White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study

MAN ZHANG, Kristiania University College, Oslo, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Oslo, Norway
YONGGANG LI, YANG LIU, and KAIMING XUE, Meituan, Beijing, China

Remote Procedure Call (RPC) is a communication protocol to support client-server interactions among services over a network. RPC is widely applied in industry for building large-scale distributed systems, such as Microservices. Modern RPC frameworks include, for example, Thrift, gRPC, SOFARPC, and Dubbo. Testing such systems using RPC communications is very challenging, due to the complexity of distributed systems and various RPC frameworks the system could employ. To the best of our knowledge, there does not exist any tool or solution that could enable automated testing of modern RPC-based services. To fill this gap, in this article we propose the first approach in the literature, together with an open source tool, for fuzzing modern RPC-based APIs. The approach is in the context of white-box testing with search-based techniques. To tackle schema extraction of various RPC frameworks, we formulate a RPC schema specification along with a parser that allows the extraction from source code of any JVM RPC-based APIs. Then, with the extracted schema we employ a search to produce tests by maximizing white-box heuristics and newly defined heuristics specific to the RPC domain. We built our approach as an extension to an open source fuzzer (i.e., EvoMaster), and the approach has been integrated into a real industrial pipeline that could be applied to a real industrial development process for fuzzing RPC-based APIs. To assess our novel approach, we conducted an empirical study with two artificial and four industrial web services selected by our industrial partner. In addition, to further demonstrate its effectiveness and application in industrial settings, we report results of employing our tool for fuzzing another 50 industrial APIs autonomously conducted by our industrial partner in their testing processes. Results show that our novel approach is capable of enabling automated test case generation for industrial RPC-based APIs (i.e., 2 artificial and 54 industrial). We also compared with a simple gray-box technique and existing manually written tests. Our white-box solution achieves significant improvements on code coverage. Regarding fault detection, by conducting a careful review with our industrial partner of the tests generated by our novel approach in the selected four industrial APIs, a total of 41 real faults were identified, which have now been fixed. Another 8,377 detected faults are currently under investigation.

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

Additional Key Words and Phrases: Mircoservices, RPC, fuzzing, test generation, SBST, gRPC, Thrift

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement No. 864972).

Authors' addresses: M. Zhang, Kristiania University College, Kirkegata 24-26, 0153 Oslo, Norway; email: man.zhang@kristiania.no; A. Arcuri, Kristiania University College, Kirkegata 24-26, 0153, and Oslo Metropolitan University, Oslo, Norway; email: andrea.arcuri@kristiania.no; Y. Li, Y. Liu, and K. Xue, Meituan, Floor 8, Block D, Dingcheng Times Square, Datun Street, Chaoyang District, Beijing, 100012, China; emails: {liyonggang, liuyang352, xuekaiming}@meituan.com.

1049-331X/2023/07-ART122 $15.00
https://doi.org/10.1145/3585009

ACM Reference format:
Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 122 (July 2023), 38 pages.
https://doi.org/10.1145/3585009

1 INTRODUCTION

It is a common practice in industry to develop large enterprise systems with microservice architectures [59, 68, 79]. For example, Meituan is a large e-commerce enterprise with more than 630 millions customers in China, with microservice systems like Meituan Select comprising more than 1,000 different web services. Testing this kind of systems is very complex, due to their distributed nature and access to external services such as databases. There is a dire need in industry for automation for this kind of systems.

Although in recent years there has been an interest in the research community on fuzzing REST web services [45] (e.g., with tools like Restler [29], RestTestGen [67], Restest [58], RestCT [69], bBOXRT [51], and Schemathesis [47]), to the best of our knowledge there is no work in the literature on the testing of modern **Remote Procedure Call** (**RPC**) web services. There is a large body of knowledge in the scientific literature on the topic of software testing automation, with several successful stories in many different software testing domains [14, 35, 44]. Addressing an important industrial testing problem for the first time does not start from scratch, especially when aiming at providing useful results for engineers in industry. It rather builds on top of decades of scientific research on the topic. On the one hand, some research challenges are similar to other domains (e.g., how to deal with SQL databases when fuzzing a web service [24], regardless of whether it is a REST, GraphQL, or RPC-based API). On the other hand, scientific research and empirical evaluations are needed to address the specific peculiarities of each different software testing problem. For example, to the best of our knowledge, none of the existing fuzzers in the scientific literature can be directly applied on fuzzing RPC systems without major engineering and scientific effort, as, for example, the API schemas and communication protocols are different.

As part of an industry-driven collaboration [18, 40–43], when we first tried to use our EvoMaster fuzzer [17] for RESTful APIs on the web services developed at Meituan, we could not apply it directly [74]. We had to manually write REST APIs as wrappers for the RPC systems (which use Apache Thrift). Not only it is time-consuming, but also the generated tests are more difficult to use for debugging any found fault. Two web services were used as a case study. Such study (with interviews and questionnaires among the developers at Meituan) pointed out few research challenges, including the need for a native support for RPC systems for web service fuzzers. Such support not only requires not trivial engineering effort (our extension to the existing fuzzer EvoMaster required more than 10,000 lines of code, not including test cases), but also there are several scientific research challenges that need to be addressed to best handle RPC-based APIs (as we will discuss in more detail later in the article).

In this article, we provide a novel approach[1] to automatically fuzz RPC-based APIs, built on top of EvoMaster. To adapt to various RPC frameworks, in this approach, a RPC schema is defined to formulate the API specification that could document all necessary info to make a RPC call and possible responses (e.g., throwing exception, failure). The schema of the RPC-based services can be automatically extracted from the source code with our approach. This allows one to test the

[1]EvoMaster is open source, and it is available at www.evomaster.org. A replication package for this study is available at https://github.com/anonymous-authorxyz/fuzzing-rpc. Accessed August 26, 2022.

services developed with different RPC frameworks. With the extracted schema, a test for a RPC-based API can be reformulated as an *individual*, i.e., a sequence of RPC calls under a certain state of the API (e.g., database if it has). Thus, search-based techniques (such as the MIO algorithm [19]) can be employed to evolve tests (e.g., seek various values of input parameters of RPC calls in order to cover more code and find more faults). To better solve our testing problems with search, we define new heuristics specific to the RPC domain. The approach was implemented as an extension of EvoMaster and has been integrated into an industrial pipeline. To assess the effectiveness of our novel approach and its application on industrial context, we empirically compared it with a gray-box technique with two artificial and four industrial RPC-based APIs, and further reported its performances on 50 industrial APIs in real industrial settings. The main contributions of the article include:

(1) the first approach in the literature for fuzzing RPC-based APIs;
(2) an open source tool support (i.e., an extension to the existing fuzzer EvoMaster);
(3) an empirical study carried out in industrial settings that involves in total 54 industrial RPC-based APIs comprising 1,489,959 lines of code (computed with JaCoCo) for business logic;
(4) an in-depth analysis on four selected industrial APIs with our industrial partner; and
(5) identifying lessons learned and research challenges that must be addressed before better results can be obtained.

The article is organized as follows. Section 2 provides the needed background information to better understand the rest of the article. Section 3 analyzes related work. The details of our novel approach are presented in Section 4. Our empirical study is discussed in Section 5, followed by lessons learned in Section 6. Threats to validity are discussed in Section 7. Finally, we conclude the article in Section 8.

2 BACKGROUND

2.1 RPC

RPC enables one to call methods in other processes, possibly on a different machine, communicating over a network. This is a common practice in distributed systems, particularly in microservice architectures [59]. There are different frameworks to develop RPC-based APIs, like, for example, Apache Thrift [13] (originally from Facebook), Apache Dubbo [2] (originally from Alibaba), gRPC [6] (from Google), and SOFARPC [10] (from Alibaba). All these popular frameworks were developed to address the scale of large distributed systems. Compared to other types of web services (e.g., RESTful APIs), RPC-based APIs aim at optimizing performance at the cost of stronger coupling between client and server applications (there is no silver bullet). For example, given a schema for the API (e.g., a `.thrift` file for Thrift or a `.proto` file for gRPC), a compiler is used to create a server application (which then can be extended with the business logic of the API) and a client library. A process that wants to communicate with the server API must include this client library, and use these client stubs to remotely call the API in the server process. For example (*rpc1* in Figure 1), a client process (in the service `A`) would have a reference to a class-stub `B`, and, every time that `stubB.bcd()` is called, then the client library will make a network call to execute `B.bcd()` on the server API.

The actual communications between the client and the server depend on the framework implementation, e.g., typically HTTP/2 using Protobuf for gRPC, but it can be configured to use other protocols. Both Thrift and gRPC support the generation of client/server code in different languages (e.g., Java, C# and JavaScript), whereas SOFARPC and Dubbo support only Java.

Figures 2 and 3 represent examples of a schema specified with different frameworks (i.e., Thrift and gRPC), and snippets of code of client-stub classes and server classes (e.g., interface/abstract

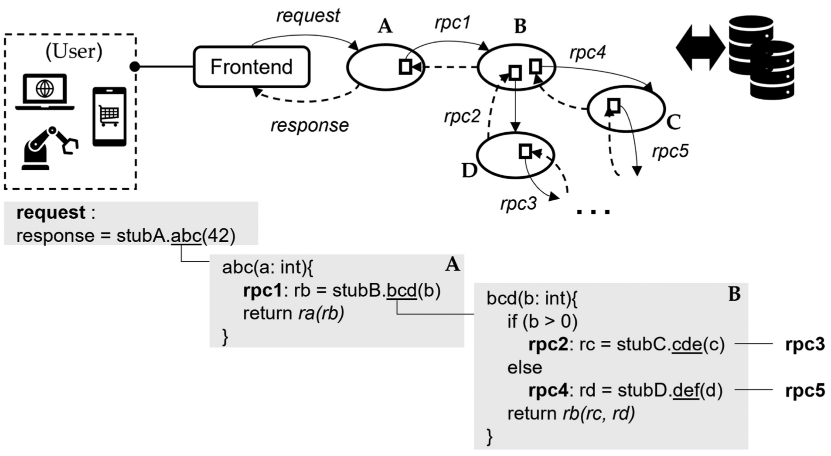


Fig. 1. RPC-based APIs in microservices.

classes) to implement/extend that are automatically generated by the framework based on the schema. As shown in Figure 2(a) for Thrift and Figure 3(a) for gRPC, the two schemas for `NcsService` have the same function, i.e., `bessj`, that evaluates Bessel function by taking one `integer` and one `double` numbers as inputs, then returning a **Data Transfer Object** (**`DTOs`**), which comprises the same fields. Based on the schemas, the compiler could automatically generate source code of client libraries, such as `Client` class at line 13 in Figure 2(b) and `NcsServiceBlockingStub` class at line 22 in Figure 3(b). Then, with such client libraries, the functions of RPC-based API could be accessed. For instance, snippet code shown in Figure 2(c) represents an example of a test for the `NcsService` implemented with Thrift framework. Lines 4 and 6 in Figure 2(c) represent how to instantiate a client to access the service, such as a URL with *http://localhost:8080/ncs* and an accepted protocol to perform communications between client and service as `TBinaryProtocol`. Line 12 is to make a network call to `bessj` function with the instantiated client, then receive a response, and lines 14 and 15 show assertions on the response.

Besides the client-stub classes, the compiler also generates the source code that users could extend for implementing business logic of the services. In this example, with Java language, Thrift outputs `Interface` class, which comprises a list of methods to implement, and each method corresponds to a RPC function (see lines 7–9 in Figure 2(b)). For gRPC, it is similar that the gRPC compiler outputs `abstract class`, which comprises a list of methods to extend (see lines 13–18 in Figure 3(b)). As the examples, such methods in the classes define specifications about how to access the services. In addition, a schema specification (e.g., `.thrift` file) might not be always available for a RPC-based API, and the service could be initially defined with programming language, such as SOFARPC[2] and Dubbo[3] with Java `interface` classes. To fuzz RPC-based APIs, extracting specifications to access the API is a prerequisite. The various RPC frameworks allow *abstraction* classes (i.e., `Interface` and `abstract class` in Java) to define RPC-based APIs (we refer to the classes that define RPC-based APIs as *RPCInterface*s in later sections). Thus, if we could enable an extraction of the specification based on such *RPCInterface*s, it would generalize the application of the approach for fuzzing the APIs with different RPC frameworks (as we propose in this article).

[2]https://www.sofastack.tech/en/projects/sofa-rpc/getting-started-with-sofa-boot/. Accessed August 26, 2022.
[3]https://github.com/apache/dubbo. Accessed August 26, 2022.

```
namespace java org.thrift.ncs

struct Dto {
  1: i32 resultAsInt,
  2: double resultAsDouble
}

service NcsService {

  Dto bessj(1:i32 n, 2:double x)
  ...
}
```

(a) Snippet of an example of RPC schema (i.e., `ncs.thrift`) specified with Thrift

```
package org.thrift.ncs.client;

@SuppressWarnings({"cast", "rawtypes", "serial", "unchecked", "unused"})
@javax.annotation.Generated(value = "Autogenerated by Thrift Compiler (0.15.0)", date = "
     2021-10-21")
public class NcsService {

  public interface Iface {

    public Dto bessj(int n, double x) throws org.apache.thrift.TException;
    ...
  }

  public static class Client extends org.apache.thrift.TServiceClient implements Iface {

    public Dto bessj(int n, double x) throws org.apache.thrift.TException
    {
      send_bessj(n, x);
      return recv_bessj();
    }
    ...
  }
}
```

(b) Snippet of an example of an interface generated with Thrift framework

```
@Test(timeout = 60000)
public void test() throws Exception {

  TTransport transport = new THttpClient("http://localhost:8080/ncs");
  TProtocol protocol = new TBinaryProtocol(transport);
  NcsService.Client client = new NcsService.Client(protocol);

  org.thrift.ncs.client.Dto res_1 = null;
  {
    int arg0 = 577;
    double arg1 = 0.20491354575856158;
    res_1 = client.bessj(arg0,arg1);
  }
  assertEquals(0, res_1.resultAsInt);
  assertTrue(numbersMatch(0.0, res_1.resultAsDouble));
}
```

(c) An example of a JUnit test for NcsService implemented with Thrift (see Figure 2(b))

* note that complete versions of the schema and its implementation with Thrift framework can be found at https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rpc/thrift/artificial/thrift-ncs

Fig. 2. An example of RPC schema, its automatically generated source code with Thrift framework and a test for the RPC-based service.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.grpc.ncs.generated";

service NcsService {

  rpc bessj(BessjRequest) returns (DtoResponse) {}
  ...
}

message BessjRequest{
  int32 n = 1;
  double x = 2;
}

message DtoResponse {
  int32 resultAsInt = 1;
  double resultAsDouble = 2;
}
```

(a) Snippet of an example of RPC schema (i.e., ncs.proto) specified with gRPC

```
package org.grpc.ncs.generated;

import static io.grpc.MethodDescriptor.generateFullMethodName;

@javax.annotation.Generated(
value = "by gRPC proto compiler (version 1.41.0)",
comments = "Source: ncs.proto")
@io.grpc.stub.annotations.GrpcGenerated
public final class NcsServiceGrpc {

  public static final String SERVICE_NAME = "NcsService";

  public static abstract class NcsServiceImplBase implements io.grpc.BindableService {

    public void bessj(org.grpc.ncs.generated.BessjRequest request,
    io.grpc.stub.StreamObserver<org.grpc.ncs.generated.DtoResponse> responseObserver) {
      io.grpc.stub.ServerCalls.asyncUnimplementedUnaryCall(getBessjMethod(), responseObserver)
     ;
    }
    ...
  }

  public static final class NcsServiceBlockingStub extends io.grpc.stub.AbstractBlockingStub<
     NcsServiceBlockingStub> {

    public org.grpc.ncs.generated.DtoResponse bessj(org.grpc.ncs.generated.BessjRequest
     request) {
      return io.grpc.stub.ClientCalls.blockingUnaryCall(
      getChannel(), getBessjMethod(), getCallOptions(), request);
    }
    ...
  }
  ...
}
```

(b) Snippet of an example of an interface generated with gRPC framework

* note that complete versions of the schema and its implementation with gRPC framework can be found at https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rpc/grpc/artificial/grpc-ncs

Fig. 3. An example of RPC schema and its automatically generated source code with gRPC framework.

Moreover, in the context of microservice architectures, as shown in Figure 1, the microservices comprise of a set of connected RPC-based APIs, and the API could have multiple stub-classes of other direct interacted APIs (e.g., B has stub classes of the services C and D). Processing a request from the user typically involves multiple APIs. With different inputs in the request, it could result in various sequences of RPC calls with different APIs. For instance, assume that a user sends a request that results in a function call to abc of the service A. In order to provide a response to the user, it needs to involve multiple APIs (e.g., B, C, and D) that could trigger various sequences of RPC communications, e.g., *rpc1* → *rpc2* → *rpc3* or *rpc1* → *rpc4* → *rpc5* as shown in Figure 1. Note that this example illustrates communications among RPC-based services for processing one request. To test one RPC-based API (e.g., B is the **System Under Test** (**SUT**)), C and D could be considered as external services of B, and the test for the API B would consist of a sequence of network calls to the SUT as the example shown in Figure 2(c).

2.2 Automated System Testing

In the scientific literature, there has been a lot of work on the automation of software testing [35]. Different techniques have been investigated, like search-based algorithms [14] and symbolic execution [31].

System testing refers to the testing of applications as a whole, using the same input interfaces as the actual users. The process of generating test cases to find errors in these systems (e.g., typically a crash) is often referred to with the term *fuzzing* [44].

When dealing with the fuzzing of Web APIs, there are two main types of testing: *black-box* and *white-box* testing. The difference is that in white-box testing there is access to the internal details of the API, like its source code or bytecode. This information can be exploited to design heuristics in order to improve the testing process, e.g., to increase metrics like code coverage. This can lead to higher fault detection, as a fault cannot be triggered if its code is never executed. On the other hand, in black-box testing, the API is treated as a black box, with no info on its internal details.

Black-box testing is of more general application, as the API could be written in any programming language and running on a remote machine. As white-box testing requires one to analyze the source code, not only the API needs to be run locally, but also there are restrictions on the programming language it is implemented with. Code analyses and white-box heuristics require complex engineering effort, and supporting several programming languages in the same tool might not be viable. This also complicates scientific comparisons. For example, it would not be viable to directly compare a fuzzer for C++ with one for Java.

Even in the cases in which the API is treated as a black box, some general information is required to be able to fuzz it. Typically, a specification (also known as *schema*) is used to determine the type of inputs to send to the API. Sending random bytes over a TCP connection would unlikely result in any meaningful message that the API will not discard immediately. The type of schema will depend on the type of API, e.g., OpenAPI [9] for RESTful APIs. Alternatives are to use existing test cases or replayable traffic messages as a starting point, and then do small modifications to them to see how the API behaves.

Another common term used in the literature is *gray-box* testing. This usually refers to a mix of black and white box testing, where only partial information on the internal details of the API is available. At times, this term is also used to specify when only *lightweight* code instrumentations are applied to the tested API. A lightweight instrumentation could be just measuring code coverage, for example to drive the fuzzer to focus on the least covered parts of the API. A lightweight instrumentation is much easier to implement and start to experiment with compared to a full-blown search-based or symbolic execution approach.

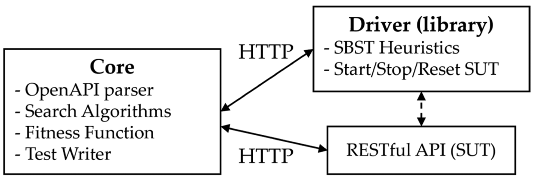


Fig. 4. Architecture of EvoMaster.

2.3 EvoMaster

EvoMaster is an open source tool for fuzzing enterprise applications with search-based techniques, in the context of both white-box and black-box testing [17, 21, 27]. To enable white-box testing, the tool is composed of two components, i.e., *driver* and *core*, as shown in Figure 4. The *driver* is responsible for collecting **Search-Based Software Testing** (**SBST**) heuristics with code instrumentation (currently targeting JVM [19] and JavaScript [75]) and controlling the SUT (i.e., start/stop/reset). It is implemented as a library, so that it is easy to be applied by the SUT using, for example, Maven and Gradle. The *core* encompasses the main functionality of generating test cases with search-based techniques, e.g., various search algorithms and fitness function.

To generate more effective white-box tests for enterprise APIs, EvoMaster is equipped with a set of novel techniques. For instance, *Boolean flag* is a common problem in handling white-box testing with search, i.e., no gradients for search to solve a constraint that is either `true` or `false`. To enable fitness gradients for such problems in the source code, EvoMaster is integrated with *testability transformations* [25, 26]. This enables *branch distance* computations for such flag problems by transforming source code (e.g., via *replacement* methods) with code instrumentation. The *replacement* methods also track inputs (referred to as *taint analysis*) for providing additional information to the search for solving this kind of problem. In addition, enterprise APIs typically interact with databases. A database with various data would represent various states of the SUT. To test the APIs with various states, *SQL handling* [23, 24] was developed in EvoMaster that can extract SQL queries and calculate heuristics for the queries at runtime. Then, with these heuristics, EvoMaster can directly generate data into the database (with SQL commands such as `INSERT`). REST is one of the popular architectural styles for building web services. To better support it, EvoMaster employs a set of techniques designed in particular for the REST domain, e.g., OpenAPI parser, smart sampling, test reformulation for REST APIs [20], *resource- and dependency-based strategies* [72, 77].

Furthermore, EvoMaster is enhanced with *adaptive hypermutation* [71]. It is as an advanced search mutator for handling long and structured chromosomes, like the tests for REST APIs that comprise a set of `INSERT` commands and a sequence of HTTP requests with query parameters and body payloads (e.g., JSON objects).

To serve as a SBST fuzzer, EvoMaster includes the implementation of different search algorithms, i.e., MOSA [61], WTS [39, 63], and Random. MIO [15, 19] is a search algorithm that was designed specifically for system test generation in the context of white-box testing. MIO has been empirically studied by comparing with the other existing work (e.g., WTS, MOSA and Random), using artificial and real case studies. Results showed that MIO achieved the overall best performance [15, 19] when applied on the problem of fuzzing RESTful APIs.

3 RELATED WORK

To the best of our knowledge, there does not exist any technique for fuzzing modern RPC-based APIs (e.g., using frameworks like Apache Thrift, gRPC, and SOFARPC). In addition, EvoMaster seems the only open source tool that supports white-box testing for Web APIs, and it gives the overall best results in recent empirical studies comparing existing fuzzers for REST APIs [49, 73]. However, currently EvoMaster only supports fuzzing RESTful APIs [20] and GraphQL APIs [33].

In the literature, there has been work on the fuzzing of other kinds of web services. The oldest approaches deal with black-box fuzzing of SOAP [36] web services, such as, for example, [30, 32, 46, 52, 53, 56, 60, 64, 65, 70]. SOAP is a type of RPC protocol. However, SOAP's reliance on XML format for schema definitions and message encoding has led this protocol to lose most of its market share in industry (i.e., apart from maintaining legacy systems, it is not used so much anymore for new projects).

In recent years, there has been a large interest from the research community in testing RESTful APIs [37, 45], which are arguably the most common type of web services. Several tools for fuzzing RESTful APIs have been developed in the research community, such as, for example (in alphabetic order), bBOXRT [51], EvoMaster [16], RESTest [57], RestCT [69], RESTler [29], and RestTestGen [67]. Another recently introduced type of web services is GraphQL [5], which is gaining momentum in industry. However, there is only little work in academia on the automated testing of this kind of web services [33, 34, 48, 66].

The automated testing of different kinds of web services (e.g., modern-RPC, SOAP, REST, and GraphQL), shares some common challenges (e.g., how to define white-box heuristics on the source code of the SUT, and how to deal with databases and interactions with external services). However, there are as well specific research challenges for each type of web service, as we will show later in the article. A fuzzer for SOAP or REST APIs would not be directly applicable to a RPC web service, and vice versa.

In the literature, there are many applications of scientific research on the automation of software testing [14, 35, 44]. Popular examples are AFL [1] for parsers and Sapienz for mobile applications [54]. Albeit possible, extending these kinds of tools from other testing domains for fuzzing RPC APIs would likely require major engineering and scientific effort. Other domains like fuzzing network protocols (e.g., AFLNet [62]) and network devices (e.g., NDFuzz [78]) are closer to the fuzzing of Web APIs. Still, a non-trivial amount of work would be needed to adapt them to white-box fuzzing of RPC-based APIs. For example, this could also explain why, to the best of our knowledge, none of these existing tools has been used so far to fuzz RESTful APIs, albeit their recent popularity in academia.

Given a client library for a RPC-based API, a unit test generator could be used directly on it, such as, for example, the popular EvoSuite [38] for Java classes. This might work if the SUT and the client library are run in the same JVM. However, all the issues when dealing with system testing of web services would still be there, e.g., how to deal with databases and what to use as test oracle. Also, likely such unit testing tool would need some modifications (e.g., to collect coverage from all the classes and not just the RPC-client one). Therefore, how a unit test generator could be adapted and fare in such a system testing scenario is an open research question.

4 FUZZING RPC-BASED APIs

When addressing a new testing problem like the fuzzing of RPC-based APIs, several design decisions need to be made, especially when using search-based techniques. There is the need to specify the *search space* (Section 4.1), how to represent the *genotype* of an evolving individual (i.e., a test case in this context) (Section 4.2), how to define the *fitness function* to guide its evolution (Section 4.3), which *search operators* to employ to modify the evolving individuals (Section 4.4), and how to *output* the final results to the user (Section 4.5).

Building a fuzzer that can scale and be used on tens of industrial systems requires major engineering efforts, throughout a few years. To evaluate the novel techniques presented in this article,

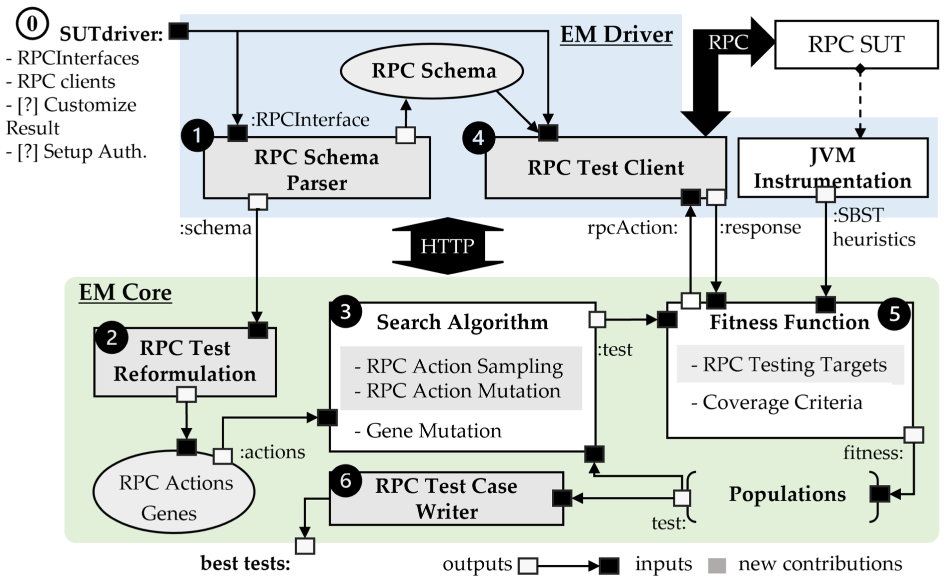


Fig. 5. Overview of the approach built with EvoMaster.

we did not start from scratch, but rather re-use and extend an existing open source fuzzer. In particular, our novel approach is built on top of EvoMaster (recall Section 2.3).

Figure 5 represents an overview of our novel approach. In order to fuzz RPC-based APIs, we propose *RPC Schema* specification, which formulates necessary info to allow the execution of RPC function calls and the analysis of execution result. In addition, with the specification, as shown in the figure, the approach is composed of six steps distributed between the *driver* and *core* of EvoMaster, plus initial settings manually provided by the user, for enabling automated fuzzing of RPC-based APIs with search techniques. We briefly summarize these steps, where their details will be provided in the rest of this section.

To employ EvoMaster, a *SUTdriver* is required to be specified for implementing how to start/stop/reset the SUT (recall Section 2.3). In the context of RPC-based API testing, in the *SUTdriver*, we further need the user to specify (1) *RPCInterfaces*: what interfaces are defining the API in the SUT with their class names and (2) *RPC clients*: the corresponding client instances used to make RPC calls during test generation (Step 0). Then, with the specified interface info, *RPC Schema Parser* will extract and identify the API schema based on proposed *RPC Schema* specification, in order to access the RPC functions (Step 1). At the *core* side, the extracted schemas will be further reformulated (Step 2) to be as components (i.e., *RPC Actions* and *Genes*) of the search for producing tests (Step 3). In our approach, a generated test is evaluated by its execution on the SUT (Step 4) performed on the *driver* side. Then, responses, SBST heuristics (e.g., code coverage with code instrumentation), and identified potential faults resulted in the execution will be returned to the *Fitness Function* (Step 5) for calculating the fitness value of the executed test. Producing and evaluating tests are performed iteratively (i.e., Steps 3–5), within a given search budget. At the end of the search, a set of the best (in terms of code coverage and fault detection) tests for the RPC-based SUT will be outputted (Step 6) with a given format (e.g., JUnit 5).

4.1 Search Space

At a high level, a RPC-based API can be seen as a process that opens a TPC/UDP port on a given host, and then replies to incoming messages formatted with a given application-layer protocol. Such protocol could vary among the different RPC implementations. Furthermore, the API would reply only to requests for its defined methods, requiring the right number and type of input parameters. This means that sending random bytes over the TCP/UDP connections would unlikely result in any meaningful response from the API, and possibly no execution of the code of its business logic.

To address this issue, it would be important for a fuzzer to send well-formatted messages for the different remote APIs exposed by the web service. Given a *schema* that specifies which methods can be called, a fuzzer can then generate calls with the right input parameters. Considering that these methods can take as inputs complex data such as strings, objects, and arrays, the search space of possible inputs is huge, even when using a schema to constrain what will be sent. Only with some specific inputs, faults could be revealed and code coverage optimized. Furthermore, to test a specific endpoint, there might be the need to call a previous one to set the state (e.g., a database) of the API. This means that a test case would hence be a sequence of one or more remote calls toward the API, which increases the search space even further. To complicate this even further, to achieve higher code coverage the API might require setting up the *environment* in which it operates. For example, advance fuzzers can also add data directly into SQL databases as part of an initialization phase, based on what queries the API executes on the database. This further extends the search space of possible test cases that the fuzzers need to explore.

Nowadays, there exist various RPC frameworks for building modern RPC-based APIs, e.g., Thrift [13], gRPC [6], Dubbo [2], and SOFARPC [10]. As discussed in Section 2.1, most of the techniques would result in *RPCInterface*s (e.g., implemented as `interface` or `abstract class`) in their API implementations representing how the services can be accessed, together with a client stub to make the actual RPC calls. Considering all the possible types of communication protocols supported by the different RPC frameworks, calling a RPC API directly from a fuzzer would be a major technical endeavor. Furthermore, it would require one to support the different schema languages for each framework, such as, for example `.thrift` (see Figure 2(a)) and `.proto` (see Figure 3(a)) formats, and there would be limitations when the schema file might not be available, such as the APIs implemented with SOFARPC and Dubbo.

In order to enable automated testing of RPC-based APIs in a more generic way, in this article we propose a schema specification specific to the RPC domain that formulates main concepts for facilitating invocations of RPC function calls and result analysis. Such a specification can be automatically extracted based on *RPCInterface*s, regardless of which RPC framework is employed by the API. This schema defines the *search space* for the fuzzing, as we will evolve test cases complying with such schema. Then, we employ the actual client libraries of the APIs to make the RPC calls.

4.1.1 RPC Schema Specification. Our RPC Schema is defined with a **Data Transfer Object** (**DTO**), which can then be instantiated in different formats, such as, for example, JSON. Figure 6 shows our RPC schema specification with a UML class diagram.

To extract info for enabling invocations of RPC function calls, there exist five main concepts to define *RPCInterfaces* (denoted as classes with white background in Figure 6):

— *RPCInterfaceSchemaDto*: it represents the *RPCInterface*, such as the `Interface` with Thrift (see Figure 2(b)) and `abstract class` with gRPC (see Figure 3(b)). A *RPCInterfaceSchemaDto* comprises one or more *RPCActionDto* (see *1..* functions*), a set of functions for authentication handling (see ** authFunctions*) and a set of specifications of data types (see ** types*). For instance, `NcsService.Iface` interface has a `bessj` function and employs `Dto` data structure (as shown in Figure 2(b)). Note that a RPC-based API might have multiple interfaces as industrial APIs, which we studied in this article.

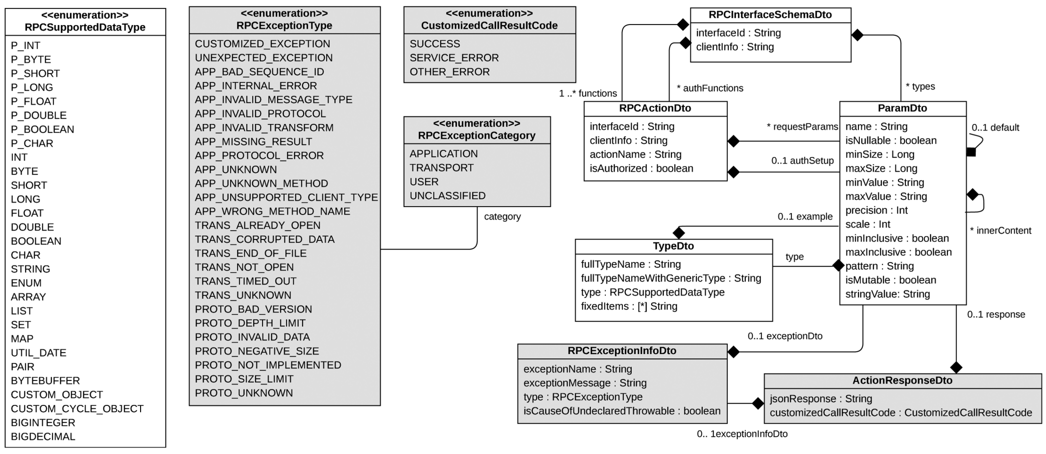


Fig. 6. Data Transfer Objects defined in our RPC schema specification.

— *RPCActionDto*: it captures information to make a RPC function call, i.e., input parameters if they exist (see * *requestParams*) and additional authentication setup (see *0..1 authSetup*). Each *RPCActionDto* also has *interfaceId*, *clientInfo*, and *actionName* properties to identify the RPC function to call. In addition, we identify a property *isAuthorized* representing whether the *RPCActionDto* is restricted with authentications in its implementation.

— *ParamDto*: it is used to describe values of input parameters and return. A *ParamDto* links to an explicit datatype (see *type*) and might be composed of a set of *ParamDto*s for representing complex data types, such as object, collection, and map (see * *innerContent*). The *ParamDto* might be specified with a default value (see *0..1 default*), e.g., a field in a DTO can be assigned with a default value. In addition, we define *stringValue* to assign a value for the input parameter or represent the actual value of the return. Note that *stringValue* is applicable only if there are no any internal elements. To construct constraints of the input parameters if they exist, we define a set of properties in *ParamDto* as follows:
  — *isNullable* represents whether the parameter is nullable to make the call.
  — *isMutable* indicates whether the parameter is mutable. A value of the property is derived based on whether the parameter is assigned with a fixed value. For instance, a parameter must be `true` if it is specified with `@AssertTrue`, thus the parameter is considered as immutable.
  — *minSize* and *maxSize* represent boundaries in size if specified. The constraint could be applicable to data types, i.e., collection, map, array, and char sequence (e.g., string).
  — *minValue* and *minInclusive* are used to represent a minimum value, and a value of the parameter must be higher than or equal to the minimum.
  — *maxValue* and *maxInclusive* are used to represent a maximum value, and a value of the parameter must be lower than or equal to the maximum.
  — *precision* and *scale* capture constraints for numeric values regarding its precision and scale (e.g., number of digits in their decimal part).
  — *pattern* represents a regular expression that a string value must match.

  Such captured constraints could contribute to test data generation for fuzzing Web APIs, by sampling values within the boundaries of these constraints. Values of all of the constraint properties could be derived automatically based on the *PRCInterface*, which is explained in the *RPC Schema extraction* (see Section 4.1.2).

— *TypeDto* and *RPCSupportedDataType* identify the data type info of the *ParamDto*. A list of data types we support is defined as an enumeration *RPCSupportedDataType* that covers the most commonly used data types, i.e., *array*, *byte buffer*, *date*, *enumeration*, *list*, *map*, *set*, *string*, *integer*, *Boolean*, *double*, *float*, *long*, *character*, *byte*, *short*, *big integer*, *big decimal*, and any customized DTO object, for enabling the fuzzing of RPC-based APIs. In *TypeDto*, it can be specified with an example (see *0..1 example*) for representing a generic type of collection, array, and map. Note that this list of supported data types is not meant to be complete for all RPC frameworks. But, if needed, it can be extended.

4.1.2 RPC Schema Extraction and Execution Support. As a white-box fuzzer, besides the source code of SUT, a *SUTdriver* is the only input that EvoMaster needs a user to specify (recall Section 2.3). Then, the *SUTdriver* is employed at the *driver* side for, e.g., starting/stopping/resetting the SUT. In the context of RPC-based API fuzzing, we further need the user to provide info of *RPCInterface*s and corresponding client instances for extracting the API schema and accessing the SUT. As shown in Figure 5, in the *driver*, with the provided *SUTdriver* (Step 0), we developed a *RPC Schema Parser*, by directly extracting the interface definitions (which do represent the API schema) from the source code using a *reflection technique*, such as Java Reflection.[4] Thus, with any RPC framework, if the available RPC functions are defined as an `interface/abstract class` (which is usually the case), our approach could be applicable. The extracted information is further formulated as a generic *RPC Schema* (see Section 4.1), i.e., a *RPCInterface* will be formulated as a *RPCInterfaceSchemaDto* that contains specifications to invoke RPC function calls (i.e., *RPCActionDto* (Step 1 $\rightarrow$ Step 2)). In addition, we developed *RPC Test Client*, which allows one to make a RPC function call against the SUT with *RPCActionDto*, then return *ActionResponseDto* (Step 5 $\leftrightarrow$ Step 4) using specified RPC client instances. The *driver* is implemented as a service using REST API, and the two components (i.e., *RPC Schema Parser* and *RPC Test Client*) are exposed as two HTTP endpoints, i.e., `/infoSUT` for extracting RPC API schema and `/newAction` for executing RPC function calls. Thus, with a provided *SUTdriver*, our *driver* employed with proposed *RPC Schema* would allow a unique interface of our tool to support invocations of RPC functions and result analysis. This is an essential prerequisite for fuzzing RPC-based API.

Note that instead of enabling RPC function execution at the *driver* side, an alternative approach would have been to include the two components and RPC API client library directly into the *core* process, which might be more efficient (as calls from the *core* do not need to go through the *driver* with HTTP requests). But that would introduce a lot of *usability* issues to configure it up (e.g., how to dynamically load a library at runtime, and how to deal with different JVM versions and different programming languages). When introducing a novel approach, it is important to take into account how complex it is to set it up by practitioners. For this, industry collaborations, where actual engineers use these techniques on their systems (as we do for this article), are paramount.

4.1.3 SUTdriver Implementation. Figure 7 represents an example of a *SUTdriver* for manipulating the SUT and specifying the info of a RPC-based API. For instance, a `startSut` method at lines 9–24 represents how to start a RPC-based SUT, which is implemented with the Thrift framework and SpringBoot. The method also instantiates needed clients to access the SUT after it starts (see lines 18–20). To provide info specific to the RPC problem, lines 29–31 specify the *RPCInterface* (i.e., `NcsService.Iface`; see Figure 2(b)) and corresponding client instance. Note that the info is specified with a map since an API might have multiple *RPCInterface*s as we observed in our industrial case studies.

[4] https://www.oracle.com/technical-resources/articles/java/javareflection.html. Accessed August 26, 2022.

```java
 1 public class EmbeddedEvoMasterController extends EmbeddedSutController {
 2 
 3   private ConfigurableApplicationContext ctx;
 4   private NcsService.Client client;
 5   private TTransport transport;
 6   private TProtocol protocol;
 7 
 8   @Override
 9   public String startSut() {
10 
11     ctx = SpringApplication.run(NcsApplication.class, new String[]{
12       "--server.port=0"
13     });
14 
15     String url = "http://localhost:"+getSutPort()+"/ncs";
16 
17     try {
18       transport = new THttpClient(url);
19       protocol = new TBinaryProtocol(transport);
20       client = new NcsService.Client(protocol);
21     } catch (TTransportException e) {}
22 
23     return url;
24   }
25 
26   @Override
27   public ProblemInfo getProblemInfo() {
28 
29     return new RPCProblem(new HashMap<String, Object>() {{
30         put(NcsService.Iface.class.getName(), client);
31     }});
32   }
33 
34   @Override
35   public CustomizedCallResultCode categorizeBasedOnResponse(Object response) {return null;}
36 
37   @Override
38   public List<AuthenticationDto> getInfoForAuthentication() {return null;}
39 
40   @Override
41   public List<CustomizedRequestValueDto> getCustomizedValueInRequests() {return null;}
42 
43 }
```

Fig. 7. Snippet code of a driver for NcsService (see Figure 2).

In addition, each framework or each company might define their own rules to represent results. For instance, we found that, in our industrial case study, in most cases a failed function call would not result in any exception thrown to avoid propagation of exceptions in the distributed system, since the services are connected with each other. Thus, inside the response, our industrial partner has its own customized specification to reflect the results of RPC function calls that are linked to their business logic. Without a thrown exception, a response representing an error might be falsely identified as a success if no further info is provided. To address this concrete issue in industrial APIs, in our approach, we provide an extensible method (i.e., `getCustomizedValueInRequests` at line 35) to enable customized categorization of responses with the three levels as *CustomizedCallResultCode* defined in our *RPC Schema* (Section 4.1). By extending the method, the user could directly link their own rules into our testing context. Note that such setup can be easily reused by multiple SUTs if they use same customized specification (as it was for all web services developed by our industrial partner).

As an enterprise system, authentication is typically required to be handled. However, there are many different ways to implement an authentication system in a RPC API, as it is usually not supported natively (at least not in Thrift). For this article, we are mainly supporting the authentication systems used by our industrial partner. Authentication tokens need to be sent as a field in payloads of the messages (similarly as HTTP authentication headers in RESTful APIs). An authentication token can be either *static* (i.e., pre-fixed) or *dynamic*. The latter requires one to get the token from an endpoint (e.g., a `login` RPC endpoint where valid username/password info must be provided), and then add it to all following RPC calls. In our implementation, we support both approaches, which needs to be configured in the *driver*, i.e., by extending the method `getInfoForAuthentication` at line 38 and `getCustomizedValueInRequests` at line 41 as shown in Figure 7. To serve a more fine-tuned setup for authentication, we enable options to specify (1) if either the authentication is applied for all API functions; or (2) specific only to some functions in that SUT, that could be filtered by names or by special annotations applied on these functions. More details about how to configure the option could be found in two DTOs, i.e., `JsonAuthRPCEndpointDto` and `CustomizedRequestValueDto`, in our implementation.[1]

4.1.4 RPC Schema Parser. Regarding the extraction of RPC interface definitions, currently, we target JVM RPC-based APIs using Java Reflection. As for the examples shown in Figures 2 and 3, a client-stub *RPCInterface* is composed of a set of available RPC functions to be extracted. Each operation in the interface depicts a RPC function to be called in this service. Then, with reflection, for each interface, we identify all such public methods, and then further extract info on their input parameters, return type, and declared exception types.

Regarding datatype, as currently targeting JVM projects, we have supported the most commonly used data types, i.e., `Array`, `ByteBuffer`, `Date`, `Enum`, `List`, `Map`, `Set`, `String`, `Integer`, `int`, `Boolean`, `bool`, `Double`, `double`, `Float`, `float`, `Long`, `long` `Character`, `char` `Byte`, `byte`, `Short`, `short`, `BigInteger`, `BigDecimal`, and any customized DTO object. For the handling of generics, we support their instantiations for any of these common data types. Note that all of the datatype could be mapped to an item defined in *RPCSupportedDataType*.

Regarding the parameter, besides its datatype, we also need to extract info, such as accessibility and constraints if they exist. Extracting accessibility is needed for the parameter typed with DTO, then its fields might be publicly accessible or not, i.e., declared as `public` or not in Java. If the field is not publicly accessible, there is a need to further extract its existing getter and setter that would be used in assertion generation (with getter) and parameter construction (with setter) in our context. Note that the accessibility info for each parameter is maintained inside the *RPC Test Client* that does not expose in DTO, since the user does not need to care about how to construct the data instance for the parameter and assertion generation. More details about how the info is constructed can be found in the class `AccessibleSchema`.[1]

Regarding the constraints, a parameter might be specified with constraints in its implementation. For example, an integer representing the day of the month could be constrained between the values 1 and 31. To make RPC function calls that do not fail due to input validation, we need to handle such constraints when generating input data for the call. Therefore, for each parameter, with the proposed schema, we define possible constraints as properties of *ParamDto* (see Section 4.1.1 and Figure 6). With the extraction, we further identify the properties based on the data types. For instance, all parameters are defined with a property named *isNullable* representing whether a parameter object can be *null* (the value of this property for all primitive types is always false). Parameters with numeric data types are defined with *min* and *max* properties. For parameters representing collections (e.g., maps and lists) and string types, properties for constraining their size/length are defined, i.e., *minSize* and *maxSize*. For strings, we define *pattern* for supporting a constraint specified with regular expressions. If a string has to represent a numeric value, we use *minValue* and *maxValue* for supporting a possible range constraint for it.

To identify constraints defined in the interface definitions (typically with annotations), we enable constraint extraction on `javax.validation.contraints` [8], which is the standard library for defining built-in constraints for Java objects. We support 16 commonly used constraints, i.e., `AssertFalse`, `AssertTrue`, `DecimalMax`, `DecimalMin`, `Digits`, `Max`, `Min`, `Negative`, `NegativeOrZero`, `NotBlank`, `NotEmpty`, `NotNull`, `Pattern`, `Positive`, `PositiveOrZero`, and `Size`. Besides standard `javax` annotations, constraints could be defined in other ways as well. For instance, in Thrift, whether a field is *required* is represented by a `requirementType` property of the `FieldMetaData` class. Thus, in order to deal with constraints in the Thrift framework, we further extract and analyze the `metaDataMap` object in the interface for obtaining such constraints.

In addition, since there is no general standard to restrict such interface implementations (as long as it compiles), the method and the data type might use Java Generics (as we found in our industrial case study). Therefore, we further handle such generic types when processing RPC function extraction, e.g., analyze `getParameterizedType` for each parameter.

With the *RPC Schema Parser*, it is capable of formulating each *RPCInterface* as *RPCInterfaceSchemaDto* shown in Figure 6.

4.2 Genotype Representation

Given an extracted *RPCSchemaDto* schema, we need to define how to represent the genotype of the evolving test cases. In our context, a test could be reformulated as an individual which is composed of a sequence of RPC function calls. Each function call is formulated as *RPCCallAction*, which comprises the method name, input parameters (if any), optional authentication info, and a response (if declared).

For each input parameter, we define a *gene* with a specific type to represent the parameter. A gene is an instance for the specific type, with constraints on how it can be *mutated* (i.e., modified by the search operators) during the search. For example, a numerical parameter could be internally represented as an integer, initialized with a random value, where the search operators could add or move a delta from such value during its evolution. Textual parameters could be represented with a string, where search operators can either modify its characters, and add or delete some of them (and so changing the length of the string). For more complex types, genes can be hierarchically combined in a tree structure. For example, an object is represented with a gene that has one child gene for each field of the object (and so on recursively, if any of these child fields is an object itself).

There are many types of possible parameters to handle. To achieve a full support to be able to handle RPC-based APIs, we re-use (and extended where needed) the gene system already present in our EvoMaster fuzzer. Regarding the input parameters, we could re-use existing *Gene* objects already defined in EvoMaster for supporting REST API testing, such as, for example:

— Straightforward mapping: *ArrayGene* for `Array`, `Set` and *List*; *BooleanGene* for `Boolean` and `bool`; *DoubleGene* for `Double` and `double`; *LongGene* for `Long` and `long`; *FloatGene* for `Float` and `float`; *EnumGene* for `Enum`; *DateGene*, *DateTimeGene* and *TimeGene* for `DateTime`;
— *MapGene* for `Map`. Note that the original version of *MapGene* only supports keys with string type. However, other types such as enum and integer are quite common in RPC-based APIs. Therefore, we further extended *MapGene* for enabling key to be specified with *IntegerGene*, *StringGene*, *LongGene*, and *EnumGene*.
— *IntegerGene* for `Integer`, `int`, `Short`, `short`, `Byte`, and `byte` (various types here are distinguished by min value and max value, e.g., max value is configured as 127 for `Byte` by default if it is not constrained);

— *StringGene* for `Character`, `char`, `String`, and `ByteBuffer` (various types are distinguished by min and max length, e.g., max length is configured as 1 for `char` by default if it is not constrained);
— *RegexGene* for a pattern specified in `String` parameter;
— *ObjectGene* for representing customized class object;
— *CycleObjectGene* for a field in the customized class object that leads to a cycle;
— *Optional* is for handling any parameter whose *isNullable* property is true.

In addition, we also purpose new genes, such as *BigDecimalGene* and *BigIntegerGene* for `BigDecimal` and `BigInteger`, respectively. In the original implementation of *Gene*s in EvoMaster, constraints for all types are not fully supported. Therefore, to fully support testing the RPC APIs in our case study, we extended genes by enabling all constraints we defined in *RPC Schema*, such as handling precision and scale for numeric genes, and min and max size constraints for *ArrayGene*, *MapGene*, and *StringGene*. This means that, when these genes are either sampled at random, or modified throughout the search via mutation operators, all (linear) constraints are kept satisfied (e.g., a mutation operator would not try to increase a numeric value if it is already at its maximum as defined in its gene constraints).

To test a RPC-based API, the input parameters could be either automatically generated or manually configured by the user (e.g., unlike header in HTTP request, in RPC function call, authentication info could be specified as parts of input DTOs). The former one would be handled by search techniques in our approach. The way to enable authentication as part of the input parameters can be identified as the latter option, i.e., manually defined inputs. To allow further combinational handling with both automatic and manual solutions, we decided to extend the test reformulation with a new gene, i.e., *SeededGene*, for handling manual inputs in a more generic way. A *SeededGene*, representing a gene that has a set of candidates, is constructed with the following: (1) *gene* is the original genotype of the parameter that could be mutated with the search; (2) *seeded* is an *EnumGene* with the same type as *gene* presenting enumerated candidates; and (3) *employSeeded* is a Boolean to indicate whether the original *gene* or the *seeded* gene is used for the phenotype of *SeededGene*. Besides handling authentication info, this kind of gene also allows further seeding with existing data (if any) that would be useful, in particular, in solving industrial problems.

To be able to efficiently fuzz real-world APIs, currently EvoMaster has more than 80 different types of genes in its search-based fuzzer engine [28]. A full description of each of them is not viable here. For low-level technical details, the interested reader can check out our implementation [28], in particular, the code under the `org.evomaster.core.search` package.

4.3 Fitness Function

To evaluate the fitness of a test case, we need to be able to make calls toward the API, with the right inputs, in the right format. The fitness itself will be based on two different kinds of metrics: white-box heuristics based on the execution in the source code (which requires the API to be instrumented with probes), and black-box heuristics based on the responses returned from each RPC call.

To make calls on the API, we use the *client library* provided by the API itself (recall Section 4.1.2). Most RPC frameworks (e.g., Thrift and gRPC) provide ways to automatically generate client libraries (recall Section 2.1). However, there would be several technical issues in dynamically loading such library inside the core process of EvoMaster. Our solution is to let the user specify (and link) such client libraries in the *driver* classes that need to be written to run the white-box mode of EvoMaster (recall Section 2.3). This means that, when a test case needs to be evaluated, the *core* process sends a representation (in JSON format) of such a test case to the *driver*, and then the driver executes the actual RPC call and collects its response. Plus, the driver also collects any white-box heuristics from the instrumented API. Then, all this information is sent back to *core*, where the fitness value for the test is computed.

This architecture to support fuzzing of RPC APIs introduces some latency, as the EvoMaster core does not communicate directly with the API. However, it has major benefits, as it is much easier to set up and implement (e.g., there is no need to parse any `.thrift` or `.proto` file), as well as enabling supporting all different kinds of RPC frameworks with little effort.

4.3.1 RPC Execution Result Analysis. By using client to invoke RPC function call, a result received at the client side could be a return value as defined or an exception thrown from the API. To enable the result analysis, we proposed five main concepts denoted as classes with gray background in Figure 6, i.e., *ActionResponseDto*, *RPCExceptionInfoDto*, *RPCExceptionCategory*, *RPCExceptionType*, and *CustomizedCallResultCode*. *ActionResponseDto* is a DTO that captures all info returned from a RPC function call, i.e., throw an exception (see *0..1 exceptionInfoDto*) or return a value as specified (see *0..1 response*).

Regarding exception, handling the exception info for RPC functions is crucial for testing purposes, e.g., to be able to use automated oracles to identify faults in the SUT. To analyze an exception, in our proposed schema, we define *RPCExceptionInfoDto*, which captures *exceptionName*, *exceptionMessage*, *type*, and *exceptionDto*, which is an optional DTO representing possible additional info for customized exceptions (e.g., the exceptions declared with the keyword `throws` in Java). In addition, when invoking RPC function calls with clients that could be proxy clients, an exception caught at the client side might be wrapped, such as `UndeclaredThrowableException`[5] in Java. To get the exact exception info, we further extract and analyze the actual exception (e.g., with `cause` of `UndeclaredThrowableException`) as *RPCExceptionInfoDto*. We also perform further exception analysis on `UndeclaredThrowableException`, as was needed for our industrial case study, and the property *isCasueOfUndeclaredThrowableException* represents whether such a wrapped exception is thrown from the SUT. Note that the actual exception analysis could be extended in the future when needed.

Besides *exceptionName* and *exceptionMessage*, to better identify exceptions in the context of RPC-based APIs, based on domain knowledge, we classify exceptions into four categories as *RPCExceptionCategory*: *APPLICATION* (e.g., internal server errors), *TRANSPORT* (e.g., connection timeouts), *USER* (e.g., sending invalid data), and *UNCLASSIFIED*. Different RPC frameworks can define their own exceptions for handling various situations for RPC (e.g., type of `TApplicationException` [12] defined in `TException` for Thrift, status [11] defined in `StatusException` and `StatusRuntimeException` for gRPC). To cover such knowledge captured in various RPC frameworks, we define *RPCExceptionType*, and each of the types should belong to a category in *RPCExceptionCategory*. The *RPCExceptionType* now provides full support for analyzing exceptions in the Thrift framework, which covers the complete 24 exception types from `TApplicationException` (refer to *APPLICATION* category), `TProtocolException` (refer to *USER* category), and `TTransportException` (refer to *TRANSPORT* category). In addition, we define two generic exception types, i.e., *CUSTOMIZED_EXCEPTION* representing a declared exception (e.g., `throws` keyword in Java), and *UNEXPECTED_EXCEPTION* representing an exception that is not declared in the function and does not belong to any other identified types (e.g., `RuntimeException` in Java). The generic exception types link to the *UNCLASSIFIED* category that covers the cases whereby the exception type is unspecified or its identification is not supported yet for linking it to a specific RPC exception (like Thrift).

[5]https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/InvocationHandler.html. Accessed August 26, 2022.

With *ActionResponseDto*, considering how a RPC call is handled by the SUT and if there is any exception, we classified it into seven kinds of execution results that would contribute to define search heuristics for optimizing generated tests:

— (ER1) *internal error*: an exception that represents an internal error is thrown, e.g., `TApplicatinException` with `INTERNAL_ERROR` type in Thrift.
— (ER2) *user error*: if an exception was thrown that can be traced to a failed input validation, based on Thrift's protocol errors.
— (ER3) *transport error*: an exception that represents transport errors is thrown.
— (ER4) *other exception*: other exception (e.g., other types of `TApplication` except internal error) is thrown.
— (ER5) *declared exception*: an exception declared in the function is thrown.
— (ER6) *unexpected exception*: any other exception that is not declared in the function is thrown.
— (ER7) *handled*: a value is returned as declared without any exception thrown. If users specify their result categorization, this label is further refined as one of *success*, *service error*, and *other error*.

With HTTP, a result for a request could be identified based on *status code* in its response, e.g., 2xx indicates a success, 4xx indicates a client error, and 5xx indicates a server error. Such a standard is useful in developing automated testing approaches, e.g., reward requests with 500 status code (for finding potential faults in the SUT) and 2xx status code (for covering a successful request). However, in the context of RPC, there does not exist such standard, and a result (e.g., success or failure) of the call cannot be directly determined based on the return value if there is no exception thrown. Therefore, we propose *CustomizedCallResultCode*, which defines three categories (i.e., *SUCCESS*, *SERVICE_ERROR*, and *OTHER_ERROR*) to better identify a return value of a RPC function call. Identifying the return value could vary from SUTs to SUTs, and from companies to companies. So, we expose an interface to allow a customization of the identification (see Section 4.1.2).

Thus, with our RPC result analysis specification as shown in Figure 6, each result by a RPC function call would be constructed as an instance of *ActionResponseDto*. If there is an exception thrown, *RPCExceptionInfoDto* could be instantiated to describe info of exception in detail, such as exception class, message, type, and category. If a value is returned as defined, the value could be represented as a JSON object (if could) and an instance of *ParamDto*, and the result could be further identified with *CustomizedCallResultCode*.

4.3.2 RPC Test Client. This component mainly enables invocation of RPC function call with *RPCActionDto*, analysis of the response or exception after the invocation, then outputting *ActionResponseDto*. With the *RPCActionDto*, we could know what interface the action belongs to and what parameters are needed to construct, then the invocation is made based on the provided RPC client instance. Result analysis is performed based on concepts we discussed in Section 4.3.1. For instance, now we support extract name and message info from all exceptions that inherit from `java.lang.Exception`. Its explicit type could be identified if it belongs to the Thrift framework, i.e., `org.apache.thrift.TException` can be found in the client library, the class of the thrown exception inherits from `TException`, then extract its super classes to recognize the exception category (e.g., *APPLICATION*) under *RPCExceptionCategory* and its `type` property to identify a type (e.g., *APP_INTERNAL_ERROR*) under *RPCExceptionType*. If the exception is not from the Thrift framework, its explicit class would be extracted, then set it with *CUSTOMIZED_EXCEPTION* and *UNEXPECTED_EXCEPTION* based on whether the exception is a part of the throw clause declared in the RPC function. Note that the result analysis needs to be extended if one wants to support other RPC frameworks, such as gRPC. However, exceptions in the context of the RPC domain have

Table 1. RPC Testing Targets with Heuristics

#	Execution Results	*Handled*	*Error*	*isFault*
1	ER1: *internal error*	0.5	1	Yes
2	ER2: *user error*	0.1	0.1	-
3	ER4: *other exception*	0.5	1	-
4	ER5-6: *unexpected/declared*	0.5	1	Yes
5	ER7: *handled*	1	0.5	-
		Success	*Fail*	*isFault*
6	*success*	1	0.5	-
7	*server error*	0.5	1	Yes
8	*other error*	0.1	0.1	-
		NotNull	*Null*	
9	a null response is returned	0.5	1	-
10	a non-null response is returned	1	0.5	-
		NotEmpty	*Empty*	
11	an empty collection is returned	0.5	1	-
12	a non-empty collection is returned	1	0.5	-

been formulated in our schema. The additional work would be only technical details that we need to cope with, e.g., add additional types if they are not covered yet, then extract the specific info to identify the type.

More technical details on this implementation (e.g., how the parameters could be constructed for each data type, how to automatically recognize input parameters with customized info, and how to extract data and type from a Java object) can be found in our open source repository.[1]

4.3.3 Test Execution. With our RPC handling support in the *driver*, we enable tests to be executed during the search. Then, with the JVM instrumentation provided by EvoMaster, various SBST heuristics (e.g., code coverage, branch distance, and SQL queries heuristics) can be returned after the test is executed (see *JVM Instrumentation* → *Fitness Function* in Figure 5), additionally to the RPC function call execution results (i.e., *ActionResponseDto*). Regarding the authentication handling, dynamic tokens acquired via a login endpoint can be regarded as an additional action that needs to be invoked before the other RPC functions can be called. This has been enabled automatically in our implementation.

For white-box heuristics, we rely on the current state-of-the-art in white-box fuzzing of Web APIs given by EvoMaster [50, 73]. This includes adaptation of traditional SBST heuristics like the *branch distance* [20], as well as advanced *testability transformations* [26] and *SQL handling* [24].

In the context of testing RPC-based APIs, besides using SBST heuristics at code coverage level, we propose additional novel testing targets (with their heuristics) on the responses of the RPC calls for guiding the test case generation, as shown in Table 1. Note that, with MIO, each testing target has a fitness value between 0.0 and 1.0, where a higher value is better. A value with 1.0 means that the target is *covered*, and any value more than 0.0 but less than 1.0 indicates that a testing target is *reached* but not *covered*.

For each RPC function, we create two testing targets *Handled* and *Error*, representing that the call is handled or in error, respectively, by the SUT. Based on the execution results we reformulated in Section 4.2, we set a fitness value of *Handled* and *Error* testing targets as #1–#5 in Table 1, after the call is executed. For instance, if the execution result is identified as *handled*, fitness values are set as 1.0 for *Handled* and 0.5 for *Error* (0.5 here represents the target is *reached* but not *covered*, which is heuristically better than not calling the method at all). If any unexpected or declared exception is thrown, the fitness values are set as 0.5 for *Handled* and 1.0 for *Error*. Since the exception type for the unexpected/declared exceptions is unclear, the execution would be further rewarded with a testing target for potential fault finding. If the exception type could be further identified, the fitness values of *Handled* and *Error* would be handled as #1–#3. Note that, for these three types of categorized exceptions, only *internal error* is rewarded for potential fault finding. Considering that the *protocol error* typically refers to user errors, compared with other exceptions, it would be less important; then it is set with lower fitness values (i.e., 0.1) for both *Handled* and *Error*. As *transport error* (ER3) is usually due to issues in the testing environment (e.g., timeouts), we do not reward such exception with any fitness values.

In addition, if the *handled* results could be further categorized by the user in terms of their business logic, we propose two additional testing targets *Success* and *Fail*, representing whether the request succeeds or fails to be performed on the SUT. Heuristics for handling the two targets regarding execution results are defined in #6–#8. The strategy to decide the fitness values is similar with *Handled* and *Error* (e.g., *server error* is rewarded with potential finding and *other error* is recognized as less important) that aims at covering both *Success* and *Fail* of the RPC function actions in terms of business logic. Moreover, to maximize response coverage, we also propose another four testing targets by considering whether any null or non-null value is ever returned (i.e., #9 and #10), and whether any empty or non-empty value is ever returned for collection datatypes (i.e., #11 and #12). Note that, although some of these fitness values do not provide much gradient for the search (e.g., only two values such as 0.5 and 1), they are still useful. Test cases for reached but not covered targets (e.g., 0.5) are kept in the archive of MIO, and will still be sampled and mutated throughout the search.

4.4 Search Operators

Our test reformulation enables its use in various search algorithms for supporting RPC-based API fuzzing. In this work, we use MIO because it is the default in EvoMaster, as it achieved the overall best results in an empirical study conducted by comparing it with various other algorithms [15] on the fuzzing of RESTful APIs (recall Section 2.3). However, other search algorithms might be better on the problem of fuzzing RPC APIs. But, without further empirical analyses, this is not something that can assessed for sure. Due to the high cost of running this type of comparison experiments, a comparison of different search algorithms for fuzzing RPC-based APIs is not in the scope of this article.

MIO is an evolutionary algorithm inspired by (1+1) EA that uses two search operators, for sampling and mutation, respectively. We employ the same strategies as EvoMaster for RESTful API testing. The sampling is implemented to produce a valid test by selecting a sequence of one or more available actions at random. Values of *Gene*s in these tests are initialized at random, within the constraints, if any (e.g., a *ArrayGene* will have n randomly generated elements based on its min and max length). Authentication info, if any, is enabled with a given probability, i.e., 95%, which is the default one used in EvoMaster. In addition, at the beginning of the sampling, we also prepare a set of ad hoc tests that cover all available RPC function calls and all authentication combinations, i.e., each test has an action configured with and without authentication. In other words, the structure of the first k tests are not sampled at random, where $k = a \times n$, with n being the number of functions in the RPC API and a being the different authentication settings.

Regarding the mutation operator, actions in a test can be added or removed for manipulating the structure of the test, given a certain probability. To mutate the values of *Gene*s inside the tests, we employ the default value mutation in EvoMaster, which has been integrated with *taint* *analysis* [26] and *adaptive hypermutation* [71]. How each gene is mutated depends on its type and constraints (if any), as previously discussed in Section 4.2.

Given a typical evolutionary algorithm with an individual representation having n bits, then on average each gene would be mutated with probability $1/n$. However, the genes defined in EvoMaster can have massive differences in terms of their genetic information. For example, a Boolean gene would represent only two possible values (for `true` and `false`), whereas an object gene for a complex DTO could have hundreds of internal fields. The search engine of EvoMaster can deal with genes of different *weight*, and mutate the ones with more weight more often. Furthermore, *adaptive hypermutation* [71] enables having a higher mutation rate, and automatically detects which genes have less (or no) impact on fitness, and automatically mutates them less often.

If the SUT interacts with a SQL database, genes to represent `INSERTION` operations will be automatically added to the tests, in the same way as done in EvoMaster for RESTful APIs [24].

4.5 Test Suite Output

In the same context of API testing, we could re-use parts of EvoMaster test writer to generate the SUT test scaffolding. For example, we use the same `initClass` for setting up the necessary testing environment (e.g., start SUT), `tearDown` for performing a cleanup after all tests are executed (e.g., shutdown SUT), and `initTest` for resetting the state of the SUT for making test executions independent from each other. To enable a more efficient test execution and fit industrial-scale API testing, we extended `initTest` with our smart database clean procedure, by considering the union of all accessed tables, and their linked tables, for all tests that are generated.

Regarding handling of action execution and assertion generation, with EvoMaster, tests are generated with *RestAssured* to make HTTP calls toward the tested REST API. This is not applicable in the context of RPC testing. Then, to support RPC-based API testing, we develop a *Test Writer* that could handle instantiation of input parameters, RPC function call invocation (based on the RPC client library), and assertions on response objects with JUnit. An example of generated tests can be found at this link.[6]

In our industrial case study, we found that some responses contain info such as timestamps and random tokens, and they could change over time. In order to avoid test failing due to such flakiness, we defined some general keywords (e.g., date, token, time) to highlight those cases. If any keyword appears in either datatype, field name, or value with string type, the assertion would be commented out to avoid the test becoming flaky. We comment them out instead of removing them completely since it would still be interesting, for the users, to show what the response was originally.

In addition, there might exist quite large responses in some API endpoints, especially when dealing with collections of data. For example, in one SUT in our case study, a response contained 470 elements, and each element further contains data with list type, and 7,579 assertions were generated for this response. As such a large number of assertions would reduce the readability of the tests, we then developed a strategy to randomly select only n (e.g., $n = 2$) elements from the returned collections to generate assertions on in the tests. More details on the writer can be found in our open source repository.[1]

Generating this kind of tests has two main advantages. First, as the generated tests are *self-contained* (because they are able to start and stop the API directly without manual intervention), they can be used for *regression testing*. Second, they help *debugging* any found fault, as each generated test can be run independently, because they take care of initializing and reset the state of

[6]https://github.com/anonymous-authorxyz/fuzzing-rpc/blob/main/example/src/em/EM_RPC_1_Test_others.java. Accessed August 26, 2022.

Table 2. Descriptive Statistics of Case Studies

	#Interfaces	*#Functions*	*#Services* (*#U*, *#D*)	*#Classes*	*#LoC$_f$* (*#LoC$_j$*)	*#Tables*
thrift-ncs	1	6	0	7	506 (254)	0
thrift-scs	1	11	0	12	695 (260)	0
CS1	3	24	7 (6, 1)	101	12,559 (4,019)	6
CS2	5	20	11 (7, 4)	144	18,987 (1,821)	17
CS3	8	51	18 (14, 4)	339	45,987 (18,800)	156
CS4	8	55	36 (27, 9)	868	116,340 (20,760)	50
Total	26	167	72 (54, 18)	1,471	195,072 (45,914)	229

#Interfaces represents the number of *RPCInterface*s, *#Functions* represents the number of available RPC functions, *#Services* represents the number of direct interacted external services (divided between *#U* of upstream and *#D* downstream services), *#Classes* is the number of Java class files, *#LoC$_f$* is the number of lines of code in Files (*#LoC$_j$* is the number of lines of code reported by JaCoCo), and *#Tables* is the number of SQL tables.

the API (e.g., SQL databases). This feature was critical when analyzing the faults found during our empirical study.

5 EMPIRICAL STUDY

5.1 Research Questions

In this article, we conduct an empirical study to answer the following research questions:

RQ1: How does our white-box fuzzing perform compared with a baseline gray-box technique?
RQ2: How does our novel approach perform in terms of code coverage?
RQ3: Does our novel approach find real faults in industrial settings?

5.2 Experiment Setup

To evaluate our approach (denoted as RPC-EVO), we carried out an empirical study with two artificial and four industrial RPC-based APIs selected by our industrial partner. The industrial case studies are from a large-scale e-commerce platform (comprising hundreds of web services referred to as microservices) developed by Meituan. Descriptive statistics of the case studies are summarized in Table 2. *thrift-ncs* and *thrift-scs* are re-implemented by us with Thrift, based on existing artificial RESTful APIs that have been used to assess the effectiveness of solving *numeric* and *string* problems [20, 21, 26, 71]. *CS1–CS4* are from our industrial partner. Each one of them is a part of a large microservice architecture, where each API interacts with other services and a database. *#Services* (in Table 2) shows the amount of external services that a SUT directly interacts with (see an example in Figure 1), where *#U* is the amount of its upstream services that the SUT depends on, and *#D* is the amount of its downstream services that call the SUT. The lines of code (*#LoC*) numbers include everything, such as comments, empty lines, and import statements. The actual lines with code (which results in `LINENUMBER` instructions in the compiled `.class` files) are calculated with the coverage tool JaCoCo (i.e., *#LoC$_j$*).

Ideally, experiments should be carried out on real industrial systems. However, we also employed two artificial case studies (which we open sourced) to make at least parts of our experiments replicable,[1] as of course we cannot share the code of the industrial systems. In addition, to further demonstrate its adoption and performance in industrial settings, we also report preliminary results on 50 further industrial APIs. Note that the testing of those 50 APIs was autonomously performed by our industrial partner (e.g., prepare EvoMaster drivers, without any researcher involved), as part of an internal evaluation to see whether/how to integrate EvoMaster in their CI pipelines.

For the choice of baselines for comparisons, regarding other tools in the literature, to the best of our knowledge, there does not exist any other automated testing solution for RPC-based APIs that could be applied as a baseline in this study, as discussed in more detail in Section 3. Therefore, we adapted our approach to be used by the Random Search Algorithm in EvoMaster, which can be regarded as a gray-box technique (testing targets such as code coverage are still employed to evaluate tests to produce a final test suite as output at the end of the search). This random search serves as a baseline to evaluate our approach in the context of white-box testing. To be comparable, the same search budget (i.e., 100,000 RPC function calls) are applied for all settings with these techniques. In addition, to further evaluate the performance of our generated tests, we also compare them with existing tests in the industrial case studies.

Regarding the applied evaluation metrics, in these comparisons we used three main metrics: *line coverage*, *target coverage*, and *fault detection*. Line coverage is measured with the code instrumentation of EvoMaster, and it is based on the `LINENUMBER` bytecode instructions in the business logic of the tested APIs (i.e., no third-party library). Fault detection is based on the oracles defined in Section 4.2, i.e., it counts the number of unique thrown exceptions related to server errors. They are differentiated not only based on the entry point of the API (i.e., the actual method that is called remotely), but also on the last executed statement in the business logic of the API (as each endpoint can fail for different reasons, while executing different lines of the code). The target coverage is an aggregated metric in EvoMaster that considers *all* its code coverage metrics (e.g., besides covered lines, it also considers, for example, the number of loaded classes, covered branches in jump instructions, methods called without throwing exceptions, and Boolean methods returning `true` and `false` at least once), as well as the number of found faults, and black-box metrics on the API responses (recall Table 1).

Considering the stochastic nature of the search algorithms, all experiments on each of the six main APIs were repeated 30 times, by following common guidelines in the literature [22]. *thrift-ncs* and *thrift-scs* were executed on an HP Z6 G4 Workstation with Intel(R) Xeon(R) Gold 6240R CPU @2.40 GHz, 2.39 GHz processor, 192 G RAM, and 64-bit Windows 10. The four industrial APIs were executed on the actual hardware pipelines of our industrial partner. With these pipelines, all external services of the SUTs are up and running. In an industrial testing environment, databases can be pre-loaded with lots of data (e.g., replicas of the production database), for covering their specific business logic. The amount of such data can be quite large, e.g., 256,024 data entries in *CS3*. In an automated testing process, it is difficult to maintain such large data cost-effectively (e.g., clean and re-insert them back after each test execution) for ensuring that each test is executed with the same state of the SUT. Therefore, we decided to use empty databases to conduct our experiments with the industrial APIs.

To get a better insight into the applicability of our novel techniques in real industrial contexts, we also report on the use of EvoMaster on the current testing pipelines at Meituan. This included 50 APIs, in which no researcher was involved in the running of these fuzzing sessions. No comparison with the *gray-box* technique was made here. The preliminary results on these 50 APIs are based only on one run, each one where EvoMaster was run for 10 hours.

5.3 RQ1: Comparison with Gray-Box Technique

To answer RQ1, we applied our approach and the random search strategy on all of the six case studies with the same search budget, i.e., 100,000 RPC calls. The computation cost of two settings with at least 30 repetitions is 30.64 hours for the two artificial APIs, and 129.12 days for the four industrial APIs (maximum 18.27 hours and minimum 9.41 hours per run on the industrial APIs). Note that the 30 repetitions are only applied in this empirical study for evaluating the approach. When used by practitioners on their systems, the approach can be run just once.

Table 3 reports the results of target and line coverage, with comparison results on the two metrics using statistical analysis, as recommended in [22]. In particular, for pair comparisons we use

Table 3. Pair Comparisons between Our Approach (RPC-EVO) and Random with #*Targets* and %*Lines* on All SUTs

SUT	Metrics	RPC-EVO	Random	$\hat{A}_{12}$	p value	Relative
thrift-ncs	#*Targets*	**542.2**	376.3	**1.00**	**≤0.001**	**+44.07%**
	%*Lines*$_e$	**88.2%**	60.0%	**1.00**	**≤0.001**	**+47.00%**
thrift-scs	#Targets	**658.4**	549.4	**1.00**	**≤0.001**	**+19.85%**
	%*Lines*$_e$	**71.9%**	59.6%	**1.00**	**≤0.001**	**+20.59%**
CS1	#Targets	**1,953.3**	1,773.1	**0.97**	**≤0.001**	**+10.16%**
	%*Lines*$_e$	**25.6%**	23.1%	**0.99**	**≤0.001**	**+11.15%**
CS2	#*Targets*	**3,434.1**	3,099.1	**0.96**	**≤0.001**	**+10.81%**
	%*Lines*$_e$	**27.2%**	24.7%	**0.94**	**≤0.001**	**+10.18%**
CS3	#*Targets*	**4,453.4**	4,067.2	**0.96**	**≤0.001**	**+9.50%**
	%*Lines*$_e$	**15.4%**	14.0%	**0.99**	**≤0.001**	**+9.78%**
CS4	#*Targets*	**6,229.7**	6,046.8	**0.80**	**≤0.001**	**+3.02%**
	%*Lines*$_e$	**6.5%**	6.3%	**0.91**	**≤0.001**	**+4.46%**

the Mann-Whitney-Wilcoxon U-test (see p values) and Vargha-Delaney standardized effect size ($\hat{A}_{12}$). The U-test is used with the standard $\alpha = 0.05$ level (i.e., we claim statistical difference if a p value is lower than 0.05). In this context, the $\hat{A}_{12}$ effect size provides a probability estimate that an algorithm produces better results than the other compared one. If two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A value like $\hat{A}_{12} = 1$ means that, in every single run, the first algorithm always gave better results. Note that the $\hat{A}_{12}$ effect size only computes *how often* an algorithm gives better results, but not by *how much*.

Given these results, our approach demonstrates significantly better results than the baseline technique, with a low p value (i.e., ≤ 0.001) and a high effect size (i.e., $\hat{A}_{12} > 0.80$) on all of the six case studies with the two metrics.

In addition, Figure 8 plots the average covered targets over time (i.e., at every 5% of the used budget) for two techniques on each case study. Based on these line plots, our approach clearly outperforms Random by a large margin throughout the whole process of the search, and the results are consistent on all of the case studies. This further demonstrates the effectiveness of our white-box techniques in both artificial and industrial settings.

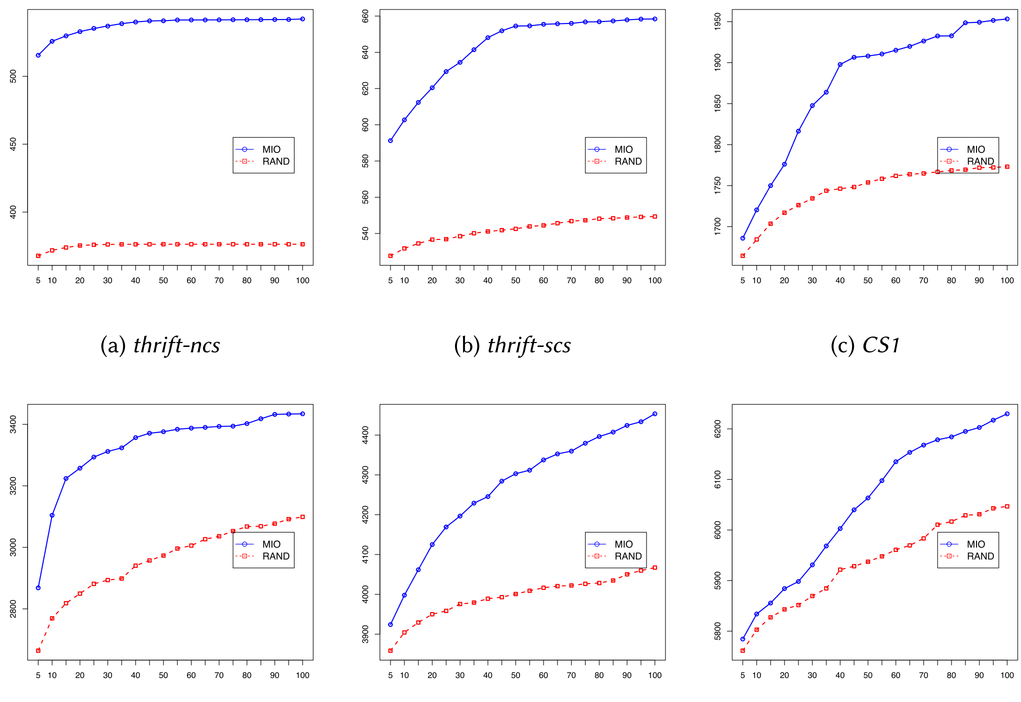
RQ1: Based on the target and line coverage results, our approach significantly outperforms random search on all of the six case studies. The relative improvements are up to 47% on the artificial case studies and 11.15% on the industrial case studies.

5.4 RQ2: Results of Code Coverage

5.4.1 Artificial APIs. Based on coverage (i.e., %*Lines*$_e$) reported in Table 1, on the two artificial case studies representing numeric and string testing problems, our approach achieves high line coverage (i.e., 88.2% on *thrift-scs* and 71.9% on *thrift-scs*) when using $100k$ calls as budget. This high code coverage could demonstrate that RPC-EVO effectively enables the white-box fuzzing for RPC-based APIs, i.e., based on the white-box heuristics, effectively optimize the inputs of extracted/reformulated RPC function calls.

RQ2.1: Our approach achieves high line coverage on the two artificial case studies, demonstrating its effectiveness in enabling white-box fuzzing of RPC APIs.

5.4.2 Industrial APIs. Regarding the four industrial case studies, as a fully automated solution, our approach achieved useful (for our industrial partner) coverage on *CS1* and *CS2* (more than 25%),



(a) *thrift-ncs* (b) *thrift-scs* (c) *CS1*

(d) *CS2* (e) *CS3* (f) *CS4*

Fig. 8. At every 5% of the used budget (x-axis), average covered targets (y-axis) achieved by RPC-EVO and Random.

but limited coverage on *CS3* and *CS4* (especially *CS4*). The results are also related to the complexity of these SUTs (as shown in Table 2), given the same limited search budget. For example, based on *#LoC* values, *CS3* and *CS4* are much larger (and likely more difficult to fully cover) than *CS1* and *CS2*, where *CS4* has more than 2.5 times *#LoC* than *CS3*. In addition, based on the line plots in Figure 8, the slope of the lines in *CS1–CS4* is greater than *thrift-ncs* and *thrift-scs*, especially for *CS3* and *CS4*. This indicates that more targets would likely be covered if more budget is used, i.e., if the fuzzers were run for longer, like 24 or 48 hours. However, without actual experiments, it is not possible to be completely sure.

Comparison with existing tests. To study the performance on code coverage in industrial settings, we compare our generated tests with existing tests. The analysis was conducted with three groups of tests:

— *W*: a test suite generated by RPC-EVO that achieves the worst result out of the 30 repetitions;
— *B*: a test suite generated by RPC-EVO that achieves the best result; and
— *E*: a set of existing tests.

The code coverage was collected by executing the generated tests and existing tests on the SUTs with Intellij [7].

In Table 4, we report the numbers of tests in these groups for each of *CS1–CS4*. Regarding the existing tests in the industrial setting of our partner, those are the actual tests currently used at Meituan to find regression faults in these APIs. These tests were prepared by engineers and testers at Meituan, completely independently from our experiments. There exist two types of these

Table 4. Numbers of Generated Test Cases in the Worst Run (RPC-EVO$_w$) and the Best Run (RPC-EVO$_b$) of Our Approach Out of the 30 Repetitions, Compared to the Number of Existing Tests in the Industrial APIs

SUT	RPC-EVO$_w$	RPC-EVO$_b$	Existing (Manual, Replay)
CS1	80	75	12 (0, 12)
CS2	89	71	5 (5, 0)
CS3	186	156	27 (1, 26)
CS4	236	232	46 (12, 34)

Table 5. Results of Line Coverage Achieved by the Worst Run and the Best Run of our Approach (Denoted as W and B Respectively) and Existing Tests (Denoted as E)

	RPC-EVO (%)	Existing (%)	Total (%)	Uncovered (%)
SUT	$[W, B]$	E	$[W \cup E, B \cup E]$	$[E \setminus W, E \setminus B]$
CS1	[22.50, 26.81]	14.29	[22.95, 27.85]	[0.45, 1.04]
CS2	[25.46, 26.02]	16.28	[25.46, 26.40]	[0.00, 0.37]
CS3	[13.37, 15.89]	5.29	[13.46, 15.92]	[0.09, 0.03]
CS4	[$\underline{8.31}$, 9.15]	8.31	[10.22, 10.90]	[1.91, 1.75]
Avg.	[17.41, 19.47]	11.04	[18.02, 20.27]	[0.61, 0.80]

"Total"represents the union coverage achieved by RPC-EVO and existing tests, and "Uncovered" represents the coverage achieved by the existing tests but not by our approach.

tests. One type is manually written automated tests (e.g., JUnit), and the other is with a replay of manual testing (using a custom testing tool). The manual testing would be driven from the user side, as for the example shown in Figure 1. For instance, a tester at Meituan would perform a real business scenario as a user, by directly interacting with an app on their mobile phone. Then, with the requests from the user, it would result in various RPC communications among the services. Those communications are recorded (such as what calls are invoked) along with the states of the connected external services. With an industrial testing tool, such records are performed as a replay (such as re-execute the calls) on the SUTs for conducting manual regression testing. In order to collect code coverage for the record for the comparisons in this article, we converted the records as JUnit tests by extracting the calls and their inputs (but a setup of the states of external services with the replay tool cannot be transferred into the JUnit tests).

The results of line coverage reported with Intellij achieved by the three groups for *CS1*–*CS4* are reported in Table 5. Note that the line coverage might be slightly different with the results in Table 3 that are reported with EvoMaster bytecode instrumentation. However, the comparison is always performed with the results obtained from the same coverage runner.

In this table, we also provide a union of code coverage achieved by RPC-EVO and existing tests (see *Total%*), and a code coverage achieved by the existing tests but not RPC-EVO (see *Uncovered%*). Then we observed the following:

— First, by checking *Total%* with existing tests E in industrial APIs, on all SUTs, RPC-EVO (i.e., both the worst and the best test suites) can attribute to additional code coverage compared with the existing ones.
— By comparing RPC-EVO with E, on all of the four industrial case studies, the code coverage by RPC-EVO (i.e., the worst and the best) are clearly greater than the existing ones, except *CS4*, which achieves the equivalent results (i.e., $\underline{8.31}$%).

- Regarding *Uncovered*, the percentage is minor (i.e., the max is 1.91%). This indicates that RPC-EVO is capable of covering most of the code achieved by the existing tests, i.e., above 77.02% (=1 − 1.91/8.31), up to 100% of the code covered by *E*.
- One interesting observation here is that the selected worst run perform better in covering lines achieved by existing tests on *CS1* and *CS2* than the best run. This might further reveal various promising regions of search space in industrial problems.

Based on the observation, we can conclude the following:

RQ2.2: Compared with existing tests, RPC-EVO is capable of contributing additional code coverage and demonstrates clear better results. In addition, RPC-EVO could cover above 77.02% line coverage achieved by the existing tests.

In-depth analysis of the coverage reports. To further study why higher coverage was not achieved, we performed a manual analysis on the code coverage reports generated by the best test suite (i.e., *B*) and the source code of these APIs. For *CS4*, we found that 10 RPC functions out of the 55 functions are not accessible with the given client library. By checking with our industrial partner, they think these problems are due to some issues in their testing environment (e.g., which uses a service discovery mechanism and load balancers) that were found as well in executing existing tests. These problems are currently under investigation. This might be a reason for the least line coverage achieved by our approach on *CS4*.

Based on the coverage reports on the four APIs, we found that our approach achieved limited code coverage on the code that is related to database handling and communications with external services. Regarding the database handling code, most of it is automatically generated with an in-house framework for facilitating various manipulations on the database, e.g., to perform a query with various conditions. As discussed with our industrial partner, usually, not all of the generated manipulation code would be used in implementing their business logic. However, they are still generating and keep such code in case of future use. Therefore, a lot of this code is infeasible to be covered with any system test.

Regarding the code related to the communications with the external services, all these services were up and running in the test environment of our industrial partner. Since these external services are not mocked nor is their code instrumented with our SBST heuristics, i.e., not being part of our testing process, then we mostly fail to get different responses with our automatically generated inputs that maximize the code coverage in the SUT (e.g., all the code used to read and act upon the responses given by these external services). How to deal with external services is a major research challenge that applies to all kinds of web services.

Another main issue that we found is related to input validation. In our approach, we have handled all the constraints specified as parts of interface definitions, but there also exist further restricted checks on the inputs. The inputs could be restricted in the internal business logic of the SUT, e.g., an input parameter x could be validated with an external service regarding whether it exists: then, if it exists, it could further query another service for the related data with x. With our current heuristics, such valid inputs could be rarely generated. Moreover, the needed inputs are often complex. For instance, we noticed that, in a generated test, there are 2,024 lines for instancing a single input DTO. As we checked, the length for all lists in that instance is less than 5. Then, we further checked the implementation of the DTO, which contains 25 fields, and the fields could be other DTOs or lists of DTOs. Such very large DTO would lead to additional difficulty to generate valid inputs, e.g., if any element (e.g., in a collection) violates any constraint, then the whole DTO would be considered as invalid, and fail the input validation.

Table 6. Results of Line Coverage of 50 Industrial APIs Achieved by RPC-EVO with 1 Run Using 10 Hours Budget

#	$\#LoC_j$	*#Targets*	*%Lines*	*#Faults*	#	$\#LoC_j$	*#Targets*	*%Lines*	*#Faults*
#01	19,867	26,439	15.44	59	#02	54,457	34,141	15.02	90
#03	39,308	11,459	14.53	82	#04	31,663	27,485	34.65	129
#05	34,872	23,363	28.44	71	#06	27,859	14,071	19.38	172
#07	38,179	15,960	17.96	168	#08	39,075	25,665	27.41	295
#09	58,232	81,279	17.15	292	#10	28,814	14,556	21.93	146
#11	63,108	81,200	9.37	371	#12	30,556	40,379	16.03	265
#13	30,954	29,830	6.11	252	#14	40,644	16,720	14.1	179
#15	7,314	11,280	15.72	66	#16	34,369	48,320	16.51	315
#17	80,929	72,592	11.95	354	#18	38,914	22,582	15.83	160
#19	16,880	13,726	21.59	56	#20	6,597	11,284	16.65	141
#21	6,882	2,294	4.3	4	#22	2,019	5,478	20.97	16
#23	53,565	35,282	11.49	99	#24	28,604	102,260	10.42	**2,250**
#25	15,047	27,929	9.44	82	#26	58,578	54,496	15.01	164
#27	52,878	13,921	12.64	127	#28	8,548	4,735	19.85	35
#29	89,303	18,350	12.64	233	#30	9,782	1,442	8.17	19
#31	10,054	5,273	22.3	67	#32	23,864	5,445	9.11	83
#33	21,055	10,625	27.48	109	#34	19,191	4,709	12.25	41
#35	6,602	6,161	27.72	40	#36	10,676	6,059	15.29	85
#37	16,534	24,051	9.02	95	#38	23,349	11,695	24.84	57
#39	7,006	6,574	17.8	14	#40	8,096	2,478	8.95	16
#41	12,081	3,840	14.41	35	#42	20,769	7,744	15.03	55
#43	41,450	13,774	12.72	74	#44	2,217	8,533	15.99	32
#45	58,439	20,297	19.3	148	#46	10,230	6,059	10.5	13
#47	56,242	20,230	17.69	235	#48	9,691	33,559	23.97	74
#49	3,301	4,594	**59.16**	63	#50	35,401	19,609	30.4	349
***#LoC*$_j$**	**Sum**: 1,444,045; **Avg**: 28,880.90; **Max**: 89,303; **Min**: 2,019								
%Lines	**Avg**: 17.49; **Max**: 59.16; **Min**: 4.3								
	60~50%: 1; **50~40%**: 0; **30~40%**: 2; **20~30%**: 10; **10~20%**: 29; **4~10%**: 8								
#Faults	**Sum**: 8377; **Avg**: 167.54; **Max**: 2250; **Min**: 4								

# is an index of industrial SUTs; $\#LoC_j$ is the number of lines of code reported by JaCoCo; *#Targets* is the number of targets covered by our approach that is composed of lines, branches, and potential faults; *%Lines* is the line coverage achieved by our approach; *#Faults* is the number of potential faults identified by our approach.

Note that there could be more issues at play here that could explain these results. These could include the complexity of the source code, and/or possible side effects of existing search algorithms such as MIO on this problem domain. Without further in-depth analyses, it is currently not possible to pinpoint the main culprit for this low coverage. Regardless, the identified issues will need to be addressed, paving the road ahead for further research on improving the fuzzing of RPC APIs. Once fixed, re-running these experiments will be needed to identify whether there are still any major issues impacting the achieved code coverage.

RQ2.3: Our approach achieves useful coverage (26.81% and 26.02%) on two out of the four industrial case studies, and limited coverage (15.89% and 9.15%) on the other two larger industrial case studies. Based on a manual analysis on code coverage and the source code, we found that the main issues are related to the communications with external services and to generate inputs for complex DTO with various constraints.

Table 7. Results of the Potential Distinct Faults Automatically Reported by Our Approach with 30 Runs for Each Industrial SUT

	Potential Faults	Real Faults			
SUT	Avg. [Min, Max]	L1	L2	L3	*Total*
CS1	40.2 [40, 41]	17	0	5	22
CS2	21.8 [21, 26]	3	1	15	19
CS3	51.6 [51, 55]	1	0	29	30
CS4	91.8 [74, 111]	3	16	36	55
Total		**24**	**17**	85	126

We report as well the number of real faults manually identified and confirmed with the industrial partner. *L1*: faults that will be fixed; *L2*: faults that are needed to be fixed but less important; *L3*: faults that are tolerable, and likely no need to fix.

Additional analysis with code coverage collected by our industrial partner. In Table 6, we report the preliminary results of target coverage and line coverage in 50 industrial APIs achieved by RPC-EVO with 1 run using 10 hours search budget. Note that all these 50 industrial APIs plus *CS1*–*CS4* are parts of one single microservice architecture, with hundreds of web APIs. The 10-hour budget was decided by our industrial partner by considering their application context and time cost per run in this experiment.

Based on the results, on 50 industrial APIs with 1,444,045 lines of code ($\#LoC_j$) in total and 28,880.90 lines of code on average (ranging from 2,019 to 89,303), RPC-EVO achieves 30%–60% line coverage on 3 SUTs, 20%–30% line coverage on 10 SUTs, 10%–20% line coverage on 29 SUTs, and 4%–10% line coverage on 8 SUTs. These code coverage results are consistent with those reported in Table 3 for the other 4 APIs we analyzed in more detail.

RQ2.4: RPC-EVO has been successfully applied in white-box fuzzing 50 industrial RPC-based APIs in practice by our industrial partner. With a 10-hour search budget, results show that our approach is capable of achieving on average 17.49% (up to 59.16%) line coverage.

5.5 RQ3: Results of Fault Detection

To assess the fault detection capabilities of our novel approach, we performed a detailed analysis on the identified faults with our industrial partner, as researchers and industrial practitioners might have different views on the severity and importance of the found faults. The manual analysis is based on the test suites that achieved the best code coverage (out of the 30 runs) for each of the four industrial APIs we analyzed in detail. We applied such selection due to the time constraints of manually checking all the generated test suites in all the 30 repetitions. With this selection, the amount of tests to be reviewed is 534 as RPC-EVO$_b$ shown in Table 4. With these tests, faults are identified based on (1) any exception thrown in the calls; (2) service error represented by assertions on the responses; (3) failed tests when executing them on the SUT (mainly due to flaky assertions); and (4) whether responses are expected based on the given inputs. The review was first conducted by the first author, then an employee of our industrial partner (a QA Manager who has 8 years of testing experience in industry) performed the same kind of analysis on these tests. At the end, a meeting was held to discuss and confirm the final results reported in Table 7.

As shown in Table 7, in total 126 real unique faults were found with the selected test suites on the four industrial APIs. The faults could be further classified into three levels, i.e., *L1*, *L2*, and *L3*, based on the willingness of our industrial partner to fix these faults. *L1* represents the number of faults that are serious enough that they should be fixed. These faults are related to mistakes in the code implementation, errors in handling databases, errors in transaction processing, and potential risky errors in returning a misleading response. At the time of writing this article, the identified faults have all been confirmed and fixed. *L2* is the number of faults that should be fixed but are less critical. Most of the faults at *L2* are related to the implementation of input validation and external service response handling when exceptions are thrown. In their context, it is better to properly handle exceptions within the SUT, as such thrown exceptions might lead to further problems in the services that depend on the tested application. *L3* is the number of minor faults that are tolerable, and most likely our industrial partner will not fix them. These faults are mainly due to input validation throwing exceptions such as `NullPointerException`, `IndexOutOfBoundsException`, and `java.text.ParseException`. However, if the exceptions are caught and handled within the SUT, they consider that such faults are tolerable.

In Table 7, we also report the number of potential faults automatically reported by our approach. As expected, the number of real faults we manually identified is less than the potential ones. This is mainly due to (1) problems in the test environment (e.g., some external services might not have been up and running when the experiments were carried out); (2) data preparation in databases (e.g., an empty database might lead to some problems that would never happen in production); (3) communications over the network (e.g., connection timeouts); (4) client problems (e.g., some remote functions fail for some configuration issues when we ran the experiments for *CS4*). However, with the generated tests, our employed automated oracles could identify most of the real faults, except the errors related to returning a misleading/unexpected response (as this requires the users to manually check the content of these responses, as no formal specification is available).

Regarding the further experiments with 1 run on a further 50 APIs, RPC-EVO identified in total 8,377 and on average 167.54 potential faults in these 50 industrial APIs as shown in Table 6. For the industrial API #24, the number of detected faults is significantly higher than for the other APIs (i.e., 2,250 faults). By performing a further investigation on this API, we found that the API is for handling authentication, and its functions are invoked by many other services in the microservices. Thus, a request to this API requires one to link with a valid account and be specified with valid data, i.e., the account should have an access to the data, and the data should be accessible and satisfy corresponding business features for the account. Any request volatilizing such constraints would throw exceptions under the current implementation. With a 10-hour search budget, tests generated by our approach led to such unexpected exceptions thrown from 2,091 different locations in the code. This might explain why such a high number of potential faults were detected in this API.

At this point in time, we do not know yet how many of these detected 8,377 faults are critical, and must be fixed as soon as possible. This is currently under evaluation by the engineers and testers at Meituan. Going through and debugging thousands of potential faults is a time-consuming task.

RQ3: With an in-depth analysis of the generated tests with our industrial partner, we confirm that our approach was capable of finding 41 actual real faults that have now been fixed. Another 8,377 potential faults are currently under investigation.

6 LESSONS LEARNED

Automated testing requires a reset of the SUT; however, it is challenging to reset the state of a real industrial API. To enable the generated tests to be used for regression testing, and to properly evaluate the fitness of each test case in isolation, it is needed to execute every test with the same state of SUT (i.e., test case executions should be independent from each other). Thus, it requires one to perform a state reset of the SUT before a test is executed on it, e.g., clear all data in the database or reset databases to a specific state. With open source case studies, it is trivial, e.g., clean data in database. For instance, EvoMaster provides a utility *DbClearner* for facilitating the cleaning of data for various types of SQL databases, e.g., Postgres and MySQL. Such a clean on the database does work fine for small-scale applications. However, in large-scale industrial settings, cleaning all data in the database is quite expensive, even when the database is empty. For instance, in one of the industrial APIs used in this article, it takes 5.3 seconds to clean an empty database, and it takes more time if there exist data. Thus, within 1 hour as the search budget, a fuzzer can execute at most 680 RPC function calls. This would significantly limit the fuzzer in terms of cost-effectiveness. To better enable our approach in industrial settings, by taking advantage of existing *SQL handling* in EvoMaster, we developed an automated *smart clean* on the database, by considering only what tables are actually modified during the search. With the smart clean, after a test is executed, data only in the accessed tables and linked tables (e.g., with foreign key) will be removed. In addition, we also allow SQL commands/scripts to initialize data into the database (e.g., for username/password authentication info). If a table that has initial data is cleaned, a post action will be performed to add the initial data for the table again. With such smart database clean, we could effectively reduce time spent by more than 90%, e.g., from 5.3 seconds to 285 milliseconds. This is because there can be tens/hundreds of tables in an industrial API, but only few of them are actually accessed during the executing of a single test. However, how to reset the state of the databases with a large amount of existing data still needs to be addressed. Besides the database, the states of direct connected external services also need to be reset. Currently, fuzzing by our approach is performed on the industrial test environment where all services are up and running. With such an environment, the states of external services might be varied over time (e.g., failed tests as discussed in Section 5.5). Mocking technique could be a potential solution to address this, e.g., set up specific states of the external services before test execution. However, mocking RPC-based services in microservices is also challenging, e.g., due to network communications and environment setup in industrial settings. It could be considered as important future work.

Real industrial APIs have more complex inputs and apply stricter constraints on input validations with considerations of various aspects. By checking code coverage and fault detection, we found that most codes and faults are related to the parts of implementation for input validation. One reason could be due to the complexity of the input with cycle objects and collections in DTOs. For instance, we found that a DTO is initialized with more than $2k$ lines, and generating a valid input for such a huge DTO would not be trivial. In enterprise applications, often there exist several constraints on the inputs when processing their business logic. This can lead to major challenges for automated testing approaches to generate such inputs. The input validation is performed at two levels, i.e., in the schema and business logic. The schema level would perform simple checks (e.g., null, format, and range) and checks on constraints related to multiple fields in inputs. Although we have supported the handling of all these constraints defined with `javax` annotations, it is clear that it is not enough in industrial settings. Because not all constraints are fully specified in the interface definitions, e.g., with `javax.validation.constraints`, the validation could be implemented as a utility or with libraries, e.g., `com.google.common.base.Preconditions`, directly in the code of the business logic. To address this, further white-box handling is required to provide more effective gradient to cover the code.

Regarding the validation in terms of business logic, it could perform a check with database and external services. **Data preparation in database and mocking external services would be vital in the testing of industrial RPC-based APIs**, not only for input validation. For databases, currently our approach employs the *SQL handling* in EvoMaster [24] for facilitating data preparation in the database. However, as identified in this study, there might exist some limitations in handling industrial settings cost-effectively, e.g., currently EvoMaster lacks support for composite primary keys. This does limit the performance on code coverage. For instance, we found that a query action with no input parameters is always failing with an exception thrown. In this case, we could do nothing by manipulating the input parameters. Then, with a manual check on the source code, we found that the query is required to have data in the database, but the data fails to be generated due to some unsupported SQL features. In addition, with only SQL query heuristics, it might not be cost-effective to build meaningful links between RPC function calls and inserted data into the database. Smart strategies would be required here to handle industrial RPC-based APIs, such as the enhanced SQL handling strategies for REST APIs [72]. For external services, if we could mock such external services, then the problem might be solved by directly manipulating their responses. Automating such manipulation as parts of the search would be another important challenge.

Another possibility to improve code coverage would be to develop advanced search operators for the RPC domain. For instance, we found that, in the generated tests, function calls in a test may not be related to each other for testing a meaningful scenario. In order to better generate tests with related function calls, we could have strategies to sample function calls by considering dependency among functions (e.g., [76]) in the context of RPC testing, e.g., based on which SQL tables they do access.

An industrial RPC-based API is often a part of large-scale microservices that closely interacts with multiple APIs. Such interaction would result in a huge search space. To test a single API or an API in a small-scale microservice system, testing targets (such as lines of code) could be feasible to reach with an empty database (with/without a small amount of data initialized by SQL script) by manipulating input parameters and data into the database (e.g., `INSERT`). However, testing an industrial API in microservices is not like this case. As the example shown in Figure 1, **the states of other services and databases often have a strong impact in processing business logic that would result in code coverage.** Therefore, all such possible states are considered as a part of search. In this article, we provide descriptive statistics for 54 industrial APIs with $\#LoC_j$ (in total 1,489,959). All of the APIs are parts of one microservice architecture, and there exist hundreds of other APIs that were not used in these experiments. To cope with such a huge complexity of the state, an empty database (as we employed) might limit performance. In addition, as discussed with our industrial partner, they think that **it is important to involve their real historical data (collected in production) in the automated testing**. Likely it would improve the chances to cover more of their business scenarios in the generated tests. Furthermore, such tests would be more valuable for them, e.g., they would consider that all faults identified by these tests would have higher priority to be addressed. However, such data is complex and possibly huge, and how to effectively and efficiently utilize this data with search would be another research challenge that we will address in the future.

Enabling fuzzers on CI would promote their adoption in industrial settings. Our approach is now integrated into one of the industrial development pipelines (same as for the experiments we ran in this article), as a trial to check its applicability into the daily testing activities of our industrial partner. Since all services are developed with the same framework, by studying one of the EvoMaster driver configurations for our approach, our industrial partner has implemented an automated solution to automatically generate such drivers for their services to be tested (e.g., identify all available interfaces and instantiate corresponding clients). For instance, the drivers of the 50 industrial APIs in Table 6 were automatically generated with this automated solution. Regarding the application context, as discussed with our industrial partner, our approach is planned to be employed on the services for generating white-box system tests when the implementation for a requirement of the services is considered as done, as a kind of extra check before putting these new features into production. In addition, the generated tests would be kept for further usage in (1) regression testing of the services and (2) industrial test environment validation as scheduled tasks (e.g., to see whether all services on the pipeline are up and running correctly before QA engineers start manual test sessions).

Flakiness and readability are required to be considered in test generation in industrial APIs. As we found in the industrial APIs, responses could contain information such as timestamps and random tokens, and they could change over time. In order to avoid test failing due to such flakiness, we defined some strategies with general keywords (e.g., date, token, time) observed in the industrial APIs to comment out assertions with such sources. How to systematically identify possible sources of flakiness existing in the industrial APIs (e.g., timestamps, results of SQL queries) and properly handle it during the automation and in the test generation would be another important problem that researchers should address. During the process of reviewing the generated tests with an industrial partner, we found that test readability requires improvement. This is mainly due to very large blocks of code for input instantiations and large numbers of tests in the test suites. As identified in the review, our industrial partner found that the tests that lead to exception thrown are more interesting for them. Therefore, to improve test readability, we now provide a simple strategy to split such tests into different files (the implementation is straightforward, but it is quite useful for our partner). Further possible improvements could be achieved by better organizing the code for large input instantiations, and sorting/splitting tests based on various considerations, e.g., fault classification [55].

7 THREATS TO VALIDITY

Conclusion validity. Our study is in the context of SBST, and our experiments were conducted by following common guidelines in the literature to assess randomized techniques [22]. For instance, with a consideration for the stochastic nature of the employed search algorithms, we collected results for all settings with at least 30 repetitions. The results were interpreted with statistical analysis, such as Mann-Whitney-Wilcoxon U-tests (*p value*) and Vargha-Delaney effect sizes ($\hat{A}_{12}$) for pair comparisons. Regarding fault detection capability, a number of real faults were identified, and those were reviewed together with our industrial partner. Regarding the choice of search budget, since the time cost of executing RPC calls might vary depending on the operating environments (e.g., hardware and OS), we employed a fixed number of RPC calls as the search budget (i.e., 100,000), in order to make our experiment replicable. Studying different settings of search budgets (such as 1 million RPC calls, 1 hour, 24 hours, or 48 hours) might provide us more insights and more concrete evidence for drawing conclusions relating to the choice of the search budget (e.g., more budget might result in better performance on *CS3* and *CS4*, as discussed in Section 5.4.2). However, it is expensive to conduct such an empirical experiment with industrial APIs, as these APIs are typically large-scaled and complex. For instance, with one search budget setting (i.e., 100,000 RPC calls), the computational cost of two settings with 30 repetitions is 129.12 days for the four APIs. Therefore, we consider the experiments with various search budgets as possible future work.

Construct validity. To avoid bias in the results among different settings and techniques, all results to be compared were executed on the same physical environment, e.g., experiments on artificial case studies were deployed on a local machine, and experiments on industrial case studies were deployed on the pipeline of our industrial partners.

Internal validity, Our implementation was tested with various unit tests and end-to-end tests, but we cannot guarantee no fault in our implementation. However, our tool and artificial case studies are open source. This enables further verification on our implementation and replication of our experiments on the artificial case studies by other researchers. Note that, due to the confidential info of the employed industrial case studies, detailed results of these industrial APIs cannot be made publicly available.

External validity. In this study, our approach was assessed with artificial case studies using Thrift and 54 industrial case studies (from one company) using their own RPC framework that is initially built based on Thrift. There might exist a threat to generalize our results to other RPC frameworks or other companies. Experiments on real-world industrial APIs show the usefulness and scalability of our novel techniques in practice. However, these results cannot be replicated by other researchers, as such industrial APIs are not publicly available. Collecting and preparing a corpus of non-trivial open source RPC-based APIs for experimentation (e.g., like EMB [4] for RESTful APIs) will be important for future research work.

8 CONCLUSION

RPC is widely applied in industry for developing large-scale distributed systems, such as microservices. However, automated testing of such systems is very challenging. To the best of our knowledge, there does not exist any tool or solution in the research literature that could enable automated testing of such systems. Therefore, having such a solution with tool support could bring significant benefits to industrial practice.

In this article, we propose the first approach for automatically white-box fuzzing RPC-based APIs, using search-based techniques. To access the RPC-based APIs, the approach is developed by extracting available RPC functions with *RPCInterface*s from the source code. This can enable its adoption to most RPC frameworks in the context of white-box testing. To enable search techniques (e.g., MIO) in the RPC domain, we reformulate the problem and propose additional handling and heuristics specialized for RPC.

The approach is implemented as an open source tool built on top of our EvoMaster [3] fuzzer. A detailed empirical study of our novel approach was conducted with two artificial and four industrial APIs, plus a preliminary (e.g., no fault analysis) study on a further 50 APIs. In total, more than a million lines of business code (excluding third-party libraries) were used in this study. When third-party libraries are considered as well (e.g., for carrying out *taint analysis* [26]), several millions of lines of code were analyzed and executed in these experiments.

Results demonstrate the successful applicability of our novel approach in industrial settings. Our tool extension presented in this article is already in daily use in the Continuous Integration systems of Meituan, a large e-commerce enterprise with hundreds of millions of customers. In addition, to evaluate the effectiveness of our approach in the context of white-box search-based testing, we compared our approach with a gray-box technique. The results show that our approach achieves significant improvements on code coverage. To further evaluate the capability of fault detection, we carried out an in-depth manual review with one employee of our industrial partner on the tests generated by our novel approach. A total of 41 real faults were identified that have now been fixed. Another 8,377 detected faults are currently under investigation.

Considering how widely used RPC frameworks such as Apache Thrift, Apache Dubbo, gRPC, and SOFARPC have been in industry in the last decade, it can be surprising to see how such an important software engineering topic has been practically ignored by the research community so far. One possible explanation is the lack of easy access to case studies for researchers, as these kinds of systems are used to build enterprise applications. Therefore, these systems are seldom available on open source repositories, or online on the internet (i.e., general access web services are usually developed as REST APIs). To be able to empirically evaluate our novel techniques, industry collaborations (e.g., with Meituan) were a strong requirement.

Although our tool extension is already of use for practitioners in industry, more needs to be done to achieve better results. Future work will focus on improving white-box heuristics to increase the achieved code coverage, and how to handle and analyze the interactions with external web services.

Our tool extension of EvoMaster is freely available online on GitHub [3] and Zenodo (e.g., EvoMaster version 1.5.0 [28]), and the replication package for this study can be found at the following link.[1]

REFERENCES

[1] [n. d.]. AFL. https://github.com/google/AFL. Accessed August 26, 2022.

[2] [n. d.]. Dubbo. https://dubbo.apache.org/en/. Accessed August 26, 2022.

[3] [n. d.]. EvoMaster. https://github.com/EMResearch/EvoMaster. Accessed August 26, 2022.

[4] [n. d.]. EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Accessed August 26, 2022.

[5] [n. d.]. GraphQL Foundation. https://graphql.org/foundation/. Accessed August 26, 2022.

[6] [n. d.]. gRPC. https://grpc.io/. Accessed August 26, 2022.

[7] [n. d.]. Intellij IDEA Code Coverage. https://www.jetbrains.com/help/idea/code-coverage.html. Accessed August 26, 2022.

[8] [n. d.]. javax.validation.constraints. https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html. Accessed August 26, 2022.

[9] [n. d.]. OpenAPI/Swagger. https://swagger.io/. Accessed August 26, 2022.

[10] [n. d.]. SOFARPC. https://www.sofastack.tech/en/. Accessed August 26, 2022.

[11] [n. d.]. Status Code in gRPC. https://grpc.github.io/grpc/core/md_doc_statuscodes.html. Accessed August 26, 2022.

[12] [n. d.]. TApplicationException in Thrift. https://javadoc.io/doc/org.apache.thrift/libthrift/latest/org/apache/thrift/TApplicationException.html. Accessed August 26, 2022.

[13] [n. d.]. thrift. https://thrift.apache.org/. Accessed August 26, 2022.

[14] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742–762.

[15] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'17)*. 3–17.

[16] Andrea Arcuri. 2017. RESTful API automated test case generation. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 9–20.

[17] Andrea Arcuri. 2018. EvoMaster: Evolutionary multi-context automated system test generation. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST'18)*. IEEE.

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

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

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

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

[22] A. Arcuri and L. Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219–250.

[23] Andrea Arcuri and Juan P. Galeotti. 2019. SQL data generation to enhance search-based system testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'19)*. Association for Computing Machinery, New York, NY, 1390–1398. https://doi.org/10.1145/3321707.3321732

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

[25] Andrea Arcuri and Juan P. Galeotti. 2020. Testability transformations for existing APIs. In *Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST'20)*. IEEE, 153–163.

[26] Andrea Arcuri and Juan P. Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–34.

[27] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A search-based system test generation tool. *Journal of Open Source Software* 6, 57 (2021), 2153.

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

[29] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API fuzzing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'19)*. 748–758.

[30] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*. IEEE, 207–212.

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

[32] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. 2009. WS-TAXI: A WSDL-based testing tool for web services. In *Proceedings of the International Conference on Software Testing Verification and Validation (ICST'09)*. IEEE, 326–335.

[33] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'22)*.

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

[35] A. Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[36] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. 2002. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing* 6, 2 (2002), 86–93.

[37] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph. D. Dissertation. University of California, Irvine.

[38] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE'11)*. 416–419.

[39] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[40] Vahid Garousi, Matt M. Eskandar, and Kadir Herkiloğlu. 2016. Industry–academia collaborations in software testing: Experience and success stories from Canada and Turkey. *Software Quality Journal* (2016), 1–53.

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

[42] Vahid Garousi, Michael Felderer, Marco Kuhrmann, and Kadir Herkiloğlu. 2017. 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*. ACM, 65–69.

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

[44] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Communications of the ACM* 63, 2 (2020), 70–76.

[45] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A survey. arXiv:2212.14604. DOI: https://doi.org/10.48550/arXiv.2212.14604

[46] Samer Hanna and Malcolm Munro. 2008. Fault-based web services testing. In *Proceedings of the 5th International Conference on Information Technology: New Generations (ITNG'08)*. IEEE, 471–476.

[47] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 345–346.

[48] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. Automatic property-based testing of GraphQL APIs. arXiv:2012.07380. DOI: https://doi.org/10.48550/arXiv.2012.07380

[49] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. https://doi.org/10.48550/ARXIV.2204.08348

[50] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. Association for Computing Machinery, New York, NY, 289–301. https://doi.org/10.1145/3533767.3534401

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

[52] Yin Li, Zhi-an Sun, and Jian-Yong Fang. 2016. Generating an automated test suite by variable strength combinatorial testing for web services. *CIT. Journal of Computing and Information Technology* 24, 3 (2016), 271–282.

[53] Chunyan Ma, Chenglie Du, Tao Zhang, Fei Hu, and Xiaobin Cai. 2008. WSDL-based automated test data generation for web service. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, Vol. 2. IEEE, 731–737.

[54] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 94–105.

[55] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the faults found in REST APIs by automated test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–43.

[56] Evan Martin, Suranjana Basu, and Tao Xie. 2006. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS'06)*.

[57] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful Web APIs. In *Proceedings of the International Conference on Service-Oriented Computing*.

[58] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated black-box testing of RESTful web APIs. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, 682–685.

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

[60] Jeff Offutt and Wuzhi Xu. 2004. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes* 29, 5 (2004), 1–10.

[61] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2018. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering (TSE)* 44, 2 (2018), 122–158.

[62] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A greybox fuzzer for network protocols. In *Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST'20)*. IEEE, 460–465.

[63] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering (EMSE)* 22, 2 (2017), 852–893.

[64] Harry M. Sneed and Shihong Huang. 2006. WSDLTest—a tool for testing web services. In *Proceedings of the 8th IEEE International Symposium on Web Site Evolution (WSE'06)*. IEEE, 14–21.

[65] Wei-Tek Tsai, Ray Paul, Weiwei Song, and Zhibin Cao. 2002. Coyote: An XML-based framework for web services testing. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering, 2002.* IEEE, 173–174.

[66] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane Ducasse. 2018. Deviation testing: A test case generation technique for GraphQL APIs. In *Proceedings of the 11th International Workshop on Smalltalk Technologies (IWST'18)*. 1–9.

[67] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated black-box testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. IEEE.

[68] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software* 182 (2021), 111061.

[69] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial testing of RESTful APIs. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'22)*.

[70] Wuzhi Xu, Jeff Offutt, and Juan Luo. 2005. Testing web services by XML perturbation. In *Proceedings of the16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE, 10Pages.

[71] Man Zhang and Andrea Arcuri. 2021. Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).

[72] Man Zhang and Andrea Arcuri. 2021. Enhancing resource-based test case generation for RESTful APIs with SQL handling. In *Proceedings of the International Symposium on Search Based Software Engineering*. Springer, 103–117.

[73] Man Zhang and Andrea Arcuri. 2022. Open problems in fuzzing RESTful APIs: A comparison of tools. arXiv:2205.05325.

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

[75] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript instrumentation for search-based software testing: A study with RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'22)*. IEEE.

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

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

[78] Yu Zhang, Nanyu Zhong, Wei You, Yanyan Zou, Kunpeng Jian, Jiahuan Xu, Jian Sun, Baoxu Liu, and Wei Huo. 2022. NDFuzz: A non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity* 5, 1 (2022), 1–21.

[79] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering (TSE)* 47 (2021), 243–260. DOI: 10.1109/TSE.2018.2887384

Received 26 August 2022; revised 14 December 2022; accepted 31 January 2023