed budget` is 53%, then it means that the last improvement was achieved after 53 minutes, with the last 47 minutes of the search not finding any useful new test data. High values (e.g., > = 90%) likely mean that, if you leave EvoMaster running for longer, likely you are going to get better results. But what values should you aim for `Needed budget`? Hard to say for sure, as it depends on the tested API and how/where/when you run EvoMaster (e.g., if using a dedicated CI machine running long fuzzing jobs during the night, then you might not care much about `Needed budget`). But, if you run EvoMaster for at least 1 hour, and `Needed budget` is lower than 50% (e.g., 42%), then it is unlikely that you will get significantly better results by increasing the search budget."

The more complex the SUT is, the longer it will take to answer to HTTP requests (especially if the SUT accesses external services like databases and third-party Web APIs). Looking at Figure 3, this means that, if we run EvoMaster only for 1 hour, it would result in around $33k$ HTTP calls for *spacex-api* (as $100k$ take roughly 3 hours), which would lead to better results for MIO compared to Random (which is what we are going to show in the next set of analyses).

RQ2*: Our novel SBST heuristics provide significant improvement throughout the whole search, for the given search budget.*

4.3.3 Results for RQ3. Table 5 shows the average line coverage achieved by each tool on each SUT. Table 6 shows the pairwise comparison results between white-box EvoMaster (i.e., using MIO) and the three other black-box baselines. Recall that these experiments were run for 1 hour per tool (repeated 30 times), and only including RESTful APIs.

MIO gives the best results on four out of six SUTs. The worse results for *realworld-app* might be due to the bug in C8 we discussed in Section 4.2. The case for *cyclotron* is rather interesting.

Table 5. Average Line Coverage of the Four Compared Tools, on Each SUT

SUT	EvoMaster BB	RESTler	RestTestGen	EvoMaster WB
cyclotron	**70.1 (1)**	41.3 (3.5)	41.3 (3.5)	66.7 (2)
disease-sh-api	60.7 (2)	48.4 (3.5)	48.4 (3.5)	**61.9 (1)**
js-rest-ncs	92.7 (2)	44.3 (3.5)	44.3 (3.5)	**99.0 (1)**
js-rest-scs	82.3 (2)	54.1 (3.5)	54.1 (3.5)	**89.0 (1)**
realworld-app	**67.1 (1)**	65.1 (2)	59.7 (4)	63.8 (3)
spacex-api	84.8 (2)	76.3 (3)	76.1 (4)	**93.7 (1)**
Average	76.3 (1.7)	54.9 (3.2)	54.0 (3.7)	79.0 (1.5)

BB stands for black-box, whereas WB stands for white-box. The best coverage results have rank 1, and are displayed in bold.

Table 6. Pairwise Comparison on Each SUT of White-box (WB) EvoMaster with the other Three Tools, including Black-box (BB) EvoMaster

SUT	Tool	$\hat{A}_{12}$	p-value	relative
cyclotron	EvoMaster BB	0.00	$\leq$0.001	−4.73%
	RESTler	**1.00**	**$\leq$0.001**	**61.63%**
	RestTestGen	**1.00**	**$\leq$0.001**	**61.63%**
disease-sh-api	EvoMaster BB	**1.00**	**$\leq$0.001**	**1.86%**
	RESTler	**1.00**	**$\leq$0.001**	**27.88%**
	RestTestGen	**1.00**	**$\leq$0.001**	**27.88%**
js-rest-ncs	EvoMaster BB	**1.00**	**$\leq$0.001**	**6.72%**
	RESTler	**1.00**	**$\leq$0.001**	**123.53%**
	RestTestGen	**1.00**	**$\leq$0.001**	**123.53%**
js-rest-scs	EvoMaster BB	**1.00**	**$\leq$0.001**	**8.11%**
	RESTler	**1.00**	**$\leq$0.001**	**64.38%**
	RestTestGen	**1.00**	**$\leq$0.001**	**64.38%**
realworld-app	EvoMaster BB	0.23	$\leq$0.001	-4.89%
	RESTler	0.20	$\leq$0.001	-2.00%
	RestTestGen	**1.00**	**$\leq$0.001**	**6.90%**
spacex-api	EvoMaster BB	**1.00**	**$\leq$0.001**	**10.49%**
	RESTler	**1.00**	**$\leq$0.001**	**22.85%**
	RestTestGen	**1.00**	**$\leq$0.001**	**23.16%**

White-box MIO is better than RESTler and RestTestGen, but not black-box EvoMaster. For RESTler and RestTestGen, we found that they crash when parsing the schema of this case study, thus the reported coverage is based only on the bootstrapping of the SUT. Regarding *cyclotron*, this REST API is mainly for interacting with a MongoDB database,[3] such as searching and filtering data. Thus, the amount of available data in the database has a direct impact on the code coverage. In the black-box testing, the data in the database is accumulated over the course of the search, each time a fuzzer makes a call to add new data. For white-box testing (WB EvoMaster), the data in the database is cleared at every test execution (to make the tests be independent, which is a requirement for using generated tests for regression testing). But for WB EvoMaster with JavaScript, there are limited capabilities to manipulate states of the database (e.g., SQL heuristics are not enabled), and the states of the database could be only manipulated with the API endpoints. Since the data is cleared at every test execution, such manipulation might limit

[3] https://github.com/ExpediaGroup/cyclotron#rest-api.

the amount of available data, due to the length of test cases (currently up to 10 HTTP calls per test). This might be a reason for the underperformance of WB EvoMaster on this case study. Furthermore, there are additional challenges to address for NoSQL databases such as MongoDB used in *cyclotron*.

Another interesting case is *spacex-api*, where in these experiments, when running white-box EvoMaster for 1 hour we get better comparison results (Table 6) than in the comparisons when running for 100k HTTP calls (Table 3). As discussed in Section 4.3.2, this can be explained by looking at the plots in Figure 3: 1 hour would roughly mean around 33k HTTP calls for white-box EvoMaster, which is before the point in time in which a random search becomes more effective.

***RQ3**: In most cases, white-box testing of NodeJS Web APIs provides significantly higher code coverage than existing black-box fuzzers.*

4.3.4 Discussion. Based on our experiments, our white-box technique shows its advantage over the black-box techniques, especially in code coverage on the case studies which have many branches to be optimized involving numeric and string constraints (e.g., *js-rest-ncs* and *js-rest-scs*). Regarding the fault detection, most of the found faults are related to improper input validation, before the business logic of the SUTs is executed. Therefore, even a naive random search can detect this kind of fault. This is consistent with previous fault analysis on fuzzing RESTful APIs running on the JVM [77]. However, with code instrumentation, the white-box technique can provide the potential location of the bug, based on the last executed statement in the business logic. This can help to distinguish among different bugs when there exist multiple bugs in the same endpoint resulting to a 500 HTTP server error status code.

Regarding the database handling, with black-box testing techniques, there is a lack of control of the database. It means that we cannot prevent the side-effects of previously executed tests, e.g., the 100,000th request is executed with a state of a SUT which depends on the executed 99,999 previous requests. This is a major issue for debugging: if a test case reveals a fault, re-executing such test case might not reveal it again, if it depends on the state modified by all previous tests. Making each test case independent is essential for debugging purposes (e.g., as we do in our white-box technique when in EvoMaster the configuration drivers are implemented by the user [31]). Furthermore, when we have a small optimized test suite that can automatically start/stop/reset the SUT, such generated tests can be used for regression testing (e.g., added to the repository of the SUT, and run automatically as part of a Continuous Integration server).

In our white-box approach, once we prepare the driver scripts to start and stop the SUT, we also reset the state of the database. We provide library support for all the databases used in the SUTs of our empirical study (Table 1), which are MongoDB, MySQL, Postgres, and Redis. This enables EvoMaster to generate test cases that are independent from each other.

Although line coverage achieved in each SUT is more than 60% (Table 5), a manual analysis of what is not covered clearly points to issues with handling databases. In this kind of application, control-flow conditions in the code of the SUT often rely on the state of the database, and on the results of queries on it. Without any heuristics, it might be hard to generate input data for which, for example, the `WHERE` clause of a SQL query is satisfied. EvoMaster has support for handling heuristics on SQL databases for the JVM [34]. These techniques could be extended for NodeJS as well, although there are research and technical challenges when dealing with the dynamic nature of JavaScript and the fact that there is no standard way for accessing SQL databases in NodeJS (e.g., on the JVM, all drivers for the different SQL databases implement the specs of `java.sql`, and all Java libraries for accessing databases, like Hibernates and JOOQ, use those specs internally). EvoMaster would need special handling for each different driver for each different kind of SQL database. These are non-trivial challenges to overcome, although overcoming them is necessary to address when dealing with this kind of web/enterprise application running on NodeJS. Furthermore, we are aware of no technique in the literature dealing with NoSQL databases like Redis and MongoDB in the context of fuzzing Web APIs.

5 THREATS TO VALIDITY

Conclusion validity. Our experiments are in the context of search-based software engineering, and the search algorithms have a stochastic nature. To take into account such stochastic behaviors, we repeated our experiments 30 times following the guidelines from [33]. Then, we performed further statistical analyses on the results with these 30 runs. To demonstrate the effectiveness of our approach, we employed widely used metrics in the context of testing, i.e., line coverage and the number of detected faults with their average values. In the analysis of the comparisons with the baseline techniques, we applied statistical methods and reported the statistical results in detail, i.e., the Mann-Whitney U-test (p-value) at significance level $\alpha = 0.05$ for pairwise difference analysis, along with the Vargha-Delaney ($\hat{A}_{12}$) effect size. Besides, we computed relative improvement for further illustrating the improvement over the baseline techniques.

Construct validity. First, for all of the employed techniques, we used their default settings, which typically are the best configurations of the technique chosen by their authors. All experiments are conducted on the same machine to prevent side-effects, related to hardware and operating system, on the techniques and SUTs. The metrics we used such as line coverage are not direct outputs of the techniques. To properly collect such information and make them comparable, we employed the same external tool (i.e., c8) for accessing coverage reports achieved by all the different techniques. Furthermore, for black-box techniques, there might lack some code coverage with their outputs (i.e., executable output tests which are composed of a sequence of HTTP requests). Therefore, the code coverage for them are collected with all requests executed during the whole process. Comparing different tools is always a challenge. Ideally, one would want to compare techniques and not actual tools, as the performance of these latter can be strongly affected by accessible configurations and low level code implementation details. Also, tools can have bugs. For example, in some cases both RESTler and RestTestGen crash due to inability to parse the given OpenAPI schemas, or they fail to derive valid URLs to connect to the APIs (e.g., by looking at their logs, it seems they have some problems dealing with `basePath` entries declared as `'/'`). In theory, it could be possible to find APIs for which EvoMaster crashes whereas RESTler and RestTestGen work fine. Therefore, an unbiased selection of SUTs would be preferable [54]. But this is not trivial to do for Web APIs. For this reason, all APIs that we have been using for evaluating EvoMaster throughout the years (since 2017) are collected in a single open-source repository on GitHub, currently called EMB [8]. Developers of other fuzzers can use these APIs, and make sure their tools do not crash when fuzzing them.

Internal validity. This is related to the threat on our implementation used in the experiments. Although we carefully tested our implementation, we cannot guarantee that it is free of bugs. However, our implementation and case studies are open-source and available online, which allows anyone to review them and replicate the experiments.

External validity. It is related to the threat on how our results can generalize to other case studies. In this study, we conducted our experiments with eight NodeJS Web APIs, both REST and GraphQL ones. Two of them are artificial that we re-implemented with JavaScript, whereas the other six are open-source from GitHub. It is a fact that REST APIs are widely used in industry, but fewer of them are open-source and available in open-source repositories. GraphQL APIs are even less easy to find among open-source projects. Furthermore, experiments on systems testing are expensive to run. This is a problem limiting how many APIs can be used for experimentation.

6 CONCLUSIONS

JavaScript is one of the most widely used programming languages. However, testing applications written in JavaScript is quite challenging due to its dynamic nature, especially for white-box testing. To the best of our knowledge, there does not exist any other white-box system-level test generation for JavaScript Web APIs. In this paper, we enable code instrumentation for JavaScript, and further integrate it into an open-source white-box testing tool, i.e., EvoMaster. Besides, we propose an automated approach to calculate search-based heuristics like the common *Branch Distance* that enables system test generation for JavaScript applications using Search-Based Software Testing (SBST) techniques. To evaluate our approach, we conducted an empirical experiment on the automated *system testing* of RESTful and GraphQL APIs. In the experiments, we compared our approach with four baseline techniques (i.e., one grey-box testing technique and three black-box testing techniques) on eight NodeJS Web APIs, in terms of code coverage and fault finding. Results show that our technique leads to significantly better results than existing black-box and grey-box testing tools.

For future work, to improve code coverage results, it will be important to extend our JavaScript instrumentation to handle and analyze all interactions with SQL (e.g., Postgres and MySQL) and NoSQL (e.g., MongoDB and Redis) databases.

To enable replicated studies, and to support new research effort on the use of SBST techniques for JavaScript programs, our Babel plugin for JavaScript instrumentation, and our extension to the EvoMaster tool together with all the used case studies, are released as open-source on GitHub, with each new release automatically archived on Zenodo (e.g., [38, 39]). For more information, see `www.evomaster.org`.

REFERENCES

[1] (n. d.). Babel. https://babeljs.io/. Online, Accessed May 20, 2022.
[2] (n. d.). C8. https://github.com/bcoe/c8. Online, Accessed May 20, 2022.
[3] (n. d.). cyclotron. https://github.com/ExpediaInceCommercePlatform/cyclotron. Online, Accessed May 20, 2022.
[4] (n. d.). disease-sh-api. https://github.com/disease-sh/API. Online, Accessed May 20, 2022.
[5] (n. d.). E-Commerce Server. https://github.com/react-shop/react-ecommerce. Online, Accessed May 20, 2022.
[6] (n. d.). ECMAScript Specification. https://www.ecma-international.org/ecma-262/. Online, Accessed May 20, 2022.
[7] (n. d.). Electron. https://www.electronjs.org/. Online, Accessed May 20, 2022.
[8] (n. d.). EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Online, Accessed May 20, 2022.
[9] (n. d.). Ionic. https://ionicframework.com/. Online, Accessed May 20, 2022.
[10] (n. d.). JEDI. https://github.com/aelyasov/JEDI. Online, Accessed May 20, 2022.
[11] (n. d.). Jest. https://jestjs.io/. Online, Accessed May 20, 2022.
[12] (n. d.). JUnit. http://junit.sourceforge.net/. Online, Accessed May 20, 2022.
[13] (n. d.). MongoDB. https://www.mongodb.com/. Online, Accessed May 20, 2022.
[14] (n. d.). nestjs-realworld-example-app. https://github.com/lujakob/nestjs-realworld-example-app. Online, Accessed May 20, 2022.
[15] (n. d.). NodeJS. https://nodejs.org/. Online, Accessed May 20, 2022.
[16] (n. d.). React-Finland. https://github.com/ReactFinland/graphql-api. Online, Accessed May 20, 2022.
[17] (n. d.). realworld API Specification. https://github.com/gothinkster/realworld. Online, Accessed May 20, 2022.
[18] (n. d.). Redis. https://redis.io/. Online, Accessed May 20, 2022.
[19] (n. d.). RestAssured. https://github.com/rest-assured/rest-assured. Online, Accessed May 20, 2022.
[20] (n. d.). restler-fuzzer. https://github.com/microsoft/restler-fuzzer. Online, Accessed May 20, 2022.
[21] (n. d.). RESTTESTGEN. https://github.com/resttestgenicst2020/submission_icst2020. Online, Accessed May 20, 2022.
[22] (n. d.). SpaceX-API. https://github.com/r-spacex/SpaceX-API. Online, Accessed May 20, 2022.
[23] (n. d.). The State of the OCTOVERSE. https://octoverse.github.com/.
[24] (n. d.). SuperAgent. https://visionmedia.github.io/superagent/. Online, Accessed May 20, 2022.

[25] 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.

[26] 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

[27] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–36.

[28] Andrea Arcuri. 2018. EvoMaster: Evolutionary multi-context automated system test generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[29] Andrea Arcuri. 2018. Test suite generation with the many independent objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.

[30] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.

[31] Andrea Arcuri. 2020. Automated black-and white-box testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72–78.

[32] A. Arcuri and L. Briand. 2011. Adaptive random testing: An illusion of effectiveness?. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 265–275.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Andrea Arcuri, ZhangMan, asmab89, Bogdan, Amid Golmohammadi, Juan Pablo Galeotti, Seran, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2022. EMResearch/EvoMaster:. (June 2022). https://doi.org/10.5281/zenodo.6651631

[38] Andrea Arcuri, ZhangMan, Bogdan, asmab89, Amid Golmohammadi, Juan Pablo Galeotti, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2022. EMResearch/EvoMaster:. (Feb. 2022). https://doi.org/10.5281/zenodo.6106776

[39] Andrea Arcuri, ZhangMan, Amid Golmohammadi, and asmab89. 2022. EMResearch/EMB:. (Feb. 2022). https://doi.org/10.5281/zenodo.6106830

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

[41] 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.

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

[43] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In *Genetic and Evolutionary Computation Conference (GECCO)*.

[44] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. White-Box and Black-Box Fuzzing for GraphQL APIs. (2022). https://doi.org/10.48550/ARXIV.2209.05833

[45] 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

[46] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful APIs. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 226–236.

[47] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2021. Replication Package: Automated Black-Box Testing of Nominal and Error Scenarios in RESTful APIs. (Dec. 2021). https://doi.org/10.5281/zenodo.5803118

[48] 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.

[49] Domenico Cotroneo, Antonio Ken Iannillo, and Roberto Natella. 2019. Evolutionary fuzzing of Android OS vendor system services. *Empirical Software Engineering* 24 (2019), 3630–3658.

[50] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.

[51] Hamza Ed-Douibi, Javier Luis CÃąnovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. 181–190.

[52] Alexander Elyasov, I. S. W. B. Prasetya, and Jurriaan Hage. 2018. Search-based test data generation for JavaScript functions that interact with the DOM. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 88–99.

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

[54] G. Fraser and A. Arcuri. 2012. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 178–188.

[55] 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.

[56] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[57] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 474–484.

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

[59] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *ACM Symposium on the Foundations of Software Engineering (FSE) (ESEC/FSE 2020)*. ACM, 725–736.

[60] 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

[61] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

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

[63] 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

[64] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. 2002. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO)*. 1351–1358.

[65] M. Harman and B. F. Jones. 2001. Search-based software engineering. *Journal of Information & Software Technology* 43, 14 (2001), 833–839.

[66] 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.

[67] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)* 37, 5 (2011), 649–678.

[68] 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.

[69] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[70] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2013. AUSTIN: An open source tool for search based software testing of C programs. *Information and Software Technology* 55, 1 (2013), 112–125.

[71] 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

[72] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.

[73] 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.

[74] Riyadh Mahmood, Jay Pennington, Danny Tsang, Tan Tran, and Andrea Bogle. 2022. A framework for automated API fuzzing at enterprise scale. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 377–388.

[75] Valentin J. M. 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.
[76] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 94–105.
[77] 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.
[78] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing*.
[79] P. McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
[80] 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
[81] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279
[82] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. 1–14.
[83] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.
[84] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
[85] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–37.
[86] 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.
[87] 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
[88] J. Wegener, A. Baresel, and H. Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.
[89] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. *The Fuzzing Book*. (2019).
[90] 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).
[91] Man Zhang and Andrea Arcuri. 2022. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. (2022). https://doi.org/10.48550/ARXIV.2205.05325
[92] 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 (TOSEM)* (Feb. 2023). https://doi.org/10.1145/3585009 Just Accepted.
[93] 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.
[94] 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.
[95] 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.
[96] 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

Received 20 May 2022; revised 13 January 2023; accepted 15 March 2023