

.NET/C# Instrumentation for Search-Based Software Testing

Amid Golmohammadi, Man Zhang and Andrea Arcuri

Kristiania University College, Norway.
Oslo Metropolitan University, Norway.

*Corresponding author(s). E-mail(s): man.zhang@kristiania.no;
Contributing authors: amid.golmohammadi@kristiania.no;
andrea.arcuri@kristiania.no;

Abstract

C# is one of the most widely used programming languages. However, to the best of our knowledge, there has been no work in the literature aimed at enabling Search-Based Software Testing techniques for applications running on the .NET Platform, like the ones written in C#. In this paper, we propose a search-based approach and an open source tool to enable white-box testing for C# applications. The approach is integrated with a .NET bytecode instrumentation, in order to collect code coverage at runtime during the search. In addition, by taking advantage of *Branch Distance*, we define heuristics to better guide the search, e.g., how heuristically close it is to cover a branch in the source code. To empirically evaluate our technique, we integrated our tool into the EVOMASTER test generation tool, and conducted experiments on three .NET RESTful APIs as case studies. Results show that our technique significantly outperforms grey-box testing tools in terms of code coverage.

Keywords: .NET Instrumentation, white-box test generation, SBST, RESTful APIs

1 Introduction

C# is one of the most popular programming languages for building standalone and web enterprise systems [1], e.g., cloud-based applications. However, there

are barely any existing technique for automatically generating system tests for C# applications.

Search-based Testing techniques (SBST) have achieved many successful stories in both research [2] and industry [3]. To the best of our knowledge, there does not exist any SBST tool for C# applications. Considering the widespread use of C# in industry, this is a major gap in the research literature.

EVOMASTER [4] is an open-source fuzzer which applies evolutionary algorithms for enabling automated black-box and white-box testing of REST and GraphQL APIs [5, 6]. Regarding the white-box testing, however, it only supports JVM and NodeJs based APIs [7, 8]. The performance of EVOMASTER in the white-box mode has been studied in several empirical studies by comparing with other techniques [8–12]. In this paper, we extend EVOMASTER [4] for enabling the white-box fuzzing of .NET/C# REST APIs (i.e., adopt the search algorithm and fitness function of EVOMASTER with our .NET/C# SBST heuristics). C# is an object-oriented language which can be compiled into Common Intermediate Language (CIL) bytecode instruction set. To deal with white-box testing, we first develop a .NET bytecode instrumentation and add probes for enabling collecting code coverage at runtime. In addition, to be more effective to guide the search, we employ *branch distance* [13] for our white-box SBST heuristics, in particular for numeric and string data types.

To evaluate the effectiveness of our approach, we integrated our bytecode instrumentation and *branch distance* based heuristics into EVOMASTER, named EVOMASTER_{.NET}. Except the configurations needed for enabling C#, we used EVOMASTER with its default settings (e.g., *Resource-based Sampling* [14]). We conducted an experiment by comparing EVOMASTER_{.NET} with a grey-box testing approach on three open-source .NET REST APIs. Two of them are based on numerical (i.e., *cs-rest-ncs*) and string problems (i.e., *cs-rest-scs*), whereas the third one is an API which handles a restaurant’s menu and deals with a Postgres database (i.e., *menu-api*). Results show that our approach achieves a clear and significant improvement over the grey-box testing approach for *cs-rest-ncs* and *cs-rest-scs*, and statistically equivalent results on *menu-api*. With a further investigation on code coverage achieved by the generated tests, we found that our approach is capable of solving most of the numeric and string branches, demonstrated by a high line coverage (i.e., up to 98% for numeric problems and up to 86% for string problems). However, for *menu-api* which deals with a database, no better performance was achieved, as implementing and adapting techniques to handle SQL databases is necessary [15].

At the time of this writing, EVOMASTER has more than 260 stars on GitHub [16], and it has been downloaded more than 1 400 times. No large numbers by any means, but it provides some indications of its actual usage among industrial practitioners. A concrete example is Meituan, a large Chinese e-commerce company with hundreds of millions of customers, where EVOMASTER is currently successfully integrated in their Continuous Integration systems [17]. When presenting EVOMASTER at different industrial venues throughout the years, one of most common questions from practitioners has

been “*does it support C#/.NET*”? This has been the main *industry-driven* [18–22] motivation for the scientific work carried out in this paper. Unfortunately, there is a well known documented gap between academic research and industrial needs, as major software engineering efforts might be required to be able to scale academic prototypes to be applicable to industrial systems. And all the research challenges needed to be addressed to get there might be mistakenly labeled as mere “technical work”. For example, in the last few years there has been many tools presented in the scientific literature for fuzzing RESTful APIs besides EVOMASTER (e.g., [23–28]). However, to the best of our knowledge, they are **all** *black-box*, in which the source code of the tested applications is not analyzed. Building a white-box fuzzer for this problem domain (i.e., for C#/.NET APIs) took months of work with over 10 000 lines of code just for the instrumentation and driver sides. It is far from trivial, which might explain why, albeit EVOMASTER is open-source since 2016, it is still the only white-box fuzzer for Web APIs.

The main contributions in this work include:

- the first use of white-box SBST techniques in the literature for .NET applications;
- a novel bytecode instrumentation with SBST heuristics for .NET, which could be plugged-in by other existing SBST techniques;
- an automated solution with an open-source tool implementation for enabling SBST of .NET applications;
- an empirical study in which we successfully replicated the fuzzing of RESTful APIs with SBST techniques.

The paper is organized as follows. Section 2 provides necessary info to better understand the remaining of the paper. Section 3 discusses existing related work. Our approach for instrumenting C#/.NET applications with SBST heuristics is discussed in Section 4. Our empirical analyses are presented in Section 5. Threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Background

2.1 .NET programming

.NET is a developer platform by Microsoft for building many types of applications. In 2016, Microsoft introduced .NET Core which is open source and cross platform [29]. It is possible to develop programs with .NET in C#, F# or Visual Basic. Programs written in these languages are compiled into Common Intermediate Language (CIL), which is an object oriented and entirely stack-based bytecode instruction set. C# has been the most popular language for developing .NET applications, and currently it is among the most widely used programming languages in the world [1]. Let us consider the following simple method written in C#:

```
1 public void Foo(int a) {
```

4 .NET/C# Instrumentation for Search-Based Software Testing

```

2     if(a>0)
3         Console.WriteLine("a is greater than 0");
4 }

```

Listing 1: example 1 with C#

The code snippet above contains a method which takes as input `a`, which is of type `int`. Then, it prints out a string to the console if `a` is greater than zero. The CIL code for such program is as follows:

```

1 .method public hidebysig instance void
2 Foo(
3 int32 a
4 ) cil managed
5 {
6     .maxstack 2
7     .locals init (
8     [0] bool V_0
9     )
10
11     IL_0000: nop
12
13     IL_0001: ldarg.1          // a
14     IL_0002: ldc.i4.0
15     IL_0003: cgt
16     IL_0005: stloc.0        // V_0
17
18     IL_0006: ldloc.0        // V_0
19     IL_0007: brfalse.s     IL_0014
20
21     IL_0009: ldstr          "a is greater than 0"
22     IL_000e: call          void [System.Console]System.
        Console::WriteLine(string)
23     IL_0013: nop
24
25     IL_0014: ret
26 } // end of method MyClass::Foo

```

Listing 2: CIL bytecode for Listing 1

.NET has a rich set of bytecode instructions [30], which has a few similarities (and some differences) with the bytecode of the Java Virtual Machine (JVM). In this example, `ldarg.1` is pushing the value of the input `a` onto the stack. Then, the constant 0 is pushed with `ldc.i4.0`. The instruction `cgt` pops these two values, and pushes either 0 or 1 based on their comparison. `stloc.0` and `ldloc.0` are only needed for helping debugging (e.g., when putting break points from an IDE), and would not be there when compiling in “release” mode. `brfalse.s IL_0014` is a jump instruction. If the current value on stack is 0 (i.e., `a` is not greater than 0), then the computation jumps to the instruction labeled `IL_0014`, which just returns with `ret` from the method call. Otherwise, the constant string “`a is greater than 0`” is pushed onto the stack with `ldstr`, as needed to be popped by the call to `WriteLine`.

2.2 The MIO Algorithm

In the context of white-box system testing, there could exist tens/hundreds of thousands of testing targets to be optimized (e.g., each line is regarded as a

Algorithm 1: Pseudo-code of the MIO Algorithm [9]

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

Output : Archive of optimised individuals A

```

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

```

target). To effectively handle such an amount of targets, MIO employs *dynamic populations* management, i.e., each target owns a population with a maximum size n , and the targets along with populations are managed dynamically during search. Inspired by (1+1) Evolutionary Algorithm [31], MIO is composed of two main operators, i.e., sampling and mutator.

As shown in Algorithm 1, the search starts with an empty set of populations T , and an empty archive A which saves feasible solutions for the targets. At each iteration, with either sampling (i.e., sample a test) or mutator (i.e., mutate a test sampled from T) controlled by a probability P_r , a new test p could be produced and then executed (i.e., run the test on the SUT). Note that with the execution, the testing targets could be *not reached*, *reached* or *covered*. Consider the example of a branch target for `if(x == 42)` (at line 3) as below,

```

1 public int foo(x){
2   if(x <= 0) return -1;
3   if(x == 42) return 1;{
4     return 0;
5 }

```

When x is a negative number or 0 (e.g., -5), the target is *not reached* (as `return -1` is executed). When x is any positive number but not 42, the target is *reached* but not *covered* yet. Only if x is 42, the target could be considered

as *covered*. Thus, based on targets achieved by executing p , the populations T could be updated (referred as *dynamic populations*) as:

- if the target k is newly *reached*, a new population T_k which contains p is created and added to T ;
- if the target k is *covered*, p would be added to the archive A , and T_k would be removed from the populations T (i.e., the search would not need to optimize this target);
- otherwise, add p to T_k , if the size of T_k exceeds n , then remove the worst solution.

At the end, the search outputs A which contains a set of best solutions (referred as a test suite) which are feasible to solve testing targets.

Regarding white-box system testing, there might exist some infeasible targets and users would only care about what targets are covered rather than how the targets are heuristically close to be covered. Thus, in order to focus on the targets which could be possibly covered within the search budget, MIO integrates a *feedback-directed sampling* to sample tests which achieve recent improvements as candidates to perform the next mutation (see *SampleIndividual(T)* at line 7). In addition, to trade off between *exploration* and *exploitation* of the search landscape, MIO handles parameters (such as F , P_r and n) dynamically throughout the search. For instance, at the beginning of the search, the exploration (i.e., sampling) helps to reach new targets quickly. Based on the passing of time, the probability of perforating the sampling P_r is linearly reduced. Then, at a certain point F (e.g., 50% of the budget has been used), the search would start to focus more on the exploitation (i.e., $P_r = 0$ and $n = 1$) in order to focus on covering the reached targets.

2.3 Branch Distance

To achieve high code coverage, there is the need to define heuristics to guide the search to generate inputs that can solve the constraints in the system under test (SUT) (e.g., complex predicates in `if` statements). The most common heuristics in the literature is the so called *branch distance* [32]. It was originally designed to handle numerical constraints, but it has also been extended to handle string constraints [13]. As an example, consider a simple statement such as `if (x==42)`. In this statement, if x is taken randomly, there would be 2^{64} possibilities, where only one of them does fulfill the constraint. However, a value such as $x = 50$ is heuristically closer to solve the constraint compared to much larger numbers. In this example, for any given x , the branch distance would be calculated as $d(x) = x - 42$. The search would hence have gradient to modify x to minimize such distance $d(x)$. For a full list of these distance functions, we refer the reader to [32]. A major research challenge we address in this paper is how to apply these branch distances to .NET CIL bytecode.

3 Related Work

EvoSuite [33] is a SBST tool that produces unit test cases with assertions for Java classes. EvoSuite does this by employing a hybrid technique that produces and optimizes whole test suites in order to meet coverage objectives. EvoSuite is perhaps the most known SBST tool, and does bytecode instrumentation for the JVM, supporting branch distance computations.

Where there has been much work on testing Java programs in literature [34] (besides EvoSuite), comparatively not so much has been done for .NET. The most famous example is Pex [35], which uses dynamic symbolic execution to generate small unit test suites for programs developed with .NET. Pex accomplishes this by determining test inputs for Parameterized Unit Tests by a systematic program analysis. By observing execution traces, Pex learns about the program's behavior. Pex generates new test inputs with varying program behavior with the help of a constraint solver.

Randoop [36] is a tool for Java and .NET that creates unit tests by the aid of a feedback directed random testing technique. The goal is to avoid producing illegal and redundant inputs by leveraging execution feedback from executing test inputs while they are created. Randoop builds method sequences one at a time by picking a method call at random and choosing arguments from previously built sequences which acts as a guide to create the new inputs.

To the best of our knowledge, there does not exist any SBST technique in the literature for white-box testing of .NET programs.

Regarding fuzzing RESTful APIs, several tools have been presented in the literature besides EVOMASTER, such as Restler [23], RestTestGen [24], Restest [25], RestCT [26], bBOXRT [27], and Schemathesis [28]. However, they are all black-box. Different studies comparing such tools showed black-box EVOMASTER (with no instrumentation and no SBST heuristics) giving better results [11, 12], closely followed by Schemathesis [28]. In these experiments, SBST white-box fuzzing gave better results than the black-box variant. The work presented in this paper enables practitioners to use white-box testing for .NET applications as well besides JVM ones, which provides better results than black-box testing when the source code is available (e.g., in the case of developers and Continuous Integration systems).

4 .NET Instrumentation

4.1 Bytecode Instrumentation

Our implemented instrumentation for .NET programs is done by the aid of Mono.Cecil¹ library, which makes it possible to analyze and modify CIL code. It works with .NET libraries that are compiled and generated as a DLL (i.e., Dynamic Linked Library), which means that the instrumentation is performed offline and we can not instrument .NET libraries on the fly. Instrumentation needs to be integrated with an SBST technique to generate

¹<https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/>

tests. For conducting the experiments in this paper, we have taken advantage of EVOMASTER, that generates system-level test cases for RESTful APIs. There are two main components in EVOMASTER: a *core* process and a *driver* process which not only starts/stops/resets the SUT, but also is responsible for applying the instrumentation on the SUT with SBST heuristics and generate the instrumented version. The *driver* provides its functionalities as RESTful APIs which the *core* communicates with them through HTTP. The instrumentation is implemented as a .NET Core console application. The main method of this console application takes the path of the target SUT as an input parameter, performs instrumentation on it and finally saves the instrumented file in the specified location.

To use EVOMASTER for an SUT, we have implemented a *driver* written in C# that implements the same endpoints as the original JVM driver. Regarding the *core* which is written in Kotlin, all we had to do was adding a new sort of output type for C#. EVOMASTER is now able to generate test cases which are sequences of HTTP calls to the different endpoints of the SUT based on `xUnit`. More details on how our technique for .NET is integrated in EVOMASTER will be discussed in Section 4.4.

4.2 Code Coverage

A .NET program is made up of a number of assemblies, each one containing some classes. Each class contains methods, which we instrument one by one. Each statement in a method will become a testing target, and we insert probes before and after them to keep track of whenever they get covered during the search. The aim of EVOMASTER is to generate system-level test cases which yield the highest possible number of covered targets.

In order to insert probes before and after each statement, what we have to do first is to identify them. As discussed in Section 4.1, we use `Mono.Cecil` to analyze and alter CIL code that are fetched and iterated over for each method to detect the statements and insert probes before and after them. In our method, for each instruction, we consider its starting coordination (i.e., line and column numbers) in the source code as an indicator of a new statement. That information is obtained from an object of type `SequencePoint` which is assigned to the instruction. If the obtained sequence point is not null and it posses different line and column numbers compared to the previously accessed instruction which we save at the end of the loop, the current instruction is identified as beginning of a statement and we insert an *EnteringStatement* probe before it. This instruction is not only an indicator of a new statement, but also could be the end of another statement, except the cases which the statement is the first one in the method. As a result, we also insert another `CompletedStatement` probe to signal the end of previous statement. Take this simple instrumented code as an example: Take this simple variable assignment as an example:

```
1 public class MyClass {
2     public void MyMethod() {
```



```

3      int x = 123;
4  }
5 }

```

The instrumented code would be:

```

1 public class MyClass{
2     public void MyMethod(){
3         Probes.MarkStatementForCompletion("MyClass", "
MyMethod", 2, 32);
4         ProbesEnteringStatement("MyClass", "MyMethod", 3,
13);
5         int num = 123;
6         Probes.CompletedStatement("MyClass", "MyMethod", 3,
13);
7         Probes.MarkStatementForCompletion("MyClass", "
MyMethod", 4, 9);
8     }
9 }

```

The probes, i.e., `EnteringStatement` and `CompletedStatement`, are inserted before and after the variable assignment. They are nothing but invoking of static methods inside the console application developed to support the instrumentation. The parameters passed to these probes include class name, method name, line number and column number, respectively, as they are necessary to create unique *IDs* for the targets (i.e., statements). The initial and final curly braces which surround the method's body are also considered as statements. Their counterpart instructions in the CIL which is `nop` and `ret` for the opening and closing curly braces respectively, have their own line and column number. Besides, it is useful to have them as we can find out if a method is reached even if it is empty or if the execution of a method is completed. The probes (i.e., `MarkStatementForCompletion`) to mark completion of the curly braces are inserted at lines 3 and 7 in the instrumented code above. To provide the heuristics on code coverage, we take the same technique with the SBST heuristic values as it is done with EVOMASTER now. Each target will be assigned a heuristic value in the range $h \in [0, 1]$, with 1 indicating that the target has been completely covered and 0 showing that the target has not been reached throughout the test case evaluation. The values between 0 and 1 show how close a test case came to covering the target heuristically.

As mentioned earlier, by executing each probe, a static method will be called. Whenever `EnteringStatement` is executed, the targets for class and line are marked as covered by setting their values to 1. For the statement, it is set to 0.5. The reason behind this is that statements may throw exceptions, and we will not know if no exceptions were thrown until the statement is fully executed. The heuristic value for statements is set to 1 only if the `CompletedStatement` is reached. The significance of having two probes per statement is clear here. If we merely reported the line targets with $h = 1$, the search would have no way to realize if an exception was thrown, and would not exploit input data that does not lead to an exception. On the other hand, consider the case where an exception is thrown in the statement (e.g, a divide by zero operation), and $h = 0.5$. Because SBST technique (such as EVOMASTER) typically only

outputs test cases for targets that are fully covered, if there was no target for the line, the test case would not be included in the final output test suite.

Inserting `EnteringStatement` and `CompletedStatement` is not always straightforward as shown in the example above. When it comes to instructions which change the control flow, the program may become corrupted or the logic may change if not handled cautiously. For any instruction that we put `EnteringStatement` before, it is likely that it would be target of a jump instruction somewhere else in the code. In that case, the `EnteringStatement` probe may not get reached. To solve this, we have to check if the current instruction is the target of any jump. This is done by iterating over the method's body instructions which is an array. If yes, the target value of the jump(s) should change to the first instruction of `EnteringStatement` probe. For `CompletedStatement` probe, the main challenge is that they should not be put after instructions which perform jump or exit unconditionally (i.e., `br`, `throw`, `rethrow`, `endfinally`, `leave` and `ret`). If this happens, we have to insert the `CompletedStatement` probe before those instructions to mark them as completed. This should not be a problem as there is no instruction in between which may throw exception or change the control flow. Another issue that we faced during enabling instrumentation, was that there exist instructions which there is also a short form for them such as `ldarg` which is a two byte instruction and `ldarg.s` that is one-byte. When altering CIL code, the number of arguments, local variables or method bytes may change. In this case, an overflow may occur and affect the CIL. To prevent that, we convert every short form instruction to its non-short version by calling `SimplifyMacros` of `Mono.Cecil` which is an extension method for `MethodBody`. When finally the instrumentation is done, we can call another method named `OptimizeMacros` which converts them back to their short form if possible.

4.3 Branch Distance

4.3.1 Numeric

Covering an acceptable number of targets is hard to achieve without taking the branch instructions into consideration. Complex predicates, such as conditions in if statements, can affect the SUT's control flow. As the code in Listing 1, As the example shown in Section 2.1, the if-statement would be compiled into `cgt`, and `brfalse.s` is to manage the control flow with a result (either 0 or 1) of the if predicate `a>0`.

There are different types of instructions that are identified as branch instructions. Based on the values popped from the evaluation stack, numerical value types can yield either one or two instructions. Table 1 shows all branch related instructions that we have handled in our instrumentation. We categorize these branch instructions into three groups. *One-arg jump* instructions pop a value from the stack and transfer control to a target instruction based on the popped value. These instructions are `brtrue`, `brtrue.s`, `brfalse` and `brfalse.s`. They perform the transfer control provided that the popped value

Table 1: Branch instructions that our instrumentation deals with for distance calculation

Name	Branch instructions	Description
One-arg jump	brtrue, brtrue.s, brfalse, brfalse.s	Jump instruction with one argument
Two-arg compare	ceq, clt, clt.un, cgt, cgt.un	Compare instruction with two arguments
Two-arg jump	bgt, bgt.s, bgt.un, bgt.un.s bge, bge.s, bge.un, bge.un.s ble, ble.s, ble.un, ble.un.s blt, blt.s, blt.un, blt.un.s beq, beq.s, bne.un, bne.un.s	Jump instruction with two arguments

is true, not null, or non-zero. *Two-arg compare* includes comparison instructions that pop two values from the stack, compare them and push the result which could be either 0 or 1. This group consists of `ceq`, `clt`, `clt.un`, `cgt` and `cgt.un` instructions. As an example, `cgt` compares two values and pushes 1 if the first one is strictly greater, otherwise 0 is pushed. *Two-arg jump* instructions perform a jump to another instruction after popping and comparing two values from the stack. This includes `bgt`, `bgt.un`, `bge`, `bge.un`, `ble`, `ble.un`, `blt`, `blt.un`, `beq`, `bne.un` and also their short forms (i.e., followed by `.s` e.g., `bge.s`). As an example, `blt` transfers control to a target instruction if the first value is strictly lower than the second one.

In order to calculate branch distance, we insert another group of probes into the code. These probes need the value(s) passed to the branch statement (e.g., an `if` statement) to calculate how far they are to fulfill the constraint. However, since the values are on top of the evaluation stack, and the branch instruction pops them, they have to be duplicated. For one-arg jumps, the duplication is straightforward. In the code below, `brfalse.s` pops from the evaluation stack and performs a jump to another instruction at `IL_001C` if the popped value is false (i.e., zero). Examples of the jumps and the instrumented version would be:

```

1 IL_0014: ldloc.1      // push local variable at index 1
    onto the stack
2 IL_0015: brfalse.s    IL_001c

1 IL_008a: ldloc.1      // push local variable at index 1
    onto the stack
2 IL_008b: dup
3 IL_008c: ldstr         "brfalse" //opCode
4 IL_0091: ldstr         "TriangleClassificationImpl" //
    className
5 IL_0096: ldc.i4.6      //lineNo
6 IL_0097: ldc.i4.4      //branchCounter
7 IL_0098: call         void [EvoMaster.Instrumentation]
    EvoMaster.Instrumentation.Probes::
    ComputeDistanceForOneArgJumps(int32, string, string,
    int32, int32)
8 IL_009d: brfalse.s    IL_0101

```

The probe for calculation of branch distance is inserted right before the branch instruction. Apart from the values needed for marking the branch target that are `opCode`, `className`, `lineNo` and `branchCounter` (starts from zero, indicates the number of branches per line), the actual value which the `brfalse.s` pops need to be passed to the probe as well. It is done by adding the `dup` instruction which duplicates the value pushed by its previous one (i.e., `ldloc.1` at IL_008a) onto the evaluation stack.

Calculating branch distance for two-arg compare and two-arg jump instructions is more challenging. The first challenge is that these sort of instructions take two values as input, but it is not possible to duplicate the top two values on the stack as we handle for *one-arg jump* instructions, as previously shown. As a result, we take advantage of bytecode method replacements. Whenever an instruction of types two-arg compare and two-arg jump is reached, we replace it with a method which performs the same semantic as the original instruction in addition to calculating the branch distance.

```
1 IL_0020: ldarg.2 //Load the argument at index 2 onto the
    evaluation stack
2 IL_0021: ldarg.3 //Load the argument at index 3 onto the
    evaluation stack
3 IL_0022: ceq
4 IL_0024: br.s      IL_0027
```

For example, the `ceq` instruction pops two values from arguments of the method, compares them and pushes 1 if they both are equal and 0 if they are not. The instrumented code for the example above would be like this:

```
1 IL_012a: ldarg.2 //Load the argument at index 2 onto the
    evaluation stack
2 IL_012b: ldarg.3 //Load the argument at index 3 onto the
    evaluation stack
3 IL_012c: ldstr      "ceq" //pushes the opcode string
4 IL_0131: ldstr      "TriangleClassificationImpl"
5 IL_0136: ldc.i4.s    10 // push lineNo
6 IL_0138: ldc.i4.1    // push branchCounter
7 IL_0139: call       int32 [EvoMaster.Instrumentation]
    EvoMaster.Instrumentation.Probes::
    CompareAndComputeDistance(int32, int32, string, string
    , int32, int32)
8 IL_013e: br.s      IL_0141
```

There is no `ceq` instruction anymore and it is replaced by a method call, i.e., `CompareAndComputeDistance`. The method first calculates the distance by passing the first two numeric values pushed by instructions at IL_012a and IL_012b and then decides what value should it push onto the stack based on the opcode string pushed at IL_012c.

For two-arg jump instructions, the replaced method is different. These kind of instructions first compare the two input values and then jump to another point based on the comparison result which is on the evaluation stack. It means that we can replace each one of them with a two-arg compare instruction followed by a one-arg jump instruction. Table 2 contains the information on how we map those instructions.

Table 2: Mapping two-arg jump instructions to two-arg compare and one-arg jump instructions

Original Instruction	Converted Instructions	Original Instruction	Converted Instructions
bgt	cgt + brtrue	ble	cgt + brfalse
beq	ceq + brtrue	blt	clt + brtrue
bge	clt + brfalse	bne	ceq + brfalse

The second challenge for two-arg compare and two-arg jump instructions is that not only the two values should be duplicated, but also their data type has to be detected. The values in the examples above are of type `int` but they could be of any other numeric types such as `float` or `long`. Knowing the data type is a must as it is necessary for calling the right probe. A handy solution to this might be having a method which takes values of type `object` as input so it can handle any type of value. However, this is not possible because the input values first have to be boxed to `object`. There exists an instruction for boxing to `object` but it also takes the sub-type (e.g., `int`) as input which we do not have. Therefore, a feasible solution would be to have methods with different overloading for handling various numeric data types and detect the type of values pushed by the last two instructions before the instruction of type two-arg jump during instrumentation in order to insert the right probe to call.

The detected type depends on what the last two instructions are. The branch instruction could appear after many kinds of instructions that push values on the evaluation stack. Since both values are always of the same type, detecting type for either of them suffices. For `FieldDefinition`, `VariableDefinition` and `MethodReference` (e.g., calling a method which puts an `int` onto the stack), the type can be inferred by casting the instructions' operand and returning its type property. If the instruction is loading from method's argument (e.g., `ldarg_0` which pushes first parameter of the method), all we have to do is to detect its index and find the element with the same index in the method's parameters which can be inferred using `Mono.Cecil` from current method's metadata and return its datatype. Another possibility is that the previous instruction is loading a local variable (e.g., `ldloc`). These instructions load the local variable at a specific index onto the evaluation stack. Every `ldloc` variable comes after a `stloc` which stores a value at the specified index at the local variables list. However, these two instructions might not be necessarily close to each other. For tackling this issue, we store every local variable name and its datatype in a `Dictionary` by detecting `stloc` instructions. Whenever we reach a `ldloc`, all we have to do is to get the type of the current local variable by referring to the dictionary. There are also a group of instruction which their datatype can be detected based on their title. For example, `Ldc_I4`, `Ldc_I8`, `Ldc_R4` and `Ldc_R8` push a value of type `int`, `long`, `double` and `float` respectively onto the stack. Therefore, we can detect their data type just by parsing their `OpCode`.

4.3.2 String

Besides the numeric value types, there would be a need to provide branch distances for the string type. To enable this, we identify all operators and methods of `System.String` that return a boolean, such as “==” operator, `Equals`, `Contains`, `StartsWith` and `EndsWith` during the instrumentation. Then, we replace the method with a corresponding probe which calculates the distance (e.g., based on [13]) and does the intended operation (recall the instrumentation should not modify the semantics of the program). Consider the code example `a.Equals(b, StringComparison.OrdinalIgnoreCase)`. Its equivalent CIL code is:

```
1 IL_0001: ldarg.1      // a
2 IL_0002: ldarg.2      // b
3 IL_0003: ldc.i4.5
4 IL_0004: callvirt     instance bool [System.Runtime]System
    .String::Equals(string, valuetype [System.Runtime]
    System.StringComparison)
```

The code above, takes `a` and `b` from method’s parameters, compares them and pushes the result onto the evaluation stack. During instrumentation, once the `System.String::Equals` is identified, we would replace it with a method to enable SBST white-box heuristics. The instrumented version is as follows:

```
1 IL_003a: ldarg.1      // a
2 IL_003b: ldarg.2      // b
3 IL_003c: ldstr        "System.Boolean System.String::
    Equals(System.String)"
4 IL_0041: ldstr        "MyClass"
5 IL_0046: ldc.i4.s      14 // line number
6 IL_0048: ldc.i4.0      // branch counter
7 IL_0049: call         bool [EvoMaster.Instrumentation]
    EvoMaster.Instrumentation.Probes::StringCompare(string
    , string, string, string, int32, int32)
```

As the instrumented version, the `Equals` method is replaced with a method call (i.e., `StringCompare`) which does calculates the branch distance, performs the comparison and pushes the result onto the evaluation stack. There are four other CIL instruction between the method call and the instructions for loading the arguments which push the necessary information to mark the string comparison operation, i.e., string comparison operator, class name, line number and branch counter. With the four argument, we define a unique target ID as `idTemplate` for this comparison. Regarding string comparison operator, now we support “==”, `Equals`, `Contains`, `StartsWith` and `EndsWith`. The implementation of `Equals` as this example is shown in Figure 1. The method would take `caller` (e.g., `a`), `anotherString` (e.g., `b`) and `comparisonType` as inputs. The comparison operator ID, i.e., `idTemplate`, would result in two new testing targets, i.e., `true_branch` and `false_branch` for the method call. Note that besides `true_branch`, we are also considering `false_branch` as testing target, to provide better guidance to the search. In this replacement method, it first ensures that `caller` is not null, otherwise it throws a `NullReferenceException`. Then, at line 4, it performs the same semantic as the original

string `Equals` method. Regarding the distance, we employed `Truthness` which is a utility class to store the two heuristic values (i.e., `ofTrue` and `ofFalse`) between 0 and 1, representing possible outcomes, i.e., `true` or `false` respectively. 1 indicates that the outcome (e.g., `false`) is covered. As seen from line 6, if `anotherString` is null, we would set `ofTrue` with a constant value (`H_REACHED_BUT_NULL`) representing Null case (e.g., 0.05) and set `ofFalse` with 1. Note that here instead of assigning 0 to `ofTrue`, we employ a small positive value for indicating to the search that the branch is *reached* but far from being *covered*. Lines 11-17 are to handle situations whereby `result` is `true` or `false`. Regarding `true`, the distance is assigned as 1 for `ofTrue` and a constant value (`H_NOT_NULL`) representing NotNull case (e.g., 0.1 which is greater than Null case) for `ofFalse`. If the result is `false`, lines 13-15 are to calculate a distance `h` to represent how close it is to be true. For string comparison, we employed the same Left-Alignment distance [33] as EvoSuite (see line 14), and further scale the value with `H_NOT_NULL`. Thus, with the above handling, we could calculate heuristic values of `true_branch` and `false_branch`, then update them at line 18. Those heuristics would be further utilized by the search to evaluate a test.

```

1 public static bool Equals(string caller, string
    anotherString, StringComparison comparisonType, string
    idTemplate){
2     ObjectExtensions.RequireNonNull<string>(caller);
3     ExecutionTracer.HandleTaintForStringEquals(caller,
        anotherString, comparisonType);
4     bool result = caller.Equals(anotherString,
        comparisonType);
5     if (idTemplate == null) return result;
6     if (anotherString == null){
7         ExecutionTracer.ExecutedReplacedMethod(idTemplate,
            ReplacementType.BOOLEAN, new Truthness(DistanceHelper.
            H_REACHED_BUT_NULL, 1));
8         return false;
9     }
10    Truthness t;
11    if (result){ t = new Truthness(1d, DistanceHelper.
        H_NOT_NULL);}
12    else{
13        double baseS = DistanceHelper.H_NOT_NULL;
14        double distance = DistanceHelper.
            GetLeftAlignmentDistance(caller, anotherString,
            comparisonType);
15        double h = DistanceHelper.
            HeuristicFromScaledDistanceWithBase(baseS, distance);
16        t = new Truthness(h, 1d);
17    }
18    ExecutionTracer.ExecutedReplacedMethod(idTemplate,
        ReplacementType.BOOLEAN, t);
19    return result;
20 }

```

Fig. 1: An implementation of a replacement method for `System.String.Equals`

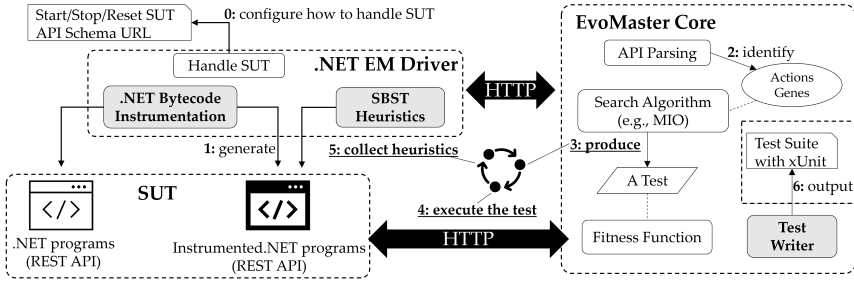


Fig. 2: Integrating our novel techniques into EVOMASTER for enabling white-box system test generation for .NET applications.

In addition, to further handling strings, we also employ a *taint analysis*, by tracking how input strings are compared at runtime, in the same way as done in EVOMASTER for programs running on the JVM [37]. For instance, for `a.Equals("foo")`, with the line 3, we would track that `a` was compared with “foo”. If the value of `a` is present in the chromosome of the test, then such value “foo” can be directly set as the input (e.g., `a="foo"`) during a mutation operation. This kind of technique can have drastic speed ups to the search process [37].

Due to their length, we do not provide full algorithms for all these string method replacements in this paper. However, note that our implementation is open-source on GitHub, with long term storage on Zenodo at each new release (e.g., 1.5.0 [38]). More details on the other replacement methods can be found there.

4.4 Integration with EVOMASTER for .NET Testing

EVOMASTER is an open-source tool [4] aimed at automatically generating system tests with SBST. It supports both white-box and black-box testing. EVOMASTER uses Many Independent Objective (MIO) which is a population-based evolutionary algorithm designed to deal with white-box system testing problem for automating test suite generation [9].

To apply our technique for testing .NET programs, we integrated it into EVOMASTER as shown in Figure 2. As shown in the figure, to enable our technique, we developed a *.NET EM driver* which contains *.NET Bytecode Instrumentation* and *SBST Heuristics*, and extended *core* with a *Test Writer* which generates test suites written in C# (using *xUnit*). Note that the *Test Writer* can also be used by EVOMASTER when applying black-box testing.

EVOMASTER is mainly composed of two parts, i.e., *driver* and *core*. At the *driver* side, users will need to manually specify how to handle the SUT, e.g., start/stop/reset the SUT and provide a URL where to access its API schema (see Step 0). Then, with *.NET Bytecode Instrumentation*, we automatically generate an instrumented SUT, where a set of probes is injected, as discussed in Section 4 (see Step 1). From the *core* side, at the beginning of the search,

Table 3: Statistics on the used APIs, including the number of lines of code (#LOCs) and number of REST endpoints (#Endpoints).

Name	#LOCs	#Endpoints
<i>cs-rest-ncs</i>	809	6
<i>cs-rest-scs</i>	759	11
<i>menu-api</i>	2706	12

API Parsing (see Step 2) would parse the API schema in order to identify what endpoints are available (referred as *Action*) and what data could be manipulated (referred as *Genes*). Then, with such *Actions* and *Genes*, the search will produce a test (i.e., a sequence of requests with manipulated inputs) with an applicable search operator (e.g., the mutator in MIO). Next, the *Fitness Function* would execute the test on the SUT (see Step 4), and collect info on the targets achieved by this execution. The achieved targets info (such as *class coverage*, *line coverage*, *statement coverage* (Section 4.2) and *branch coverage* (Section 4.3)) are collected at runtime on the driver side, based on *SBST Heuristics* with probes injected into the SUT (see Step 5). Such runtime coverage info allows *Fitness Function* to evaluate a test, e.g., with our white-box heuristics, we could know that $x=50$ is heuristically better than $x=100$ for covering the branch `if(x==42)`. A test with $x=50$ would have a higher chance to be evolved by the search for optimizing that branch target. During the search, Steps 3-4-5 would be performed iteratively within the specified search budget, and, at the end, the best tests will be outputted with a specified format, i.e., C# in our case (see Step 6).

5 Empirical Study

5.1 Research Questions

To assess our technique, we carried out an empirical study to answer the following research questions:

RQ1: Does our approach enable effective white-box SBST heuristics to guide the search for fuzzing .NET/C# RESTful APIs?

RQ2: What type of constraints can be solved? And which ones cannot?

RQ3: What is the impact of applying time as stopping criterion on the performance of the approach?

5.2 Experiment Setup

Our novel technique enables the use of white-box SBST heuristics for testing .NET programs. To evaluate it, we integrated our technique into EVOMASTER (denoted as EVOMASTER._{NET} discussed in Section 4.4) and conducted our experiments with three .NET REST APIs, i.e., C# REST Numerical Case Study (*cs-rest-ncs*), C# REST String Case Study (*cs-rest-scs*) and Menu API (*menu-api*). The first two case studies were initially designed for studying unit

testing approaches on solving numerical [39] and string [13] problems. These two have been re-implemented as RESTful APIs in various programming languages (e.g., Java and JavaScript) to evaluate white-box test generation problems [7, 37, 40]. Here, we re-implemented them with C#, and made them accessible via a REST API. The latter case study, i.e., *menu-api* is one of the backend services of the popular **Restaurant-App**, which is an existing open-source project on GitHub² (with currently more than 500 stars). Unlike the two other case studies, *menu-api* deals with a PostgreSQL database. Table 3 shows the statistics on these APIs. To ease the replication of this study, all these APIs are included in the EMB repository [41].

The APIs *cs-rest-ncs* and *cs-rest-scs* were chosen in this study to make sure that our SBST heuristics for .NET do work properly for numeric and string constraints. As previous results for the JVM show good results for SBST techniques on these APIs (e.g., [42]), then we should expect the same good results for .NET if our techniques work as intended. To show the application of our techniques on actual RESTful APIs, we searched GitHub for .NET RESTful APIs, prioritizing based on popularity (represented with number of stars). Unfortunately, although C#/.NET is widely popular in industry, for historical reasons (e.g., due to close-source tooling and tight ties to the Windows platform) it is less so among open-source projects (although in the recent years things have started to change). Finding suitable APIs among open-source projects turned out to be rather challenging. The API *menu-api* was the first one that we found that met our criteria.

As discussed in Section 3, to the best of our knowledge, there is no existing SBST technique for white-box testing of .NET programs. Thus, to evaluate the effectiveness of our approach, we performed a comparison between two algorithms developed in EVOMASTER, i.e., MIO and Random, regarding their performances achieved by generated tests. MIO is the default algorithm employed with SBST heuristics in EVOMASTER for test generations, while Random is just a naive random search, used as baseline. Note that the Random algorithm could be regarded as grey-box testing since it still tracks tests based on what targets are covered (e.g., line coverage) and outputs the best of them at the end. We do not compare with other black-box fuzzers in this paper, as black-box EVOMASTER (which just does a random search) already gives the best results in existing tool comparisons [11, 12].

In the context of software testing, we used four criteria to assess the performance of the techniques, i.e., target coverage (`#Targets`), line coverage (`%Lines`), branch coverage (`%Branches`) and detected faults (`#Faults`). `#Targets` is the aggregated criterion which considers all coverage metrics that EVOMASTER optimizes for (e.g., including coverage of HTTP status codes per endpoint). With our instrumentation, we enable *class coverage*, *line coverage*, *statement coverage* and *branch coverage* as parts of the `#Targets` to be optimized. Regarding `%Lines` and `%Branches`, they are widely applied metrics to assess testing approaches in practice. Faults are detected based on returned

²<https://github.com/chayxana/Restaurant-App>

Table 4: Average and pairwise comparison results for MIO and Random with four metrics, i.e., #Targets, %Lines, %Branches and #Faults.

SUT	Metrics	MIO	Random	\hat{A}_{12}	p -value	Relative
<i>cs-rest-ncs</i>	#Targets	992.1	656.4	1.00	≤ 0.001	+51.14%
	%Lines	85.5%	55.4%	1.00	≤ 0.001	+54.36%
	%Branches	76.4%	51.6%	1.00	≤ 0.001	+47.93%
	#Faults	6.0	5.0	1.00	≤ 0.001	+20.00%
<i>cs-rest-scs</i>	#Targets	967.8	617.6	1.00	≤ 0.001	+56.71%
	%Lines	73.6%	57.4%	1.00	≤ 0.001	+28.03%
	%Branches	32.5%	25.9%	1.00	≤ 0.001	+25.33%
	#Faults	1.0	1.0	0.50	NaN	+0.00%
<i>menu-api</i>	#Targets	333.5	333.7	0.43	0.588	-0.06%
	%Lines	29.1%	29.1%	0.50	NaN	+0.00%
	%Branches	1.8%	1.7%	0.55	0.651	+2.70%
	#Faults	22.7	23.0	0.35	0.077	-1.30%

500 HTTP status code, and on mismatches in the responses based on the API schemas. Regarding parameter settings, in the first set of experiments the search budget for the two algorithms is assigned as 100 000 HTTP calls, which is a commonly used setting by existing research work with EVOMASTER [7, 37, 40]. In the second set of experiments, we used 1 hour as stopping criterion. For the other parameters (e.g., F and P_r), we apply their default values as EVOMASTER. Considering randomness inherited from search algorithms, we repeated our experiments 10 times for MIO and Random algorithms on the three case studies, by following common guidelines in the literature [43]. The experiment was executed on a DELL laptop with the following specifications: Processor 11th Gen Intel(R) Core(TM) i9-11950H @2.60GHz 2.61 GHz; RAM 32 GM; Operating System 64-bit Windows 10.

5.3 Experiment Results

5.3.1 Results for RQ1

To answer RQ1, Table 4 presents average coverage results achieved by MIO and Random with #Targets, %Lines, %Branches and #Faults. Based on the average results, MIO consistently achieves the best for both numeric and string problems on all the four metrics. In the table, we also performed a statistical analysis to compare MIO and Random with four metrics using Mann-Whitney-Wilcoxon U-tests (p -value) and Vargha-Delaney effect sizes (\hat{A}_{12}). In the table, value in **bold** indicates that MIO is significantly better, i.e., p -value < 0.05 (significance level) and $\hat{A}_{12} > 0.5$. Considering the results shown in the Table 4 for *cs-rest-ncs* and *cs-rest-scs*, MIO significantly outperforms Random for all metrics, with high effect sizes and low p -values on the case studies. For *menu-api*, there is no significant difference between two algorithms.

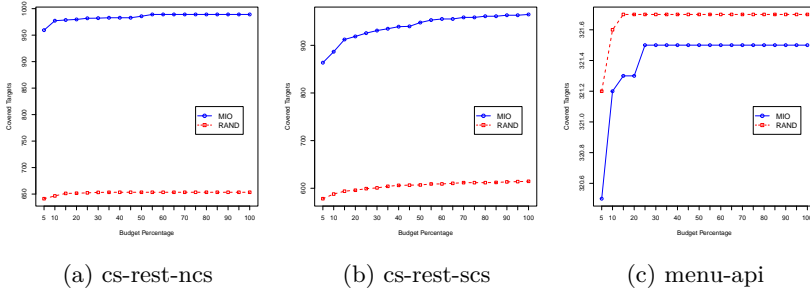


Fig. 3: Average covered targets (y-axis) achieved by MIO (blue) and Random (red) throughout the search (RQ1) at every 5% intervals of the budget spent by the search (x-axis)

Figure 3 reports plot-lines displaying changes on the number of covered targets achieved by MIO and Random throughout the search, collected at different time intervals. For *cs-rest-ncs* and *cs-rest-scs*, MIO shows a clear advantage over Random throughout the whole search. This further demonstrates the effectiveness of white-box SBST heuristics to solve numeric and string problems.

Regarding the number of found faults (i.e., #Faults), MIO has managed to find one more compared to Random for *cs-rest-ncs*. However, no improvement is achieved for *cs-rest-scs* and *menu-api*. This is understandable for *menu-api* as MIO did not have any higher coverage which reduces the likelihood of detecting more faults. The higher coverage could result in more detected errors. Regarding *cs-rest-scs*, since it is an artificial SUT with less complexity compared to *menu-api*, the one detected fault could be the only existing potential fault in the code. Figure 4 shows a generated test case for *menu-api* that detects an internal server error. The error occurs when the client tries to access a specific food item but the resulted HTTP status code is 500 which denotes an error on the server side.

With results on two of the case studies, our white-box technique achieved significant higher code coverage over the grey-box random testing in the two out of three case studies. This demonstrates the effectiveness of SBST heuristics in guiding white-box testing of numerical and string programs in .NET.

5.3.2 Results for RQ2

Based on the results in Table 4, MIO achieved 85.5% line coverage on *cs-rest-ncs*, 73.6% line coverage on *cs-rest-scs*, and 29.1% line coverage on *menu-api* on average with 10 repetitions. To analyze the performance in detail, we further investigated code coverage by executing the best and worst tests on the SUTs using JetBrains Rider [44]. Note that the line coverage reported by EVOMASTER instrumentation excludes the coverage achieved at boot-time,

thus, the coverage reported by Rider would be higher than the coverage in Table 4.

Figure 5 shows the code of one test generated for *cs-rest-ncs*. Here, an asynchronous HTTP call is made toward the endpoint `/api/triangle/653/653/653`. Then, the test verifies that the status code of the HTTP request is 200, the body payload is of type JSON, and finally the JSON response has a field called `result` with value equal to 3. Note that, when a test suite is generated containing several tests, scaffolding code is generated as well, like for example the one presented in Figure 6. Here, the API is started, reset at each test execution, and shut down once all tests are completed (using the driver classes directly, like `NcsDriver.EmbeddedEvoMasterController()` in this case). This is essential to be able to use these kinds of tests for regression testing.

Regarding *cs-rest-ncs*, we found that most of the numerical branches could be solved, i.e., the line coverage on `NCS.Imp` namespace is between 93% (the worst test suite of 10 repetitions) and 98% (the best test suite of 10 repetitions) on `Fisher`, `Remainder`, `Triangle`, `Bessj`, `Expint` and `Gammq`. By checking the uncovered code, they are due to dead-code, e.g., `if(n==2)` branch of `Bessj.BessjFunction(n,x)` cannot be covered since there is precheck before invoking the method. Then *cs-rest-ncs* could be regarded as a *solved* problem.

Regarding *cs-rest-scs*, the code coverage on `SCS.Imp` namespace is between 72% (the worst) and 86% (the best). In most of cases, MIO is capable of achieving over 73% coverage on `Costfuns`, `DateParse`, `NotyPevear`, `Ordered4`, `Title`, `Text2Txt`, `Calc`, `Cookie`, `Regex` and `Pat`. By comparing the worst and the best, `Ordered4` shows a large difference on `Ordered4` (i.e., 33% by the worst *vs.* 100% by the best). The uncovered code by the worst run is related to `int string.Compare(x,y)` (e.g., `string.Compare(z, y, StringComparison.Ordinal) > 0`). Since we do not have heuristic for this method which returns `int`, covering such branches would be by chance. Further heuristic with replacement method would be needed to better cover such branches. Another

```

1 [Fact]
2 public async Task test_16_with500() {
3
4
5     Client.DefaultRequestHeaders.Clear();
6     Client.DefaultRequestHeaders.Add("Accept", "/*/*");
7     var res_0 = await Client.GetAsync(_fixture.baseUrlOfSut
8         + "/api/v1/Foods/0hc");
9     Assert.Equal(500, (int) res_0.StatusCode); //
10    Statement_FoodRepository_00031_4
11    Assert.True(string.IsNullOrEmpty(await res_0.Content.
12        ReadAsStringAsync()));
13 }

```

Fig. 4: An example of a generated test case which detects a potential fault based on 500 HTTP status code

```

1 [Fact]
2 public async Task test_17() {
3     Client.DefaultRequestHeaders.Clear();
4     Client.DefaultRequestHeaders.Add("Accept", "*/*");
5     var res_0 = await Client.GetAsync(_fixture.
        baseUrlOfSut + "/api/triangle/653/653/653");
6
7     Assert.Equal(200, (int) res_0.StatusCode);
8     Assert.Contains("application/json", res_0.Content.
        Headers.GetValues("Content-Type").First());
9     dynamic body_1 = JsonConvert.DeserializeObject(await
        res_0.Content.ReadAsStringAsync());
10    Assert.True(body_1.result == "3");
11 }

```

Fig. 5: Example of generated test for *cs-rest-ncs*.

```

1 public class ControllerFixture : IDisposable {
2
3     public ISutHandler controller { get; private set; }
4     public string baseUrlOfSut { get; private set; }
5
6     public ControllerFixture() {
7         controller = new NcsDriver.
            EmbeddedEvoMasterController();
8         controller.SetupForGeneratedTest();
9         baseUrlOfSut = controller.StartSut ();
10        Assert.NotNull(baseUrlOfSut);
11    }
12
13    public void Dispose() {
14        controller.StopSut ();
15    }
16 }
17
18 public class EM_MIO_1_Test : IClassFixture<
    ControllerFixture>{
19
20     private ControllerFixture _fixture;
21     private static readonly HttpClient Client = new
        HttpClient();
22
23     public EM_MIO_1_Test (ControllerFixture fixture){
24         _fixture = fixture;
25         _fixture.controller.ResetStateOfSut();
26     }

```

Fig. 6: Example of generated test suite scaffolding.

large difference between the worst and the best is on **Pat** related to **length** of a string (55% vs. 95%), which does not have any direct string input. Then, in order to solve such branches, there might need a larger search budget. This is an experiment that would be conducted in the future. Moreover, both of the worst and the best run achieved 73% line coverage on **Regex** and limited coverage (i.e.,

16%) on `FileSuffix`. For `Regex`, it is related to predicates using a method `System.Text.RegularExpressions.Regex.IsMatch(txt, pattern)`. The `pattern` could be `url` or `date` in this case study. To cover such branch, *testability transformations* [37] are required to be implemented in our approach for .NET programs. For `FileSuffix`, there is no code which could be covered after line 3 as below:

```
1 var fileParts = file.Split(".");
2 var lastPart = fileParts.Length - 1;
3 if (lastPart <= 0) return "" + result;
```

The branch target could be solved if there exist at least one “.” in the string `file`. However, we now only enable replacing methods of `String` which are related to boolean predicates (as discussed in Section 4.3.2). To effectively have such `string separator`, taint analysis and replacement methods are needed to support for the methods, e.g., `Split`, which have not been handled yet. Thus, without a further handling on these methods with white-box heuristics, related branches might be not easy to solve by the search within the given budget (i.e., 100k HTTP calls). However, those methods could be further supported, e.g., involving `string separator` of `Split` as parts of our taint analysis.

Regarding *menu-api*, there is no difference between the worst and the best. Both achieve 67% line coverage on `Menu.API` namespace. In addition, as it is shown in Table 4, no significant better results is yielded in any of the metrics by MIO. As checked in the uncovered code, this is mainly due to lack of supporting databases in our instrumentation heuristics. For JVM white-box testing, EVOMASTER supports SQL handling [15] which can calculate heuristics for SQL queries and insert data directly into database. However, currently, for this replication study we did not implement yet such technique in our bytecode instrumentation, as it is a complex engineering effort. This is also a room for further improvement in the future. Based on above analysis, we can conclude that:

With a further analysis on code coverage in detail, we found that our white-box SBST heuristics are capable of fuzzing .NET/C# REST API, i.e., the line coverage achieved by generated tests is between 67% and 98%. However, we also identify some limitations due to lack of handling on database which can be addressed in future work.

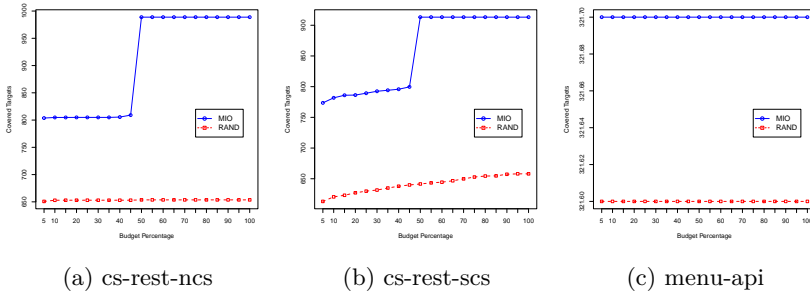
5.3.3 Results for RQ3

The evaluation conducted to answer the first two research questions used stopping criterion 100 000 as the maximum number of evaluated actions (i.e., HTTP calls), as each test case could have a different number of actions in them (and so the number of fitness evaluations would not be a fair metric for comparisons). With the maximum 100 000 HTTP calls as stopping criterion, the average execution time per run was 23 minutes. But, the time spent for different case studies or configurations could be different.

To investigate impacts of different stopping criterion (i.e., the maximum number of HTTP calls vs. time budget), we conducted a further experiment with 1 hour time budget as stopping criterion. Table 5 contains the results

Table 5: Average and pairwise comparison results of run with one hour time budget for MIO and Random with four metrics, i.e., #Targets, %Lines, %Branches and #Faults.

SUT	Metrics	MIO	Random	\hat{A}_{12}	p -value	Relative
<i>cs-rest-ncs</i>	#Targets	991.8	656.6	1.00	≤ 0.001	+51.05%
	%Lines	85.8%	55.5%	1.00	≤ 0.001	+54.63%
	%Branches	76.1%	51.5%	1.00	≤ 0.001	+47.69%
	#Faults	6.0	5.0	1.00	≤ 0.001	+20.00%
<i>cs-rest-scs</i>	#Targets	916.2	661.0	1.00	≤ 0.001	+38.61%
	%Lines	70.8%	60.7%	1.00	≤ 0.001	+16.72%
	%Branches	32.0%	27.7%	0.95	≤ 0.001	+15.81%
	#Faults	1.0	1.0	0.50	NaN	+0.00%
<i>menu-api</i>	#Targets	333.7	333.6	0.55	0.681	+0.03%
	%Lines	29.1%	29.1%	0.50	NaN	+0.00%
	%Branches	1.7%	1.7%	0.55	0.681	+2.78%
	#Faults	23.0	23.0	0.50	NaN	+0.00%

**Fig. 7:** Average covered targets (y-axis) achieved by MIO (blue) and Random (red) throughout the search (RQ3) at every 5% intervals of the one hour time budget spent by the search (x-axis)

of these new experiments. The obtained results do not show any meaningful difference with that of Table 4. It shows that applying time as stopping criterion has not made any significant difference.

In addition, Figure 7 shows changes on the number of covered targets achieved by MIO and Random throughout the experiment with one hour time budget. Similar to Figure 3, MIO has yielded better results compared to Random when applied to *cs-rest-ncs* and *cs-rest-scs*. However, there is no significant improvement by applying MIO to *menu-api* as it deals with databases which we currently lack supporting them in our instrumentation heuristics.

We applied one hour time budget as stopping criterion for the search algorithms, but the outcome did not indicate any significant difference.

6 Threats to Validity

Conclusion Validity. This study is in the context of SBST. To consider randomness of the search algorithm, our experiment was repeated 10 times for avoiding results obtained by chance. With the results, we employed statistical analysis methods, i.e., Mann-Whitney-Wilcoxon U-tests (*p-value*) and Vargha-Delaney effect sizes (\hat{A}_{12}), for drawing the conclusion.

Internal Validity. It is hard to guarantee that there is no bug in our implementation. However, we have made our implementation and case studies available online (i.e., on GitHub and Zenodo) that allows anyone to review and replicate this study.

External Validity. This study was conducted with two artificial .NET REST APIs and one open-source .NET API. To better generalize our results, there is a need to involve more case studies. However, in the context of REST API testing, there is only few open-source projects that are available. This makes difficult to find more case studies when conducting this kind of experiments. At any rate, our results clearly show the need to handle SQL databases before expanding such case study, but it is a major engineering and research endeavor. Our techniques could be used also in other contexts, like for example unit test generation (e.g., EvoSuite for Java [33]). But, without empirical validation, we cannot be sure they would be effective in those contexts as well.

7 Conclusions

.NET/C# is one of the most popular programming languages, widely used in industry for building cloud-based and internet-connected applications. However, to the best of our knowledge, there does not exist any SBST technique in the literature for automating white-box testing of .NET/C# programs.

In this paper, we developed a .NET bytecode instrumentation to apply existing white-box SBST heuristics based on *branch distance*. With such techniques, we could enable runtime coverage collection and provide effective guidance to search for testing of C# applications, replicating existing SBST success stories for Java and JavaScript programming languages.

We integrated our novel techniques as an extension to the open-source tool EVOMASTER. We conducted experiments with three .NET RESTful APIs. The results yielded by two of these experiments show that our approach achieves significantly better performance than a grey-box random testing technique. However, based on the results by one of the case studies which handles a database, our approach does not perform better than random testing. In addition, with a further analysis on code coverage achieved by the generated tests, we found that our approach is quite effective at solving numerical and string related branches. It achieves line coverage between 67% (at least) and 98% (at most), among the 10 repetitions on two of the case studies.

In theory, any application that is being converted into CIL code can use the instrumentation component. As it is based on EvoMaster, which is dependent on OpenAPI schema, our proposed approach has only been empirically tested for REST APIs. It is difficult to determine right now without adequate empirical information how it could perform on other types of applications. Our solution does not directly advance black-box testing; instead, it only concentrates on white-box testing. The ability to produce test cases in C#, if necessary, for black-box testing is still of potential value. However, rather than being a scientific novelty, that would be more of a technical/usability improvement.

As this is the first work in the literature on the application of white-box SBST for .NET/C# applications, more needs to be done (e.g., effective support for databases) to be able to scale these techniques to large industrial systems. Another possible future work can be to try to improve effectiveness of generating test inputs by the aid of machine learning techniques [45] for inferring potential relationships among actions and parameters of REST APIs. Still, this work provides the important first initial scientific steps toward such direction. Furthermore, all our code implementation is available on GitHub [16] and Zenodo [38], which can be used as bootstrap for other applications of SBST techniques for .NET applications besides web services like RESTful APIs.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 864972).

Declarations

Funding. This work is supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 864972).

Competing interests. The authors have no competing interests to declare that are relevant to the content of this article.

Conflict of interest. The authors declared that they have no conflict of interest.

Data availability. All data used to conduct the experiment (i.e., source code of `EVOMASTER.NET` with .NET bytecode instrumentation, statistics of generated tests, scripts for deploying the experiment and analyzing results) is available on GitHub (www.evomaster.org).

Authors’ contributions. All authors contributed to the writing of the main manuscript. The first draft of the manuscript was written by Amid Golmohammadi and Man Zhang. Then Andrea Arcuri updated and commented on previous versions of the manuscript. All authors reviewed the manuscript and approved the final version.

References

- [1] The State Of The OCTOVERSE. <https://octoverse.github.com/>
- [2] Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* **45**(1), 11 (2012)
- [3] Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pp. 94–105 (2016). ACM
- [4] Arcuri, A., Galeotti, J.P., Marculescu, B., Zhang, M.: Evomaster: A search-based system test generation tool. *Journal of Open Source Software* **6**(57), 2153 (2021)
- [5] Arcuri, A.: EvoMaster: Evolutionary Multi-context Automated System Test Generation. In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2018). IEEE
- [6] Belhadi, A., Zhang, M., Arcuri, A.: Evolutionary-based Automated Testing for GraphQL APIs. In: *Genetic and Evolutionary Computation Conference (GECCO)* (2022)
- [7] Arcuri, A.: Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **28**(1), 3 (2019)
- [8] Zhang, M., Belhadi, A., Arcuri, A.: Javascript instrumentation for search-based software testing: A study with restful apis. In: *IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2022). IEEE
- [9] Arcuri, A.: Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* **104**, 195–206 (2018)
- [10] Arcuri, A.: Automated black-and white-box testing of restful apis with evomaster. *IEEE Software* **38**(3), 72–78 (2020)
- [11] Zhang, M., Arcuri, A.: Open problems in fuzzing restful apis: A comparison of tools. *arXiv preprint arXiv:2205.05325* (2022)
- [12] Kim, M., Xin, Q., Sinha, S., Orso, A.: Automated Test Generation for REST APIs: No Time to Rest Yet. *arXiv* (2022). <https://doi.org/10.48550/ARXIV.2204.08348>. <https://arxiv.org/abs/2204.08348>
- [13] Alshraideh, M., Bottaci, L.: Search-based software test data generation

- for string data using program-specific search operators. *Software Testing, Verification, and Reliability* **16**(3), 175--203 (2006). <https://doi.org/10.1002/stvr.v16:3>
- [14] Zhang, M., Marculescu, B., Arcuri, A.: Resource-based test case generation for restful web services. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1426--1434 (2019)
 - [15] Arcuri, A., Galeotti, J.P.: Handling sql databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **29**(4), 1--31 (2020)
 - [16] EvoMaster. <https://github.com/EMResearch/EvoMaster>
 - [17] Zhang, M., Arcuri, A., Li, Y., Xue, K., Wang, Z., Huo, J., Huang, W.: Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. *arXiv* (2022). <https://doi.org/10.48550/ARXIV.2208.03988>. <https://arxiv.org/abs/2208.03988>
 - [18] Garousi, V., Pfahl, D., Fernandes, J.M., Felderer, M., Mäntylä, M.V., Shepherd, D., Arcuri, A., Coşkunçay, A., Tekinerdogan, B.: Characterizing industry-academia collaborations in software engineering: evidence from 101 projects. *Empirical Software Engineering* **24**(4), 2540--2602 (2019)
 - [19] Arcuri, A.: An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* **23**(4), 1959--1981 (2018)
 - [20] Garousi, V., Felderer, M.: Worlds apart: a comparison of industry and academic focus areas in software testing. *IEEE Software* **34**(5), 38--45 (2017)
 - [21] Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K.: What industry wants from academia in software testing?: Hearing practitioners' opinions. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pp. 65--69 (2017). ACM
 - [22] Garousi, V., Eskandar, M.M., Herkiloğlu, K.: Industry--academia collaborations in software testing: experience and success stories from canada and turkey. *Software Quality Journal*, 1--53 (2016)
 - [23] Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful REST API fuzzing. In: *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 748--758 (2019)
 - [24] Viglianisi, E., Dallago, M., Ceccato, M.: Resttestgen: Automated black-box testing of restful apis. In: *IEEE International Conference on Software*

Testing, Verification and Validation (ICST) (2020). IEEE

- [25] Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: Automated Black-Box Testing of RESTful Web APIs. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 682–685. ACM, ??? (2021)
- [26] Wu, H., Xu, L., Niu, X., Nie, C.: Combinatorial testing of restful apis. In: ACM/IEEE International Conference on Software Engineering (ICSE) (2022)
- [27] Laranjeiro, N., Agnelo, J., Bernardino, J.: A black box tool for robustness testing of rest services. *IEEE Access* **9**, 24738–24754 (2021)
- [28] Hatfield-Dodds, Z., Dygalo, D.: Deriving semantics-aware fuzzers from web api schemas. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 345–346 (2022). IEEE
- [29] .NET Platform. <https://github.com/dotnet>
- [30] ECMA-335, Common Language Infrastructure (CLI). <https://www.ecma-international.org/publications-and-standards/standards/ecma-335/>
- [31] Lehre, P.K., Yao, X.: Runtime analysis of (1+1) ea on computing unique input output sequences. In: IEEE Congress on Evolutionary Computation (CEC), pp. 1882–1889 (2007)
- [32] Korel, B.: Automated software test data generation. *IEEE Transactions on software engineering* **16**(8), 870–879 (1990)
- [33] Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 416–419 (2011)
- [34] Panichella, S., Gambi, A., Zampetti, F., Riccio, V.: Sbst tool competition 2021. In: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), pp. 20–27 (2021). IEEE
- [35] Tillmann, N., N. de Halleux, J.: Pex --- white box test generation for .NET. In: TAP’08: International Conference on Tests And Proofs. LNCS, vol. 4966, pp. 134–253. Springer, ??? (2008)
- [36] Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
- [37] Arcuri, A., Galeotti, J.P.: Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software*

Engineering and Methodology (TOSEM) **31**(1), 1--34 (2021)

- [38] Arcuri, A., ZhangMan, asmab89, Bogdan, Golmohammadi, A., Galeotti, J.P., Seran, López, A.M., Aldasoro, A., Panichella, A., Niemeyer, K.: EMResearch/EvoMaster:. <https://doi.org/10.5281/zenodo.6651631>. <https://doi.org/10.5281/zenodo.6651631>
- [39] Arcuri, A., Briand, L.: Adaptive random testing: An illusion of effectiveness? In: ACM Int. Symposium on Software Testing and Analysis (ISSTA), pp. 265--275 (2011)
- [40] Zhang, M., Arcuri, A.: 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)
- [41] EvoMaster Benchmark (EMB). <https://github.com/EMResearch/EMB>. Online, Accessed May 20, 2022
- [42] Arcuri, A., Galeotti, J.P.: Enhancing Search-based Testing with Testability Transformations for Existing APIs. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1--34 (2021)
- [43] Arcuri, A., Briand, L.: A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. Software Testing, Verification and Reliability (STVR) **24**(3), 219--250 (2014)
- [44] JetBrains Rider. <https://www.jetbrains.com/rider>
- [45] Mirabella, A.G., Martin-Lopez, A., Segura, S., Valencia-Cabrera, L., Ruiz-Cortés, A.: Deep learning-based prediction of test input validity for restful apis. In: 2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest), pp. 9--16 (2021). IEEE