JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs

Man Zhang
Kristiania University College
Oslo, Norway
ORCID 0000-0003-1204-9322

Asma Belhadi
Kristiania University College
Oslo, Norway
ORCID 0000-0002-7103-2179

Andrea Arcuri
Kristiania University College and OsloMet
Oslo, Norway
ORCID 0000-0003-0799-2930

***Abstract*—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 techniques. We provide an automated approach to collect search-based heuristics like the common *Branch Distance*. 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 APIs. Experiments on 5 NodeJS APIs 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.**

***Index Terms*—JavaScript Instrumentation, NodeJS, white-box test generation, SBST**

I. Introduction

As of 2020, according to the official statistics of GitHub [1] (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 [2], as well as desktop applications (e.g., using Electron [3]) and mobile apps (e.g., using Ionic [4]).

JavaScript (more formally, ECMAScript [5]) is a *weakly* and *dynamically* typed language. These language properties differentiate JavaScript from other popular programming languages, such as Java, C# and C/C++. Unfortunately, the lack of *strong typing* significantly complicates static and dynamic analyses [6], including automated test generation.

In this paper, we show how Search-Based Software Testing (SBST) [7]–[10] techniques can be used for JavaScript applications. We describe how common techniques in the literature like the *Branch Distance* [11]–[13] 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 [14] (which is a popular tool for JavaScript code transformations).

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).

Our instrumentation would enable the use of SBST techniques in different testing contexts, like for example *unit testing*. To empirically evaluate the effectiveness of our JavaScript instrumentation, we integrated it into the EvoMaster [15] tool, which does automated *system test* generation for RESTful APIs running on the JVM (e.g., compiled from programming languages such as Java, Kotlin and Scala). EvoMaster uses evolutionary algorithms like MIO [16], based on fitness functions that exploit SBST heuristics.

The empirical study was conducted on 5 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 is used), and three black-box approaches: the one provided by EvoMaster, RESTler [17] and RESTTESTGEN [18]. 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. In particular for code coverage, compared with the black-box techniques, the relative improvements are up to +521.80% (increase from 15.1% to 92.4%) of line coverage and +1842.22% (increase from 4.3% to 83.8%) of branch coverage.

This paper provides the following contributions to the state-of-the-art: 1) A full working approach to enable SBST for JavaScript applications; 2) A novel algorithmic approach to improve the branch distance computations in JavaScript applications; 3) An empirical study on the testing of RESTful APIs showing the improvement of our approach compared to three state-of-the-art tools; and 4) An open-source implementation with replication package[1] for our novel techniques.

II. Related Work

SBST [7]–[10] techniques have been successfully used for more than three decades, in several different testing contexts. Popular SBST examples are EvoSuite [19] for unit testing, and Sapienz [20] 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

[1] https://github.com/anonyreview/ICST2022-JS-Instrumentation-Test

of white-box testing, a popular technique introduced in the 90s by Korel [11] 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 [12] (e.g., `AND` and `OR`) and string comparisons [13].

To the best of our knowledge, the only existing SBST work that deal with JavaScript instrumentation is the JEDI tool [21], which aims at unit testing of JavaScript code that interacts with the DOM in the browser. However, such work 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 [21], but its repository [22] 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, code needs to be manipulated, but still we need to make sure that the semantics of the SUT is 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. However, in the past, in the literature of SBST there has been work on instrumentation of source code, in particular for the C programming language [23], [24]. 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.* [6]. However, no technique has been proposed that aimed at system test generation for web services (e.g., RESTful APIs) written in JavaScript. Most of the work has been on unit testing (e.g., the more recent [25], [26]). One of the main challenges here is that the entry point of the SUTs are TCP sockets, and this kind of applications often use 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.

As far as we know, EvoMaster [15] is currently the only tool that does white-box system test generation for RESTful 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., [17], [18], [27]–[30]), and so would not be able to exploit any source code instrumentation.

III. JavaScript Instrumentation

A. Tool Support

Although JavaScript code is not compiled, it is typical to apply transformations (referred as *transpilation*) to the 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 [14], 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 a 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.

Instrumentation itself is not enough though, as it needs to be integrated with a SBST test generation tool. For our experiments in this paper, we employed EvoMaster, which generates system-level test cases for RESTful 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 functionality 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 [31] test suite files, in either Java or Kotlin, using the library RestAssured [32] for making the HTTP calls. We simply implemented a further option to output the test cases in JavaScript, as Jest [33] test suite files, using the library SuperAgent [34] for making the HTTP calls.

B. Code Coverage

A JavaScript program will be composed of a series of source files, each one having code lines with code statements. Each of which will become a testing target, which 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 has *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 function takes 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 as currently done 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 cover 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 out 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 [35]. 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 [16] keep on sampling and mutating test cases for targets that are not fully covered (e.g., $h_s = 0.5$), whereas for their 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 [16]). Note though that a value like $h_s = 0.5$ does not really give gradient to find input data that lead to no exception. It is used to mainly tell the search to still 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, like 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.

C. Branch Distance

The control flow of the SUT can depend on complex predicates, e.g., conditions in `if` statements. The branch distance [11] 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 evaluate as `true`, and one for when it is evaluates 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 [36].

For each of these new targets, an heuristics $h \in [0, 1]$ 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` does return `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). So, the function `cmp` can be called with any type of inputs for `A` and `B`, and their type 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 [11] is used to compute the h heuristics. If they are both of type `string`, then we use the string distances defined in [13]. 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.

A major challenge here is how to deal with `AND/OR` operators. In the 90s, the branch distance for those operators was defined [12] 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). But 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, will 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 [5], 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:

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

it will be replaced by:

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

Writing something like `() => foo.bar()` declares a new function with no inputs, which, once called, it 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 to 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()` due to space constraints, and due to the fact that it is quite similar to `or()`. At any rate, as our EvoMaster extension is released open-source, it is available online.

There are a few problems that need to be taken care of here. First, there is the need to be able to access to the branch

```
1  or(left: () => any, right: () => any, 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) {
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()`

distance (if any) of the `left` and `right` operands. This is done by 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 a 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. Otherwise, unless `right` is a *pure* function (i.e., with no side-effects), 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).

The computation of the combination of the heuristics for `left` and `right` (line 31) is inspired by the Equation 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., $\neg(A \vee B) = \neg A \wedge \neg B$. So 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 as 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 anyway we 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.

IV. 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 findings?
- **RQ2**: How does our novel approach perform in terms of code coverage and fault detection?

TABLE I: Experiment design

(a) descriptive statistics of the case studies

SUT	#JS	LOCs	#Endpoints	Database	MTB(1)
cyclotron	25	5803	50	MongoDB	7.52m
disease-sh-api	57	3343	34	Redis	15.15m
realworld-api	37	1229	12	MySQL	66.07m
rest-ncs	8	775	6	-	3.97m
rest-scs	13	1046	11	-	3.80m
total	103	10987	101		

(b) experiment settings for the selected techniques

Context	Techniques	Budget	Metrics
White-Box	JS-MIO	100*k*	#Targets (**only for WB**)
	JS-Random	100*k*	%Lines(2)
Black-Box	BB-EM, BB_f-EM	100*k*	%Branches(2)
	BB_t-EM	MTB(1)	#Faults
	RESTler (v8.0.0)	$2 \times$ MTB	
	RESTESTGEN	30m	

(1) MTB is the maximum time spent by white-box techniques, and *m* presents minutes; (2) for white-box testing techniques, %Lines and %Branches are line coverage and branch coverage collected by the final generated output test files, whereas, for black-box testing techniques, %Lines and %Branches are line coverage and branch coverage collected during the whole search process.

RQ3: Compared with the selected baseline black-box techniques, does our novel approach achieve any improvement? If yes, how much is such improvement?

A. Experiment Setup

In this empirical study, we employed 5 NodeJS REST APIs as case studies. Their detailed descriptive statistics with the number of scripts, lines of codes and the number of endpoints are reported in Table Ia. Two of the case studies (i.e., *rest-ncs* and *rest-scs*) are artificial APIs from an existing benchmark [37], used in our previous work when evaluating EvoMaster [35], [38], [39]. The other three are real REST APIs selected from GitHub (a popular open-source repository).

Regarding the case studies, *rest-ncs* and *rest-scs* were previously implemented with Java, based on code that was designed for studying unit testing approaches on solving numerical [40] and string [41] problems. For this study, we re-implemented them with JavaScript. *cyclotron* [42] is an application (with currently 1.5*k* stars on the GitHub) for creating dashboards using a REST API that interacts with MongoDB [43] for persisting data. *disease-sh-api* [44] (currently with 2.2*k* 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 [45]. 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[1] into the Redis database. *realworld-api* [46] is a NestJS application (1.7*k* stars on the GitHub) that follows a real-world API specification [47] which has been implemented by over 100 different solutions. The application also connects with a MySQL database built with TypeORM/Prisma.

Table Ib shows the set of the selected techniques used in our experiments. To study the effectiveness of our white-box SBST heuristics, we integrated them into EvoMaster, then conducted a comparison between MIO [16] (i.e., the default evolutionary algorithm with SBST heuristics, denoted as JS-MIO) and the Random-Search algorithm (without SBST heuristics, denoted as JS-Random) which can be considered as grey-box testing (as it still keeps track of which targets are covered, like line coverage, when outputting the final test suites at the end of the search). Moreover, in the context of white-box testing, to the best of our knowledge, there does not exist any other available test generation approach for NodeJS REST APIs.

To further investigate our approach, 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 [48], RESTler [17], [49] and RESTTESTGEN [18], [50] were evaluated as the most robust and the most effective black-box technique, respectively, for REST APIs. Besides, we also selected a black-box technique provided by EvoMaster [38] (denoted as BB), which was not evaluated in [48].

All of the selected techniques were applied on the 5 selected case studies using their default settings. Experiments were repeated 30 times to take into account the randomness of these algorithms [51]. Regarding the search budget settings, we set 100*k* HTTP calls for the white-box testing approaches, i.e., MIO (JS-MIO) and Random (JS-Random). For black-box EvoMaster, we also set the same number of HTTP calls (BB_f). However, black-box EvoMaster typically performs faster than white-box EvoMaster (as it does not collect coverage information after each test execution). Therefore, we also considered a further configuration (BB_t) where we employed the maximum time spent by the white-box EvoMaster (MTB) as a time budget for black-box EvoMaster (as stopping criterion, EvoMaster allows to choose either time or number of HTTP calls). For RESTler, it only allows time budget. However, as RESTler seems much slower than EvoMaster (i.e., number of HTTP calls per second), to avoid possible bias due to implementation details and choice of programming language (i.e., F# and Python vs. Kotlin), for the experiments we used twice the time budget for RESTler (i.e., $2 \times MTB$). For RESTTESTGEN, the time budget is not configurable with the current available version online [50]. Therefore, the default budget (i.e., 30 minutes as stated in [18]) is applied in these experiments. Note the the MTB of each case study is shown in Table Ia.

Regarding the evaluation metrics, we selected covered testing targets (#Targets), line coverage (%Lines), branch coverage (%Branches) and the number of detected faults. The testing target (#Target) is the default coverage criterion in EvoMaster. It comprises and aggregates different metrics, such as code coverage, status code coverage and fault findings. For the experiment evaluations, #Targets can be used only by the techniques which integrate with our code instrumentation, i.e. JS-MIO and JS-Random. To collect %Lines and

TABLE II: Average #Target and pairwise comparison between JS-MIO and JS-Random using #Targets.

SUT	JS-MIO	JS-Random	$\hat{A}_{12}$	p-value	relative
cyclotron	**730.7**	719.8	**0.91**	**≤0.001**	**1.51%**
disease-sh-api	**917.3**	831.6	**1.00**	**≤0.001**	**10.30%**
realworld-app	**606.6**	583.7	**1.00**	**≤0.001**	**3.92%**
rest-ncs	**787.6**	549.1	**1.00**	**≤0.001**	**43.45%**
rest-scs	**632.4**	472.3	**1.00**	**≤0.001**	**33.91%**

Value in **bold** means that the JS-MIO is significantly better than Random, i.e., $p\text{-}value < 0.05$ and $\hat{A}_{12} > 0.5$.

%Branches, for white-box testing, we executed the output test files (i.e., at the end of the search) against the SUTs. For black-box testing, tests are typically generated by considering the endpoint coverage (i.e., different achieved HTTP status codes for each different endpoint in the API). Considering code coverage metrics (such as %Lines and %Branches) are not employed for their test generation, it might be not fair to only use the generated test files to assess their coverage performance. 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 [52]. For instance, results of %Lines by BB_f are the line coverage achieved by all executed HTTP calls during the search, i.e., the whole 100*k* HTTP calls. To assess the performance in fault detection, we also reported potential faults for all of the selected techniques. In our context, the faults are defined based on the HTTP status codes (i.e., 500) and unexpected responses (based on the schema declarations).

All settings of the experiments (in Table Ib) were executed on an HP Z6 G4 Workstation with a specification, i.e., Processor: Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz; RAM: 192 GM; Operating System: 64-bit Windows 10.

B. Experiment Results

1) Results for RQ1: To answer RQ1, Table II reports the average #Targets achieved by JS-MIO and JS-Random for each of the case studies. Results show that JS-MIO consistently achieves the best performance on average #Target for all the case studies. In addition, based on an analysis of the pairwise comparisons using Mann-Whitney-Wilcoxon U-tests ($p\text{-}value$) and Vargha-Delaney effect sizes ($\hat{A}_{12}$), JS-MIO significantly outperforms JS-Random, with a high effect size (i.e., $\hat{A}_{12} \geq 0.91$) and a low p-value (i.e., $p \leq 0.001$). Regarding the relative improvement, JS-MIO achieves the most on *rest-ncs* (i.e., +43.45%) and the least on *cyclotron* (i.e., +1.51%).

Moreover, we analyze performances of JS-MIO and JS-Random in detail with plot-lines based on the number of covered targets over the course of the search. For all of the SUTs, JS-MIO consistently outperforms JS-Random by a clear large margin throughout the whole process of the search which further shows that our white-box SBST heuristics provide an effective guide to the search for maximizing code coverage and fault finding. Note that, due to space limitation, detailed plot-lines can be found in our replicate package[1].

> ***RQ1****: In terms of target coverage, our white-box approach demonstrates a consistent and significant improvements (up to 43.45%) compared with the grey-box random testing, for all of the five case studies. This shows the effectiveness of our approach to guide the SBST for maximizing the code coverage and fault finding.*

2) Results for RQ2: To answer RQ2, we report average line coverage and branch coverage by JS-MIO in Table III that were collected by executing the generated output tests on the SUT with c8 in Table III (note that our instrumentation is used only during the search; the generated output test files do not use it, nor need it).

Based on the coverage results, for the artificial case studies, JS-MIO achieves a high code coverage for both %Lines and %Branches (i.e., %Lines ∈ [75.8%, 94.2%] and %Branches ∈ [80.9%, 83.9%]) with a low time cost (i.e., less than 4 minutes as MTB reported in Table Ia).

Regarding the real SUTs, for *cyclotron*, JS-MIO enables of covering 49.1% of lines and 22.6% of branches. By further checking the implementation, we found that there are four script files which were never executed, i.e., *api.analytics-elasticsearch.js*, *api.analytics.js*, *api.statistics-elasticsearch.js*, and *api.statistics.js*. All of the scripts are related to analytics endpoints that are needed to be enabled by manually editing a script file, i.e., *config.js*, and the default configuration is `False`. Since the four scripts take 24.9% of the total code lines of the SUT, it might be the reason for the relatively lower code coverage compared with the other SUTs. However, with tests generated automatically within just 8 minutes, 49.1% line coverage could be arguably a reasonable result. For *disease-sh-api*, JS-MIO achieves average 61.8% of lines and 91.5% of branches. Regarding this achieved high branch coverage, we further investigated its source code. In this SUT, all of the endpoints are to retrieve information (i.e., GET). There exist only a few `If(condition)` branches per endpoint, and most of the branches are to check the data from the database. Thus, the data setup in the database might have a major impact on the branch coverage. To test *disease-sh-api*, the data is properly pre-set by taking a subset (around 4.7M with JSON format) of the collected data by the external service. This pre-set may provide the possibility of achieving the high branch coverage. For system test generation, generating data directly into the database as part of the test cases could be an important step to achieve better results, and to reduce manual effort. Although EvoMaster has support for SQL databases [53], we are aware of no work dealing with Redis. For *realworld-app*, JS-MIO achieves a high code coverage on both lines and branches (i.e., 87.0% of lines and 82.2% of branches). As checked, this application is mainly to add, query or modify the data, interacting with the MySQL database. In addition, the application also has well-defined schema which clearly defines the input data and hierarchical resources to be performed on. Such schema would possibly enable a good starting point for the search. Note that compared with other SUTs, this applications took relatively longer time, i.e., 66.07 minutes for 100*k* HTTP actions, as shown in Table Ib. This is mainly due

TABLE III: Average coverage (i.e., %Lines and %Branches) achieved by JS-MIO and the selected baselines with ranks.

SUT	%Lines					%Branches				
	JS-MIO	BB_f	BB_t	RESTler	RESTTESTGEN	JS-MIO	BB_f	BB_t	RESTler	RESTTESTGEN
cyclotron	**49.1% (1)**	48.1% (3)	48.4% (2)	-	-	**22.6% (1)**	21.6% (3)	21.9% (2)	-	-
disease-sh-api	**61.8% (1)**	60.8% (3)	61.0% (2)	60.0% (4)	-	**91.5% (1)**	82.9% (3)	84.1% (2)	55.1% (4)	-
realworld-app	87.0% (3)	87.1% (2)	**87.3% (1)**	81.7% (4)	66.5% (5)	**82.2% (1)**	79.3% (3)	79.6% (2)	65.0% (4)	29.4% (5)
rest-ncs	**94.2% (1)**	62.7% (3)	70.8% (2)	46.2% (4)	15.1% (5)	**83.8% (1)**	54.4% (3)	60.6% (2)	14.4% (4)	4.3% (5)
rest-scs	**75.8% (1)**	64.2% (3)	64.9% (2)	61.1% (4)	32.1% (5)	**80.9% (1)**	37.4% (3)	42.1% (2)	17.2% (4)	8.6% (5)
Average Rank	**1.40**	2.80	1.80	4.00	5.00	**1.00**	3.00	2.00	4.00	5.00
Friedman test					χ^2=18.080, p-value=**0.001**					χ^2=20.000, p-value=≤**0.001**

Rank value 1 indicates the highest achievement, and values in bold are the best among the techniques for a SUT. We also report p-value and χ^2 of the Friedman test based on the ranks, for variance analysis of technique performances on the SUTs.

TABLE IV: Average #Faults achieved by JS-MIO and the selected baselines with ranks.

SUT	#Faults				
	JS-MIO	BB_f	BB_t	RESTler	RESTTESTGEN
cyclotron	**55.83 (1)**	55.07 (3)	55.10 (2)	-	-
disease-sh-api	**34.00 (1)**	**34.00 (1)**	**34.00 (1)**	0.00 (4)	-
realworld-app	**32.83 (1)**	23.00 (2)	23.00 (2)	3.00 (4)	0.00 (5)
rest-ncs	**6.00 (1)**	**6.00 (1)**	**6.00 (1)**	0.00 (4)	0.00 (4)
rest-scs	**13.00 (1)**	**13.00 (1)**	**13.00 (1)**	2.00 (4)	0.00 (5)
Average Rank	**1.00**	1.60	1.40	4.00	4.80
Friedman test					χ^2=18.791, p-value=≤**0.001**

to frequent interactions with the database. For the experiments on *realworld-app*, the average computation time overhead by our approach between tests is 0.35 millisecond.

In Table IV, we report the number of potential faults identified by JS-MIO, which are due to 5xx status code and unexpected responses. After a manual analyses 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 setup a proper configuration for the SUT.

```
1 test("test_7_with500", async () => {
2     await superagent.get(baseUrlOfSut + "/ldap/
      search?q=LHmM6EfXJDcSiyf").set('Accept', "*/*"
      ) // src/middleware/cors.js_32_10;
3   });
```

Based on the above analysis, we can conclude that:

> ***RQ2****: Our novel approach enables the automated generation of tests that can achieve up to 87.0% line coverage for real APIs and up to 94.2% line coverage for artificial APIs, and further detect 141.66 faults in total on these five APIs.*

3) Results for RQ3: To answer RQ3 for a comparison with existing work, we conducted an analysis on the performances in %Lines, %Branches and #Faults of JS-MIO and all selected baseline techniques. Table III shows the average achievements and their ranks, whereas Table V shows the pairwise comparison results between JS-MIO and the baselines. Note that we were not able to collect the results of RESTler for *cyclotron*, and the results of RESTTESTGEN for *cyclotron* and *disease-sh-js* due to failure when applying them on these SUTs. For instance, RESTler fails to parse the schema (of *cyclotron*) which does not fully follow the standard of OpenAPI, e.g., the error is reported due to lack of specifying a type for `Array`, and RESTTESTGEN failed in a process of generating client code with given schemas of the two REST APIs. In addition, RESTTESTGEN is not open source [48], and options we could configure for these experiments are limited. Based on its outputs, we found that some requests get 404 status code due to wrong handling of endpoint URLs, or show timeouts that might lead to side-effects on its results.

Regarding code coverage (%Lines and %Branches), in four out of the five case studies (*cyclotron*, *disease-sh-api*, *rest-ncs*, and *rest-scs*), JS-MIO achieves a strong and significant improvement over all of the black-box techniques. This is demonstrated by (1) the average achievement (see Table III): the best rank (i.e., 1) with the significant variance among the techniques (i.e., p-value < 0.05 with Friedman test); and (2) pairwise comparisons (see Table V): high effect size (i.e., $\hat{A}_{12} \geq 0.82$) and low p-value (i.e., ≤ 0.001).

For *realworld-app*, JS-MIO shows its advantage over RESTler and RESTTESTGEN. But regarding %Lines metric, BB_t performs better than JS-MIO. By further checking its source code and coverage reports of the two techniques, we found that JS-MIO fails to cover an else-condition shown at line 2 in Figure 2a, but BB_t did in some runs. The condition is related to whether a requested *user* exists in the database. If the *user* does not exist (i.e., else-condition), following lines after it cannot be reached. This condition can be considered as a flag, in which there is no straightforward way to provide gradient to the search to find data for which such expression evaluates as true. Then, covering it would typically happen by chance. Comparing the number of executed HTTP calls, BB_t executed on average 284.07k calls, which is near 3 times the number of calls (i.e., 100k) executed by JS-MIO. In addition, based on one BB_t coverage report representing the condition is covered, the if-condition at line 2 is executed 14633 times, and its else-condition is executed only 2 times. Considering that the budget of JS-MIO is 100k, it might explain the failure of covering the condition and the limited performance on the line coverage. However, the difference of %Lines between JS-MIO and BB_t is modest, i.e., -0.30%, and JS-MIO has a significantly better performance of %Branches with +3.31%

TABLE V: Pair comparison between JS-MIO and the selected black-box baselines, using %Lines, %Branches and #Faults

SUT	JS-MIO vs.	%Lines $\hat{A}_{12}$	%Lines p-value	%Lines relative	%Branches $\hat{A}_{12}$	%Branches p-value	%Branches relative	#Faults $\hat{A}_{12}$	#Faults p-value	#Faults relative
cyclotron	BB_f	**0.95**	**≤0.001**	**1.94%**	**0.91**	**≤0.001**	**4.73%**	**0.76**	**≤0.001**	**1.39%**
	BB_t	**0.83**	**≤0.001**	**1.45%**	**0.82**	**≤0.001**	**3.25%**	**0.74**	**≤0.001**	**1.33%**
	RESTler	-	-	-	-	-	-	-	-	-
	RESTTESTGEN	-	-	-	-	-	-	-	-	-
disease-sh-api	BB_f	**1.00**	**≤0.001**	**1.60%**	**1.00**	**≤0.001**	**10.33%**	0.50	NaN	0.00%
	BB_t	**1.00**	**≤0.001**	**1.25%**	**1.00**	**≤0.001**	**8.81%**	0.50	NaN	0.00%
	RESTler	**1.00**	**≤0.001**	**2.97%**	**1.00**	**≤0.001**	**66.14%**	**1.00**	**≤0.001**	**Inf**
	RESTTESTGEN	-	-	-	-	-	-	-	-	-
realworld-app	BB_f	0.39	0.050	-0.15%	**0.96**	**≤0.001**	**3.70%**	**1.00**	**≤0.001**	**42.75%**
	BB_t	0.27	≤0.001	-0.30%	**0.93**	**≤0.001**	**3.31%**	**1.00**	**≤0.001**	**42.75%**
	RESTler	**1.00**	**≤0.001**	**6.53%**	**1.00**	**≤0.001**	**26.46%**	**1.00**	**≤0.001**	**994.44%**
	RESTTESTGEN	**1.00**	**≤0.001**	**30.81%**	**1.00**	**≤0.001**	**179.92%**	**1.00**	**≤0.001**	**Inf**
rest-ncs	BB_f	**1.00**	**≤0.001**	**50.07%**	**1.00**	**≤0.001**	**54.08%**	0.50	NaN	0.00%
	BB_t	**0.99**	**≤0.001**	**32.99%**	**1.00**	**≤0.001**	**38.46%**	0.50	NaN	0.00%
	RESTler	**1.00**	**≤0.001**	**104.03%**	**1.00**	**≤0.001**	**482.67%**	**1.00**	**≤0.001**	**Inf**
	RESTTESTGEN	**1.00**	**≤0.001**	**521.80%**	**1.00**	**≤0.001**	**1842.22%**	**1.00**	**≤0.001**	**Inf**
rest-scs	BB_f	**1.00**	**≤0.001**	**18.08%**	**1.00**	**≤0.001**	**116.28%**	0.50	NaN	0.00%
	BB_t	**1.00**	**≤0.001**	**16.88%**	**1.00**	**≤0.001**	**92.33%**	0.50	NaN	0.00%
	RESTler	**1.00**	**≤0.001**	**24.14%**	**1.00**	**≤0.001**	**370.91%**	**1.00**	**≤0.001**	**550.00%**
	RESTTESTGEN	**1.00**	**≤0.001**	**135.80%**	**1.00**	**≤0.001**	**841.82%**	**1.00**	**≤0.001**	**Inf**

The analysis is conducted with Mann-Whitney-Wilcoxon U-tests (p-$value$), Vargha-Delaney effect sizes ($\hat{A}_{12}$) and relative improvement $\frac{a-b}{a}$. Values in **bold** mean that JS-MIO is statistically significantly better than the baseline, whereas values in red means the baseline is statistically significantly better than JS-MIO. Note that for p-value, NaN means that two samples are the same; for relative improvement $\frac{a-b}{b}$, **Inf** means that b is 0.

```
1  const _profile = await this.userRepository.findOne(
       {username: followingUsername});
2  if(!_profile) return;
3  let profile: ProfileData = {
4    username: _profile.username,
5    bio: _profile.bio,
6    image: _profile.image
7  };
```

(a) lines 31-39 of `profile.service.ts`

```
1  if (followingUser.id === followerId) {
2    throw new HttpException('FollowerId and
       FollowingId cannot be equal.', HttpStatus.
       BAD_REQUEST);
3  }
```

(b) lines 88-90 of `profile.service.ts`

Fig. 2: Snippet codes of *realworld-app-js*

relative improvement (see Table V). The improvement on %Branches could demonstrate the effectiveness of JS-MIO for solving conditions which have gradients, as an example shown in Figure 2b which can be covered by JS-MIO sometimes but never for BB_t.

Regarding the performance in fault detection, Table III shows the average number of identified faults by each of the techniques, whereas Table V shows the pair comparison results between JS-MIO and the baselines. Note that, in EvoMaster, faults are identified with 5xx status code and unexpected responses, similarly to RESTler and RESTTESTGEN. In terms of code coverage, compared with the four baseline techniques, JS-MIO consistently has the most improvement over RESTler and RESTTESTGEN. Based on the results on faults, JS-MIO maintains such significant improvements, i.e., $\hat{A}_{12} = 1.0$ and p-value < 0.001. By comparing with BB_t and BB_f, JS-MIO shows its advantage on *cyclotron* and *realworld-app*, but not on *disease-sh-api*, *rest-ncs* and *rest-scs*. Whether faults are found depends on whether there are faults in the first place, and their complexity. If there are no faults, or faults are simple missing input validations, then it is expected to not get much better results compared to a naive random search. Future work will be needed to study more complex injected faults, e.g., using mutation testing analysis.

Based on the above analysis, we can conclude that:

> ***RQ3**: Compared with the four selected techniques, our white-box SBST approach achieves the overall best performance in code coverage and fault detection, i.e., the relative improvements are up to 521.80% for line coverage, up to 1842.22% for branch coverage and up to 994.44% for fault detection.*

4) Discussion: Based on our experiments, our white-box technique shows its advantage over the black-box techniques in code coverage and fault detection, especially in code coverage on the case studies which have many branches to be optimized involving numeric and string constraints. Regarding the fault detection, most of the found faults are related to improper input validation, before the business logic of the SUTs is executed. So, even a naive random search can detect this kind of faults. 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 [38]). 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).

V. 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 [51]. 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, branch 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 Friedman test (χ^2 and p-value) for variance analysis of techniques on the different case studies, and 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 %Lines and %Branches 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 configuration and low level code implementation details. An online version of the tools might not be fully configurable, e.g., the version of RESTTESTGEN we used in this study. In addition, for RESTler, from our experiments it would appear that the tool is hundreds of times slower than EvoMaster in terms of HTTP calls made per second, which negatively impacts its results. This could be due to technical details, or it could be the cost of the employed advanced techniques (e.g., [27]) is very high. So, when choosing the search budget (also referred as stopping criterion) for our approach, we set the same or *higher* (e.g., twice as much for RESTler) budget for the baseline techniques to prevent bias in the analysis of their outputs [10].

Internal validity. This is related to the threat on our implementation used in the experiments. We have tested our implementation with 491 test suites (including unit tests and end-to-end tests) that achieve 63% line coverage reported by codecov [54]. But, 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. The is related to the threat on how our results can generalize to other case studies. In this study, we conducted our experiments with five NodeJS REST APIs, i.e., two of them are artificial that we re-implemented with JavaScript, whereas the other three 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. Furthermore, experiments on systems testing are expensive to run. This is a problem limiting how many case studies can be used for experimentation.

VI. Conclusions

JavaScript is one of most widely used programming language. 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 services. 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 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 five NodeJS REST 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, it will be important to extend our JavaScript instrumentation with Testability Transformations [55]–[57], which will pose new challenges when dealing with a dynamically and weakly typed language such as JavaScript.

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, are released as open-source. See `www.evomaster.org`.

REFERENCES

[1] "The state of the octoverse," https://octoverse.github.com/.
[2] "Nodejs," https://nodejs.org/.
[3] "Electron," https://www.electronjs.org/.
[4] "Ionic," https://ionicframework.com/.
[5] "Ecmascript specification," https://www.ecma-international.org/ecma-262/.
[6] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–36, 2017.
[7] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
[8] M. Harman and B. F. Jones, "Search-based software engineering," *Journal of Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
[9] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.
[10] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 6, pp. 742–762, 2010.
[11] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.
[12] M. J. Gallagher and V. L. Narasimhan, "Adtest: A test data generation suite for ada software systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 23, no. 8, pp. 473–484, 1997.
[13] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification and Reliability (STVR)*, vol. 16, no. 3, pp. 175–203, 2006.
[14] "Babel," https://babeljs.io/.
[15] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.
[16] ——, "Test suite generation with the Many Independent Objective (MIO) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.
[17] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful REST API fuzzing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, ser. ICSE. IEEE, 2019, p. 748–758.
[18] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020.
[19] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.
[20] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 94–105.
[21] A. Elyasov, I. Prasetya, and J. Hage, "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, 2018, pp. 88–99.
[22] "Jedi," https://github.com/aelyasov/JEDI.
[23] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.
[24] K. Lakhotia, M. Harman, and H. Gross, "Austin: An open source tool for search based software testing of c programs," *Information and Software Technology*, vol. 55, no. 1, pp. 112–125, 2013.
[25] M. Selakovic, M. Pradel, R. Karim, and F. Tip, "Test generation for higher-order functions in dynamic languages," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
[26] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic execution for javascript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 2018, pp. 1–14.
[27] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent rest api data fuzzing," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, ser. ESEC/FSE 2020. ACM, 2020, p. 725–736.
[28] S. Karlsson, A. Causevic, and D. Sundmark, "Quickrest: Property-based test generation of openapi described restful apis," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020.
[29] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, "Automatic generation of test cases for rest apis: A specification-based approach," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, 2018, pp. 181–190.
[30] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2020.
[31] "Junit." http://http://junit.sourceforge.net/.
[32] "RestAssured," https://github.com/rest-assured/rest-assured.
[33] "Jest," https://jestjs.io/.
[34] "Superagent," https://visionmedia.github.io/superagent/.
[35] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.
[36] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2002, pp. 1351–1358.
[37] "Evomaster benchmark (emb)," https://github.com/EMResearch/EMB.
[38] A. Arcuri, "Automated black-and white-box testing of restful apis with evomaster," *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.
[39] M. Zhang, B. Marculescu, and A. Arcuri, "Resource-based test case generation for restful web services," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1426–1434.
[40] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 265–275.
[41] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification, and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.
[42] "cyclotron," https://github.com/ExpediaInceCommercePlatform/cyclotron.
[43] "MongoDB," https://www.mongodb.com/.
[44] "disease-sh-api," https://github.com/disease-sh/API.
[45] "Redis," https://redis.io/.
[46] "nestjs-realworld-example-app," https://github.com/lujakob/nestjs-realworld-example-app.
[47] "realworld api specification," https://github.com/gothinkster/realworld.
[48] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Empirical comparison of black-box test case generation tools for restful apis," *arXiv preprint arXiv:2108.08196*, 2021.
[49] "restler-fuzzer," https://github.com/microsoft/restler-fuzzer.
[50] "Resttestgen," https://github.com/resttestgenicst2020/submission_icst2020.
[51] A. Arcuri and L. Briand, "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2014.
[52] "C8," https://github.com/bcoe/c8.
[53] A. Arcuri and J. P. Galeotti, "Handling sql databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.
[54] "codecov," https://about.codecov.io/.
[55] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.
[56] A. Arcuri and J. P. Galeotti, "Testability transformations for existing apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 153–163.
[57] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 440–451.