

Handling Web Service Interactions in Fuzzing with Search-Based Mock-Generation

SUSRUTHAN SERAN, Kristiania University of Applied Sciences, Norway

MAN ZHANG*, Beihang University, China

ONUR DUMAN, Glasgow Caledonian University, United Kingdom

ANDREA ARCURI, Kristiania University of Applied Sciences and Oslo Metropolitan University, Norway

Testing large and complex enterprise software systems can be a challenging task. This is especially the case when the functionality of the system depends on interactions with other external services over a network (e.g., external web services accessed through REST API calls). Although several techniques in the research literature have been shown to be effective at generating test cases in a good number of different software testing contexts, dealing with external services is still a major research challenge. In industry, a common approach is to *mock* external web services for testing purposes. However, generating and configuring *mock* web services can be a very time-consuming task, e.g., external services may not be under the control of the same developers of the tested application, making it challenging to identify the external services and simulate various possible responses.

In this paper, we present a novel search-based approach aimed at *fully automated* mocking of external web services as part of white-box, search-based fuzzing. We rely on code instrumentation to detect all interactions with external services, and how their response data is parsed. We then use such information to enhance a search-based approach for fuzzing. The tested application is automatically modified (by manipulating DNS lookups) to rather interact with instances of mock web servers. The search process not only generates inputs to the tested applications but also automatically configures responses in those mock web server instances, aiming at maximizing code coverage and fault-finding. An empirical study on four open-source REST APIs from EMB, and one industrial API from an industry partner, shows the effectiveness of our novel techniques (i.e., in terms of line coverage and fault detection).

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

Additional Key Words and Phrases: REST, Mocking, API, Fuzzing, Microservices, Automated mock generation, SBST, Search-based software engineering

ACM Reference Format:

Susruthan Seran, Man Zhang, Onur Duman, and Andrea Arcuri. 2024. Handling Web Service Interactions in Fuzzing with Search-Based Mock-Generation . 1, 1 (April 2024), 46 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

*Corresponding author

Authors' Contact Information: Susruthan Seran, susruthan.seran@kristiania.no, Kristiania University of Applied Sciences, Oslo, Norway; Man Zhang, manzhang@buaa.edu.cn, Beihang University, Beijing, China; Onur Duman, onur.duman@kristiania.no, Glasgow Caledonian University, Glasgow, United Kingdom; Andrea Arcuri, andrea.arcuri@kristiania.no, Kristiania University of Applied Sciences and Oslo Metropolitan University, Oslo, Norway.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 ACM.

ACM XXXX-XXXX/2024/4-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

Microservice-based software architecture design helps to tackle the challenges in developing large and complex enterprise software programs. In a microservice environment, relatively small services are interconnected using communication protocols like Hypertext Transfer Protocol (HTTP), Remote Procedure Call (RPC), and Advanced Message Queuing Protocol (AMQP) in a synchronous or asynchronous manner, to execute a common goal. Such independent microservices help to construct a complex system. For example, one microservice might represent payment processing, another authentication, and another content delivery, but altogether they might constitute an entire video streaming service. Even though the microservice architecture disentangles the development process, it adds more complexity to system-level software testing [42]. Likewise, modern web and mobile backends also connect to each other for various reasons, for instance, using services for authentication purposes (e.g., OAuth¹) or for financial transactions (e.g., Stripe² and PayPal³).

Especially during automated testing, dealing with such external dependencies can be tricky since what is executed in the system under test (SUT) purely depends on what is returned from those external web services. Also, testing for specific error scenarios or specific input data might not be possible in a systematic way if the tester has no control on those external services.

In order to deal with such dependencies, *mocking* is a popular technique used in industry [9, 49]. In the existing literature, the term “mocking” is commonly associated with the usage of *mock objects* in *unit testing* [49]. Mockito⁴ is one of the popular libraries used to create such *Mock Objects* for the Java Virtual Machine (JVM) based applications. On the other hand, for *system-level testing* of web/enterprise applications, instead of using *mock objects*, the usage of external service mock servers offers several advantages. For example, the external web service does not even need to be implemented yet or up and running to execute a test. Furthermore, the behavior of the external web service can be extended to test various scenarios (e.g., different returned HTTP status codes). One of the most popular libraries for JVM-based applications that can achieve this is WireMock.⁵

In general, testers manually configure mock servers by specifying the payload that should be returned for every possible request. This can be tedious and time-consuming [2], especially if the SUT interacts with several external services. A complementary approach is to use *record-and-replay*, where each request and response are captured using a pass-through proxy to initiate the mock server rather than configuring it manually [31]. Even though this approach tackles the problem to an extent, it falls short in dealing with the scalability issue as the system scales up in size. Furthermore, this approach is less flexible, as it may not be possible to test edge cases in this way. Another major issue is that often third-party interactions are executed using a provided software development kit (SDK) or a client library. The actual network calls and their input/output objects might be hidden away from the user. Consequently, the developer may not possess any knowledge about these network communications unless the library or SDK is reverse-engineered, and this can be very time-consuming. Consequently, the process of handwritten mocks becomes further complex in these cases. By taking all of those gaps into consideration, we aim to develop techniques to generate mock objects in an automated manner.

In this paper, we present novel search-based, white-box fuzzing techniques to automatically generate mock external web services for the JVM-based applications. White-box testing is dependent on the programming language, as the code must be analyzed. The JVM-based languages, especially Java and Kotlin, are two of the most widely used in industry. Despite the popularity of Java and Kotlin,

¹<https://oauth.net>

²<https://stripe.com/>

³<https://paypal.com/>

⁴<https://site.mockito.org/>

⁵<https://wiremock.org/>

there are other languages/runtimes that are widely used to develop web APIs, such as JavaScript, Go, and .NET. However, supporting diverse programming languages for white-box testing is a significant undertaking, which is often viewed solely as technical work by researchers. Therefore, for technical and practical reasons our objective is to focus solely on case studies based on Java and Kotlin. Moreover, JVM capabilities such as Java Reflection and bytecode instrumentation enable us to implement our approach. Notably, bytecode instrumentation is a technique employed to modify or analyze the bytecode of a program at runtime (e.g., JVM and .NET) or during the compilation process. Through bytecode instrumentation, we can automatically analyze all interactions with external web services at runtime, and we can collect heuristics (i.e., response schema from the used Data Transfer Objects (DTO)). Similarly, Java Reflection is a technique used in Java that allows a program to inspect and manipulate the program at runtime. By utilizing the collected information using both techniques, we can create mock web services with proper information automatically. To enable our novel techniques, we analyze all interactions with external services using bytecode instrumentation and automatically create mock web servers where responses from those are utilized as part of search-based testing.

Regarding data transfer formats, we focus on JavaScript Object Notation (JSON) payloads, as JSON-based DTOs are one of the widely used formats in the industry [41, 46]. Furthermore, in JSON-based DTOs, the structure of responses can be inferred automatically by instrumenting parsing libraries (e.g., Gson⁶ and Jackson⁷) and by performing taint analysis. Taint analysis is a software program analysis technique that monitors data flow within a program, facilitating the identification of software faults, particularly software vulnerabilities and data leaks. Nevertheless, this approach has the potential to be extended to other schema formats as well, like XML.

Our novel techniques can be adapted and applied to any white-box fuzzer that works on backend web/enterprise applications (or any application that requires communications with external services over a network), like for example RESTful APIs. For the experiments in this paper, to reduce the workload required to assess our novel techniques, we chose to extend an existing search-based fuzzer instead of developing a new one from scratch. We opted an open-source fuzzer EvoMASTER [13], since it performs best compared to other black-box fuzzers [30, 54]. Moreover, to the best of our knowledge EvoMASTER is the only open-source fuzzer that supports white-box testing for web applications [25]. Our extension enables EvoMASTER to generate mock external web services with zero modification required to the System Under Test (SUT). Furthermore, we generate self-contained test suites with integrated mocking capabilities to simulate the same production behavior without the need for external services to be up and running.

Our techniques are evaluated on five RESTful APIs (four open-source from the EMB⁸ corpus [16] and one industrial), showing statistically significant improvement in code coverage and fault detection. To enable the replicability of our experiments, our extension to EvoMASTER is open-source,⁹ which includes as well the experiment scripts used in the different empirical studies.¹⁰

The contributions of this paper are:

- A search-based approach in which HTTP mock servers are automatically instantiated when fuzzing web/enterprise applications.
- A use of taint-analysis to automatically infer the syntactic structure of the JSON payloads expected to be returned from these external services.

⁶<https://github.com/google/gson>

⁷<https://github.com/FasterXML/jackson>

⁸<https://github.com/WebFuzzing/EMB>

⁹<https://github.com/WebFuzzing/EvoMaster>

¹⁰<https://zenodo.org/records/10932122>

- An integration of search-based test generation, in which both inputs to the tested API and outputs from the mocked services are part of a search-based optimization.
- An open-source extension of the existing search-based white-box fuzzer EvOMASTER.
- An empirical study on both open-source and industrial APIs, providing empirical evidence that our proposed techniques increase line coverage (e.g., up to +20% on average on one of the tested APIs) and find more faults (e.g., 8.6 more on average).

This article is an extension of a conference article [48]. In this extension, to further improve performance we present new techniques, such as *Harvester* in Section 4.6. We also improved the handling of heuristics related to external web service information in Section 4.2. Additionally, to better analyze the effectiveness of our novel techniques, we conducted new experiments on artificial APIs (discussed in Section 6.2), and experiments with an additional open-source API answering two new research questions, which we discuss in detail under Section 6.

This article is structured as follows. Section 2 delves into the related work. The motivation and background of the work is discussed in Section 3. The details regarding automated mock external web service generation are discussed in Section 4. Section 5 contains the specifics of how we ensure that the mock external web services maintain the consistent behavior in the self-contained test suites as they do during the search. Section 6 includes the details of the empirical study, along with the results and their discussion. Potential threats to the validity are analyzed in Section 7. Finally, Section 8 concludes the article and discusses future work.

2 Related Work

2.1 Fuzzing

Fuzzing is a software testing technique that uses random or invalid inputs to be given to the software to analyze its behavior [39]. Fuzzing is one of the most effective methods to detect software faults (in particular security-related faults) and to achieve higher code coverage [22, 34, 59]. There are several success stories about fuzzing techniques, in particular, to test parser libraries written in C/C++ [38], with popular tools such as AFL [1], as well as network devices [45, 58]. Recently, large and complex interconnected enterprise software systems with millions of lines of code have been tested efficiently using fuzzing techniques [55]. Particularly, white-box and black-box fuzzing of REST-based web services has gained a major amount of attention among researchers lately [25, 30, 54]. White-box fuzzers have been proven to be highly effective in numerous instances [5, 23, 37]. In white-box fuzzing, internal details of the SUT, such as its source code, binaries, or bytecode, will be accessed. This information can be used to design heuristics to improve the search and produce better test cases with better line coverage and fault detection capabilities. By leveraging the advantages of search-based white-box fuzzing, in this paper we present techniques to automatically generate mock external web services to further improve fault-finding and code coverage.

2.2 Mocking

In industry, it is a common practice to use mocking techniques during unit testing to deal with dependencies. The use of *mock objects* is one of the common practices for handling dependencies in *unit testing* [32, 40, 49, 51]. A mock object can be used to represent the response to a call made to an external service or database. Especially in tests, this can be programmatically configured without executing the original code. Different libraries help instantiate and configure *mock objects*. For JVM-based applications, there exist libraries that significantly simplify the writing of mock objects [40], such as Mockito,⁴ EasyMock,¹¹ and JMock.¹²

¹¹<https://easymock.org/>

¹²<https://www.jmock.org/>

Mocking helps to expedite software testing and allows testing of the functionalities individually [49]. Although mocking has practical benefits, it can still require a significant time and effort investment when mocks are created manually, as the returned values of each method call on such mocks need to be specified.

Besides the case of writing unit tests manually, automated unit test generators can be extended to create mock objects as well for the inputs of the classes under test (CUT). An example is the popular EvoSuite [21], which can generate mock object inputs using Mockito [9]. Similarly, Pex can set up mock objects related to the file system APIs [36].

Interactions with the environment can be mocked away if they are executed on method invocations of input objects (as such input objects can be replaced with configurable mocks in the test cases). However, it cannot be mocked away directly if it is the CUT itself that is doing such interactions with the environment. This is a significant concern, especially if the CUT is performing operations on the file system (as random files could be created and deleted during the test generation, which could have severe consequences). To overcome such a major issue, tools such as EvoSuite can do bytecode instrumentation on the CUT to replace different types of environment operations with calls on a virtual file system [7] and a virtual network [8, 28]. Still, there are major issues in creating the right data with the right structure (e.g., files and HTTP responses) returned from the calls to the virtual environment [28].

In system-level testing, mock objects cannot be directly used since the SUT is executed through the user interfaces (e.g., a GUI or network calls in the case of web services). Consequently, there is still the issue of how to deal with network calls to external services during testing. During testing, instead of communicating with the actual external services, the SUT should be modified to communicate with a mock server, on which the tester has full control over how responses are returned. This may potentially help with the testing of additional scenarios.

Mocking entire external web services is different from mocking the network and dependencies through mock objects, although they share the same goal. From the perspective of the SUT, it is not aware of whether it is communicating with a real or mocked service, as it will still use the same code to make network calls (e.g., HTTP over TCP) and to parse the responses. There are several tools and libraries available in the industry that can help to set up and run a mock external web service with less effort, such as Mockoon,¹³ Postman,¹⁴ and WireMock.⁵ All of them share common functionalities to create configurable mock HTTP-based web services for testing. However, to the best of our knowledge, none of them have the functionality to automatically generate such mock web services.

Creating handwritten mock external web services can be tedious and time-consuming [2]. Furthermore, the process becomes convoluted if the third-party interactions are executed utilizing a provided software development kit (SDK) since the knowledge about interactions may not be available unless provided or the SDK is reverse-engineered. The *record-and-replay* approach can be used to reduce the effort. Instead of manually configuring the mock servers, a pass-through proxy will capture every request and response and utilize them to configure the mock server. However, the external web service should be owned by the same developers of the SUT to be able to emulate more use cases (especially error scenarios), apart from the common “happy path” scenarios. When the system scales up in size, using handwritten mock web services is not a scalable approach. Meanwhile, record-and-replay would still be a feasible method.

The work done in [8, 28] is perhaps the closest in the literature to what we present in this paper. However, there are some significant differences. First, we do not use a limited, artificial virtual

¹³<https://mockoon.com/>

¹⁴<https://www.getpostman.com/>

network but rather make actual HTTP calls using external mocked web services (using WireMock). To be able to achieve this, we have to overcome several challenges, especially when dealing with SSL encryption, and when multiple external web services use the same TCP port. Additionally, to enable effective system-level testing, we need to automatically generate mock responses with valid syntax (e.g., in JSON representing valid DTO for the mocked services), regardless of whether any formal schema is present or not (e.g., the work in [28] requires specifying XML Schema Definition (XSD) files manually with the schema of the response messages). Moreover, in [28], the authors evaluated their techniques only on artificial classes. On the other hand, we show that our novel techniques can scale to real-world systems (e.g., including an industrial API).

To the best of our knowledge, there is a lack of techniques that can fully automate the process of automatic generation of mock external web services in the scientific literature. In addition, the topic of fully automated mock generation of external web services for system-level white-box testing has not received enough attention in the research literature, despite its importance in industry (e.g., considering all the existing popular libraries/tools such as WireMock and Postman). Moreover, previous studies, conducted in mocking the networking interfaces of the Java standard library to use as an input space for search-based fuzzers, have shown significant improvement in code coverage [8, 9]. By simulating the network, the researchers were able to increase the code coverage. Additionally, they were able to produce self-contained test cases which can run independently with all the networking dependencies. Regardless, the studies [8, 9] focused on the individual classes under test rather than on a system level. Furthermore, in [8], the authors used *Mock Objects* only to simulate network connections using a virtual network, and not handling any data sent or received.

2.3 Grammars In Fuzzing

Grammar-based fuzzing is another software testing technique where highly effective test inputs are generated based on user-specified grammar. With the help of formal grammar specifications, fuzzers can generate and manipulate well-formed input data. In a previous study [43], the authors combined search-based testing with grammar-based fuzzing and demonstrated that combining those leads to significant improvement in branch coverage for JSON-related classes. Furthermore, previous studies [20, 52] indicate that the use of grammar-based fuzzing significantly improves line coverage and fault detection rate, especially with common input formats such as JSON. Grammar specification typically can be user-specified or can be inferred from documentation (e.g., OpenAPI). Another approach is mining such grammars [26].

Existing literature mainly focuses on grammar-based inputs for endpoints of the SUT. In this work, we focus on automatically generating external web services. Therefore, we need to know about the possible JSON response schemas. There is a possibility to infer the schema from the documentation if it is available. However, this may not always be possible. For instance, if the SUT is using an SDK of a third-party vendor, it may not always be possible to find the input format to create the grammar for further usage. To overcome this challenge, we take an entirely different approach compared to the previous literature. In this paper, we use instrumentation for schema extraction rather than relying on grammar-based techniques such as grammar mining or user-provided grammar (detailed in Section 4.4). The reason is that, for REST APIs, JSON is the most common format for data exchange, and, on the JVM, there are two specific libraries that are used for parsing JSON data, which is usually mapped into Java/Kotlin classes. This domain knowledge, if exploited correctly, can significantly simplify and improve the accuracy of the automated inference of the data grammars.

```

1 public interface HttpConnector {
2     /**
3      * Opens a connection to the given URL.
4      */
5     HttpURLConnection connect(URL url) throws IOException;
6
7     /**
8      * Default implementation that uses {@link URL#openConnection()}.
9      */
10    HttpConnector DEFAULT = new ImpatientHttpConnector(new HttpConnector() {
11        public HttpURLConnection connect(URL url) throws IOException {
12            return (HttpURLConnection) url.openConnection();
13        }
14    });
15 }

```

Fig. 1. The code snippet taken from *catwatch*¹⁵ shows how an external web service connection is established inside the SUT.

2.4 EvoMaster

EvoMASTER is an open-source tool used for fuzzing web services using search-based techniques with white-box and black-box fuzzing [5, 13, 17]. EvoMASTER provides client-side drivers (i.e., embedded and external) to enable white-box fuzzing and requires “no code modification” on the SUT, as instrumentation is done automatically. The key difference between embedded and external drivers is whether the SUT and EvoMASTER run on the same JVM (i.e., embedded) or in separate JVMs (i.e., external). However, the driver requires writing the necessary steps to cover the three main stages of the API’s lifecycle, such as start, reset, and stop. This needs to be manually specified by the user, as APIs can be written in different frameworks and libraries, each one with its own idiosyncratic way of performing these steps. Furthermore, EvoMASTER features different search algorithms (e.g., MIO [3] extended with adaptive-hypermutation [53]) and fitness functions (e.g., using advanced testability transformations [11, 15] and heuristics based on the SQL commands executed by the SUT [10]) to generate system-level test suites. In this paper, we implement our novel search-based, white-box fuzzing techniques as an extension of EvoMASTER, to enable it to generate system-level test suites with automatic mocking of external web services.

3 Motivation

In this paper, our goal is to analyze the impact of automatically handling external web service dependencies during white-box fuzzing. For this purpose, we utilize the EMB corpus of web services [16], which has been used as benchmark in several previous studies [30, 47, 50]. We selected all four APIs that are using external web services for various purposes, out of current 29 from the EMB corpus.⁸ Those APIs were used in previous studies without automatic mocking of external web services [54]. As an example, the code block shown the Figure 1 is taken from the *GitHub* SDK used in one of our APIs (i.e., *catwatch*¹⁵).

The class `HttpConnector` is responsible for opening a connection to the given URL using `java.net.HttpURLConnection`. The connectivity with the external web service decides the further execution of the program since the program depends on the response from the external web service

¹⁵<https://github.com/zalando-incubator/catwatch>

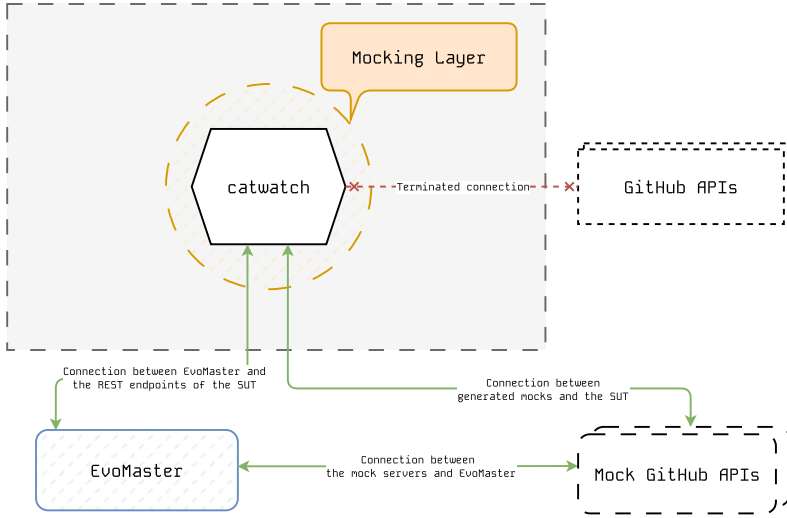


Fig. 2. The diagram depicts how one of our selected APIs (i.e., *catwatch*) depends on web services to execute its goals. The *Mocking Layer* is the virtual boundary that ends the connection with real external web services.

to execute its goals. In this particular example shown in Figure 1, the *GitHub* SDK will try to connect to *GitHub* web APIs using `URLConnection.openConnection()` to fetch information, as visible in Line 12. For instance, if we manage to mock the particular external web service (e.g., *GitHub* Web API), we can test the SUT for edge cases that may not be handled inside the SUT. Figure 2 gives a high level representation of how this would work.

Scenarios like this motivated us to build mocking techniques to improve code coverage and fault detection. Since all of our selected APIs are written in Java and Kotlin, we implemented and tested our novel techniques for the JVM ecosystem. Moreover, we implemented our novel techniques as an extension to EvoMASTER. Due to this fact, all the decisions made, as discussed in the following sections, favor the compatibility with JVM. Even though we implemented and tested our techniques in the JVM ecosystem, the same techniques can be applied to other languages and runtimes (e.g., C#).

4 Automated Mock External Services Generation

In this work, we focus on HTTP services that use JSON as data-transfer format, as those are the most common types of web services in industry [41]. To achieve this goal, a good deal of research and technical challenges need to be addressed.

As our novel techniques are designed as an extension to an evolutionary search process, we first start from describing the existing architecture of EvoMASTER in Section 4.1 and how our novel techniques fit into it. Then, we discuss the details of our novel techniques in five different subsections regarding instrumentation (Section 4.2), the mock server (Section 4.3), genotype representation (Section 4.4), schema inference (Section 4.5), and harvesting responses (Section 4.6).

4.1 EvoMaster Search Process

Figure 3 shows a high level overview of the components of EvoMASTER, and their relations with the SUT. An EvoMASTER *Driver* is responsible to start, stop and reset the SUT. When starting the

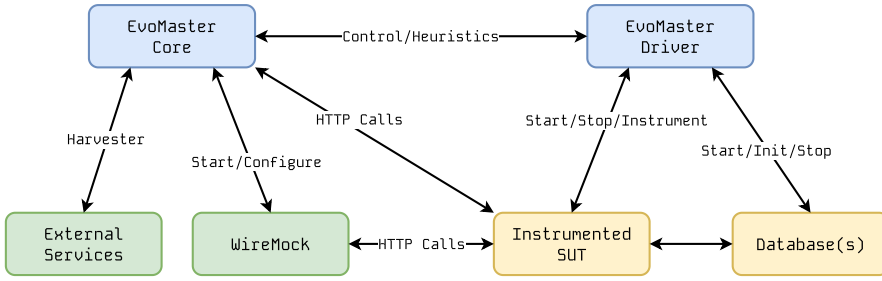


Fig. 3. High level overview of the components of EvoMASTER and their relations to the SUT.

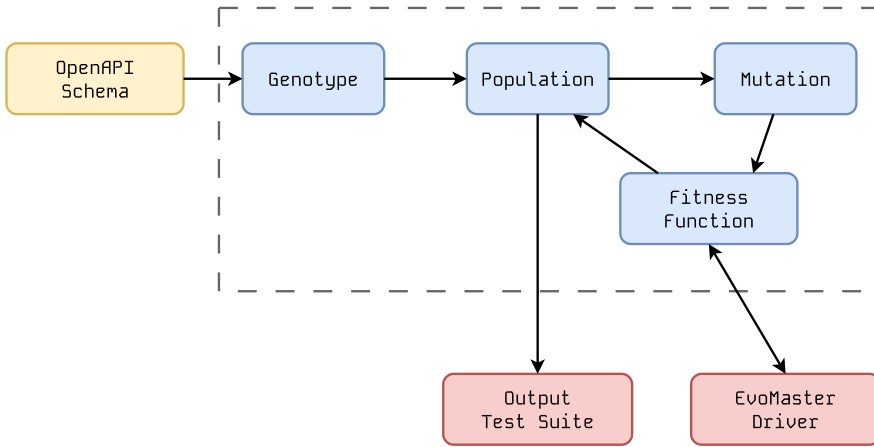


Fig. 4. Simplified overview of the evolutionary search process in EvoMASTER.

SUT, the Driver will also instrument the bytecode of the SUT, which needs to be done only once per class in the application. The instrumentation is used to compute different white-box heuristics [4], including advanced testability transformations [12, 15]. The bytecode instrumentation for our novel techniques (Section 4.2) is done here. If the SUT needs any databases (e.g., Postgres, MySQL or MongoDB) up and running, the Driver is responsible to start it (e.g., via Docker). The Driver and SUT can run in the same process (Embedded Driver) or as 2 separated processes (External Driver). The Driver is developed as a REST API, which can be accessed by the Core process.

The EvoMASTER Core process contains the fuzzing engine. It will evolve new test cases, and evaluate them by calling the SUT directly. Then, it will request the Driver to extract the white-box heuristic scores to compute the fitness value of each executed test case.

If the SUT tries to communicate with an external service, such communication will be blocked by our instrumentation. The Driver will then inform the Core of this event. It is the the Core that will start WireMock instances to use instead (Section 4.3).

Figure 4 shows a simplified overview of the evolutionary search process in EvoMASTER, which is executed inside the Core process. Based on the OpenAPI schema of the SUT (which needs to be provided as input to the tool), EvoMASTER creates a genotype representation for evolving test

cases representing HTTP calls toward the tested REST API. We do not send random bytes on a TCP socket, but rather craft HTTP calls that are valid according to the given schema.

Based on this genotype representation, different search algorithms could be used to evolve test cases. In particular, by default EvoMASTER uses MIO [3] enhanced with adaptive hypermutation [53]. At a high level, these evolutionary algorithms keep a *population* of evolving individuals (e.g., initialized at random, but according to the schema), and then start an evolutionary loop of selecting some of them (based on some fitness score criteria), *mutate* them (e.g., do small random modifications), evaluate their *fitness*, and then decide whether the mutated individuals should be saved back in the evolving populations. This loop process is executed continuously, until the conditions of a stopping criterion are met (e.g., 1 hour has passed). At the end of the search, from the evolved populations a test suite is generated (e.g., in Java using JUnit).

At each fitness evaluation, based on our instrumentation, we can detect each time the SUT tries to interact with an external service. Those interactions are automatically redirected to the WireMock instances started by the Core. Based on which requests are then made by the SUT toward such WireMock instances, the Core will extend the genotype of the evolving test cases to setup responses in the WireMock instances (Section 4.4). Based on how the payloads sent by WireMock will be marshalled by the SUT, the genotype is further extended to evolve payloads that will not make the SUT crash when parsing them (Section 4.5).

Finally, as shown in Figure 3, the Core can make direct calls toward the external services to collect possible payload examples to speed-up the search (Section 4.6).

4.2 Instrumentation

During the search, we need to automatically detect the hostnames/IP addresses and ports of all the external services the SUT communicates with. To achieve this, we applied a form of *testability transformation* [27], relying on the infrastructure of EvoMASTER to apply *method replacements* for common library methods [11] (e.g., the APIs of JDK itself). We extended the white-box instrumentation of EvoMASTER with new method replacements for networking APIs. When classes are loaded into the JVM, method calls towards those APIs are replaced with our own developed methods. Those can track what inputs were used, and then call the original API without altering the semantics of the SUT.

More specifically, we create method replacements for Java network classes, such as `java.net.InetAddress`, `java.net.Socket`, and `java.net.URL`. This enables us to collect and manipulate various information related to the connection such as hostname, protocol, and port from the different layers of the Open Systems Interconnection (OSI) model [60]. This approach enables us to capture and analyze the HTTP requests made to the external web service with all their parameters (e.g., URL path, query parameters and headers). Additionally, this allows us to reroute calls made to an external web service to our mock server.

From the first call made to the external web service, we can capture the external web service information through instrumented networking classes such as `java.net.InetAddress`, `java.net.Socket`, and `java.net.URL`. However, in `java.net.InetAddress` case, we are limited with information about the host address. In such event, we rely on other instrumentation such as `Socket` to collect the complete information (e.g., including port number) about the external web services on the subsequent steps.

Meanwhile, we encountered SSL certificate-related issues when dealing with external web service connections done using HTTPS. Typically, when an HTTP client establishes an SSL connection, the client will validate the server using the PKIX algorithm (Public Key Infrastructure X.509). This operation is usually handled by the Java Secure Socket Extension (JSSE) framework, which is part of the Java Runtime Environment (JRE). If the certificate is invalid, the connection will not be

```

1 @GetMapping(path = ["/string"])
2 fun getString() : ResponseEntity<String> {
3
4     val url = URL("http://example.test:8123/api/string")
5     val connection = url.openConnection()
6     connection.setRequestProperty("accept", "application/json")
7     val data = connection.getInputStream().bufferedReader().use(BufferedReader::
8         readText)
9
10    return if (data == "\"HELLO THERE!!!\"") {
11        ResponseEntity.ok("YES")
12    } else {
13        ResponseEntity.ok("NOPE")
14    }
15 }

```

Fig. 5. A small example of a REST endpoint written in SpringBoot with Kotlin, making an HTTP call towards an external service using a URL object.

established. As we are running mock servers within the *loopback* address range, it was necessary to find a way to circumvent this verification. In order to maintain the zero-code modification requirement, we are not able to make any modifications to the SUT to utilize our own SSL certificate. Moreover, using our own certificate will make things much more complex. Therefore, we tackle this challenge utilizing instrumentation. For this purpose, we implemented a class to facilitate a custom `X509TrustManager` and `HostnameVerifier` to allow all hostnames during the certificate checks.

Additionally, we created method replacements for various third-party classes from popular networking libraries, such as `okhttp3.OkHttpClient` and `com.squareup.okhttp.OkHttpClient`¹⁶ to use our implementation of `X509TrustManager` and `HostnameVerifier`. This covers the most common cases of dealing with HTTPS calls on the JVM. However, technically there could be other less popular HTTPS libraries in use by the SUT. In these cases, our techniques would not work, albeit supporting such libraries would be just technical work. Arguably, though, in our current implementation we cover the majority of the cases.

Furthermore, information gathered from the other instrumented classes such as `java.net.URL`, `com.fasterxml.jackson.databind.ObjectMapper`, and `com.google.gson.Gson` helped to gather information about the HTTP requests made throughout the search. The collected information is used throughout the search to improve the mock generation. Mock generation utilizing the retrieved information is detailed in Section 4.4.

To illustrate, let us consider the example in Figure 5. This is a small artificial example of a REST endpoint written in SpringBoot¹⁷ making an HTTP call toward an external service using a URL object. We utilized a reserved top-level domain name (i.e., `example.test`), as outlined in RFC 2606, as a dummy external web service in this example.¹⁸ If the request is successful with the response "HELLO THERE!!!", the application will respond with an HTTP 200 status code with the string body "YES", otherwise with a "NOPE".

¹⁶<https://square.github.io/okhttp/>

¹⁷<https://spring.io/projects/spring-boot>

¹⁸<https://www.rfc-editor.org/rfc/rfc2606.html>

```

1 public URLConnection openConnection() throws java.io.IOException {
2     return handler.openConnection(this);
3 }

```

Fig. 6. Code block from `java.net.URL` which is responsible for opening a connection.

As shown in Figure 5, when the `openConnection()` call is executed at Line 5, a connection will be established with the remote destination (at the fictional example `example.test:8123`). Once the connection is established, by invoking `getInputStream()` method from `java.net.URLConnection`, we can capture and read the response (Line 7).

The code block shown in Figure 6 contains a part of the implementation of `java.net.URL` that is used as the HTTP client library to make a connection. The handler in Line 2, represents `URLStreamHandler`. Depending on the protocol (e.g., HTTP, File and FTP), `openConnection` will invoke the respective implementation.

Since we only focus on HTTP-based connections, our instrumentation will replace the call `url.openConnection()` with our custom `URLClassReplacement.openConnection(url)`. Inside this function, we can get all the information about the HTTP connection, and create a new connection which will connect to a mock server instead of connecting to `example.test:8123`.

Note that this kind of instrumentation is applied to all classes loaded in the JVM, and not just in the business logic of the SUT (i.e., even to third-party libraries used in SUT). For example, if the SUT is using a client library to connect to the external services, this approach will still work. As an example, one of our selected APIs (all APIs used in our empirical study are detailed in Section 6), namely *catwatch*, uses the Java library `org.kohsuke:github-api`, which internally connects to `https://api.github.com`.

Meanwhile, during testing, we want to avoid messing up with the connections with controlled services, such as databases. For this reason, we do not apply any modification when the SUT connects to services running on the same host (e.g., *localhost*). This is not a problem when dealing with external services running on the internet (e.g., `https://api.github.com`) or outside of the of SUT host. However, there is a possibility that the external services could run on the same host using different ports. For example, multiple services running in *localhost* using different ports. In general, there is a possibility to change the endpoint address through the SUT configuration. We faced this issue in one of our APIs used as case study mentioned in Section 6 (i.e., *cwa-verification*). In the case of *cwa-verification*, this was easily achieved by manually overriding the configuration option `--cwa-testresult-server.url`.

The code block shown in Figure 7 is an example implementation of the instrumentation. Through instrumentation, we can control the connection made to external web services and collect various heuristics (e.g., JSON schemas) to improve the search. At the same time, throughout the search, we ensure that method replacements preserve the semantics of the actual methods. For example, in `InetAddress` method replacement, if we are unable to resolve the remote host, an `UnknownHostException` will be thrown like in the real class.

Figure 8 depicts how the decision is made when the SUT attempts to establish a connection using `Socket` to an external web service. To establish a connection using `Socket`, it is required to provide the `SocketAddress`. Such address can be presented using an `InetSocketAddress`. In `InetSocketAddress`, it is possible to directly use the remote hostname information, otherwise to use `InetAddress`. Regardless of how an address is represented at the *Socket* level, on the bottom layer, Java uses `InetAddress` to perform the hostname resolution. Due to this, we created method replacements for both `InetAddress` and `Socket`. Figure 9 has the sequence diagram of establishing

```
1 @Replacement(  
2     type = ReplacementType.TRACKER,  
3     category = ReplacementCategory.NET,  
4     id = "URL_openConnection_Replacement",  
5     replacingStatic = false,  
6     usageFilter = UsageFilter.ANY  
7 )  
8 public static URLConnection openConnection(URL caller) throws java.io.IOException  
9 {  
10     Objects.requireNonNull(caller);  
11     URL newURL = ExternalServiceUtils.getReplacedURL(caller);  
12  
13     return newURL.openConnection();  
14 }
```

Fig. 7. Implementation snippet of our custom method replacement implementation for the JDK method openConnection(URL caller) in the URLConnection class.

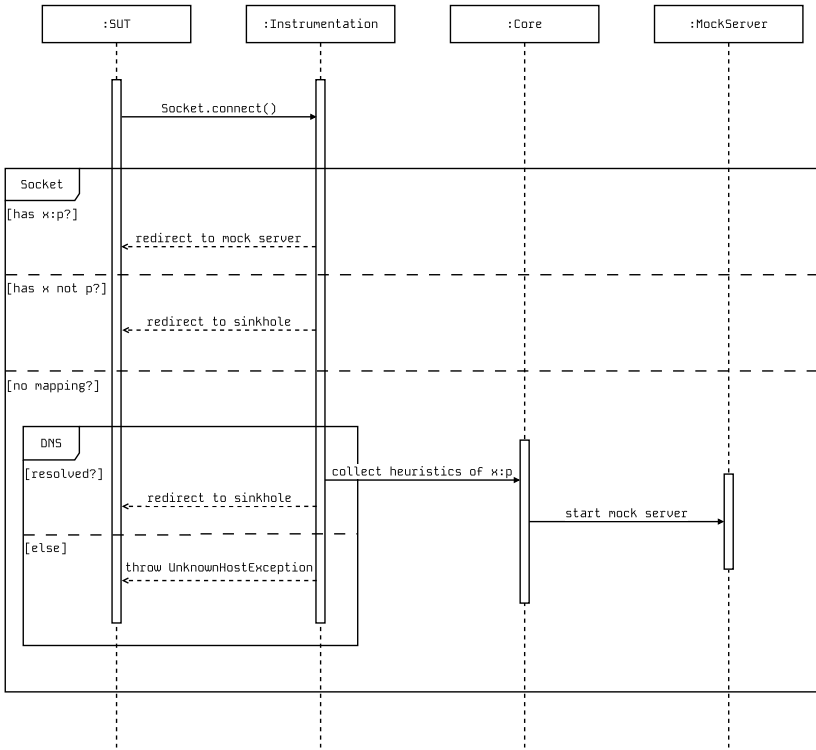


Fig. 8. Sequence diagram explaining the flow of socket connection between SUT and WireMock. The remote hostname is represented with *x* and port is represented with *p*.

a connection to an external web service using `Socket` while using `InetAddress` to represent the remote hostname in `InetSocketAddress`.

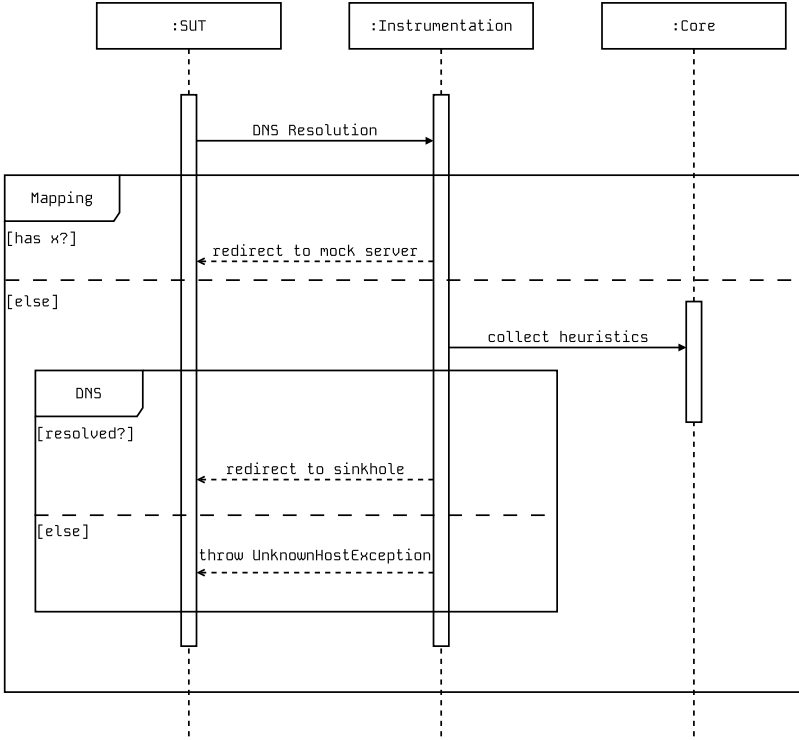


Fig. 9. Sequence diagram explaining the flow of Domain name resolution when a connection is initiated by the SUT to an external web service. The remote hostname is represented with x .

In *Socket*, we can collect information about the protocol, hostname, and port of the external web service. Using such information, we can make a decision about how the connection should be established based on the state of the search as shown in Figure 8. Meanwhile, through the *InetAddress* method replacement, we can only collect information about the hostname. Since the protocol and port information are missing, this limits us from initiating a mock server for the respective external web service at this stage. Once the protocol and port information are collected through other method replacements (i.e., *Socket*), a mock server will be initiated in the subsequent stages of the search.

As shown in Figures 8 and 9, in the *collect heuristics* phase, information about the external web service such as hostname, protocol, and port will be collected through method replacements. Apart from the Java networking classes, we created method replacements for third-party libraries such as *OkHttp*,¹⁶ since they use their mechanisms to establish a connection.

We handle the collected information related to DNS (e.g., remote hostname and resolution status) separately from the information related to external web services (e.g., protocol, hostname, and port). We collect information regarding the remote hostname and the resolution status and store it using *HostnameResolutionInfo* for further use throughout the search. Meanwhile, complete information about the external web service such as protocol, hostname, and port will be stored using *ExternalServiceInfo*. Furthermore, this will be used to initiate a mock server to a respective external web service.

Additionally, we use a network *sinkhole* to prevent the SUT from connecting to the actual service during the search. A network *sinkhole* is a security technique that is used to redirect network traffic to a controlled environment for analysis or mitigation purposes. In EvoMASTER, we allocated an address from the loopback (i.e., 127.0.0.2) range to act as a *sinkhole*, in which the user should ensure no services are running (this is the typical common case, as loopback addresses besides 127.0.0.1 are seldom used). The *sinkhole* address will be used to prevent the SUT from connecting to the actual service. As shown in Figure 8 and Figure 9, if the external web service is resolved, when the connection is being established, it will be redirected to the *sinkhole*. Otherwise, `UnknownHostException` will be thrown. Additionally, this allows testing the case where the external web service is inaccessible.

Figure 7 contains a code block taken from one of the implemented method replacements. `ExternalServiceUtils.getReplacedURL()` method collects the complete information about external web services inside the `URL` and inside the `OkUrlFactory` from `OkHttp`. Apart from that, this method redirects the connection depending on the state of the search, either to the *mock server* or to the *sinkhole*. We use a similar approach in `Socket` to manage the connection being made. However, in the `InetAddress` method replacement, we only replace the actual external web service address (i.e., `hostname`) with a *mock server* address or with the *sinkhole* address.

When a `HostnameResolutionInfo` is captured through the EvoMASTER instrumentation, the `HttpWsExternalServiceHandler` will attempt to allocate a new IP address from the available loopback range, if it is already not allocated. The allocated address will later be used to run a mock server for the respective external web service. Additionally, it will create a mapping between the *remote hostname* and *local IP address*, which will be used inside the instrumented SUT for redirecting the connection to the mock servers. As the next step, the mutator will use the available `HostnameResolutionInfo` to create `HostnameResolutionAction` respectively and will add to the individual as an initialization action. This is to enable the testing of cases in which a hostname does not resolve. Once the search is completed, `HostnameResolutionAction` will be used to set up DNS cache information using `DnsCacheManipulator` in the generated tests.

4.3 The Mock Server

Each time an `ExternalServiceInfo` is registered, the `HttpWsExternalServiceHandler` will check for local address mapping and an active mock server. If there is no active mock server available for the respective `ExternalServiceInfo`, `HttpWsExternalServiceHandler` will initiate one.

Writing a mock web server capable of handling HTTP requests is a major engineering effort. Since the focus of this work is not to develop such a tool, we decided to use an existing mock server library. As the implementation of our techniques is targeting the JVM (e.g., for programs written in Java and Kotlin), we need a mock server capable of running inside the JVM and be able to be programmatically configurable. We chose `WireMock`⁵ for this purpose as it aligns well with our requirements.

`WireMock` can be configured manually (i.e., by writing all the responses as configurations) or programmatically (i.e., during runtime). Apart from that, `WireMock` supports record-and-replay as well to configure the mock server. By reconfiguring the software to connect to the `WireMock` proxy server instead of the target destination, it is possible to capture the requests and responses. Hereafter, captured requests and responses will be remapped as stubs, so the mock server can mimic the behavior of the actual server. Nonetheless, using this approach replicates cases in which the external service works as expected unless unanticipated behaviors are simulated during the recording.

For our purpose, we use WireMock programmatically to handle external web service calls. WireMock allows us to create and run HTTP/S mock servers, and to easily imitate external web services. However, the first challenge is how to instruct the SUT to communicate with these WireMock instances instead of the actual external services. This may not be readily configurable, particularly if the network communications are performed within a client library for the external web service. We addressed this issue by white-box *instrumentation*, as detailed in Section 4.2.

Redirecting the traffic to the mock server is not always a trivial task. We tackle this challenge by instrumenting the necessary classes used by the SUT inside the JVM. This instrumentation allowed us to reroute the external web service requests to the respective WireMock server (recall the example in Figure 2 discussed in Section 3).

In general, when a WireMock server receives an incoming request, it will check any registered stub to see how it should respond to the request. A stub can be configured using a static URL path (e.g., `/api/v1/`) or a regular expression-based path (e.g., `/api/v1/.*`). WireMock provides support to extend the request pattern matching using other HTTP request parameters such as headers, body payload, HTTP method, query parameters, cookies, basic authentication, and multipart form data. EVO MASTER is designed in a way to restart the search each time when there is an exception. At the same time, the default behavior of WireMock is to throw an exception if a request pattern is not configured which caused problems during the search. To overcome this challenge, we set WireMock to return a response with an HTTP 404 for all requests if a stub is not present.

Once initiated, all the requests will be redirected to their respective WireMock servers using automated instrumentation, to change the mapping from hostnames to IP addresses. After a test case is executed, we can query WireMock to retrieve the captured request patterns. The default behavior of returning a 404 can be modified to return specific payloads (e.g., JSON responses) for those matched requests.

During the initialization of WireMock servers, it was challenging to manage TCP ports since the SUT may connect to various external web services using the same TCP port (e.g., HTTP 80 or HTTPS 443). Conversely, during the search, we have control over the SUT through instrumentation. On the other hand, it was challenging to maintain the same behavior in the generated test cases (e.g., JUnit files) where no bytecode instrumentation is applied. To ensure the same behavior in the generated test cases as the search, we rely on Java Reflection. Especially when hostnames are hard-coded in third-party libraries and cannot be easily modified by the user (e.g., the case of `https://api.github.com` in *catwatch*).

We managed to modify the JVM Domain Name System (DNS) cache through reflection in the generated test cases by using the *dns-cache-manipulator* from Alibaba.¹⁹ This way, in the generated JUnit tests, we can still remap a hostname such as `api.github.com` to a loopback address where a WireMock instance is running. However, one (arguably minor) limitation here is that we cannot handle in this way the cases in which IP addresses are hard-coded without using any hostname (e.g., using `140.82.121.6` instead of `api.github.com`). At any rate, the use of hard-coded IP addresses does not seem to be a common practice.

Additionally, different operating systems (OS) handle TCP ports in different ways, especially regarding available ephemeral TCP ports and the *TCP Time Wait* delays. *TCP Time Wait* defines the amount of time it will take to release a TCP port once it is freed from the previous process. Each operating system has a different default value for the *TCP Time Wait* delay. This is a major issue for empirical research, especially when running several experiments on the same machine in parallel.

¹⁹<https://github.com/alibaba/java-dns-cache-manipulator>

Using different addresses from the loopback address range (e.g., 127.0.0.0/8) is an easy and elegant solution to overcome the challenges we faced due to the limitation of available ephemeral TCP ports to initiate WireMock servers. This has well over 16 million addresses, which prevents clash issues in all of our experiments. For example, a WireMock server for `example.test:8123` could be started on 127.1.2.3:8123. However, some operating systems might handle loopback addresses differently. For example, *macOS* does not enable all possible loopback addresses within the 127.0.0.0/8 range by default. So, other addresses from the range should be configured as an alias on the respective network interface (in most cases, it is *lo0*). By contrast, we have not seen this kind of problem when running experiments on *Linux* and *Windows*. Additionally, it eradicated the dependency on *TCP Time Wait* delays. Furthermore, this allowed us to create and manage mock servers with no code modification required from the SUT.

The decision to pick an IP address to bind mock servers is done in an automated manner. In our extension to EvoMASTER, we developed three distinct configurable strategies for address selection. This helped to avoid conflicts while running experiments in parallel and to have more control over address allocation (e.g., in the generated test cases).

In order to have the flexibility in IP address selection during the experiments, we have developed four distinct configurations related to external web service handling inside EvoMASTER. Selecting *NONE* will disable the external web service handling. This allowed us to disable the external service handling during the experiments. Under each setting, some of the loopback addresses will be skipped for various reasons. This includes addresses 127.0.0.0, 127.0.0.1, 127.0.0.2 (i.e., which is used as DNS sinkhole) and 127.0.0.3 which is kept as a placeholder for future purposes. For instance, 127.0.0.1 is skipped due to the limited number of ephemeral ports available, because it is used for various purposes in the operating system as well as in EvoMASTER. Also, we skip the broadcast address of the range 127.255.255.255, to avoid unanticipated side effects. The *DEFAULT* option will enable the external web service handling and it uses IP addresses starting from 127.0.0.4. The IP address for the second external web service will be the next in the range (i.e., 127.0.0.5). Like the name, option *RANDOM* selects a random IP address from the range while excluding the reserved IP addresses. The option *USER* allows the user to specify the starting IP address.

4.4 Genotype Representation

As previously mentioned, the instantiated WireMocks will run with a default response of HTTP 404 for all the request patterns. However, mocking the external web services with different correct responses is necessary to improve the code coverage. Recall the example in Figure 5, where the result of an *if* statement depends on whether the external service returns the string "HELLO THERE!!!". To reach this point of execution, the mock server should give the response as expected. It would be extremely unlikely to get the right data (i.e., relevant response schema) at random. Performance improvements of the search requires better heuristics in a case like this.

Besides body payloads, the SUT can check other HTTP response parameters as well (e.g., HTTP status code and headers). For instance, the SUT could execute a different code block when the status value is 429 (which denotes too many requests sent within an amount of time). To maximize the code coverage of the SUT (and so indirectly increase the chances of detecting faults), there is the need to have various HTTP responses in each stage of the search to cover all these possible cases.

All of these factors add to the complexity. To tackle these issues, we took a divide-and-conquer approach. All the requests made to external web services are handled individually. Each one of them is represented by an executable action (e.g., `ApiExternalServiceAction`) during the search. Meanwhile, WireMock supports only HTTP(S) based mocking. Consequently, we only tackle HTTP(S) based requests. A REST call may have other functions inside the code, such as calling

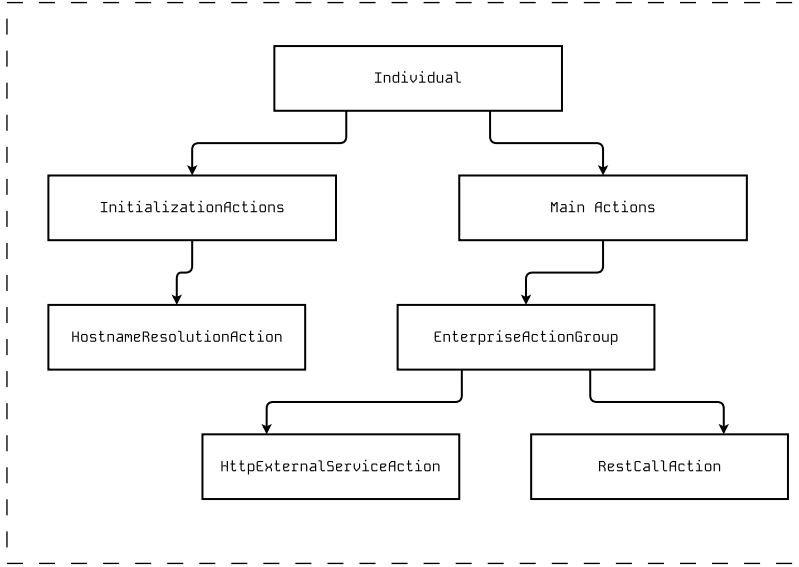


Fig. 10. The depiction of how actions are divided into groups within each individual.

an external web service, publishing to a message queue, interacting with the database, or reading and writing files. In this paper, we have solely addressed the specifics of how our HTTP(S)-based requests are handled in our methodology.

As the first step, we need to extend the internal fuzzing engine of EvoMASTER to deal with the creation of mocked responses. In the evolutionary process of EvoMASTER, an evolved *individual* (i.e., a test case) is composed of one or more *actions* (e.g., HTTP calls and data setups in SQL databases). Each action is then composed of a set of *genes*, representing the different parameters defining the action (e.g., string values for URL query parameters and JSON objects for body payloads). Mutator operators of the evolutionary process can alter the structure of an individual by adding or removing actions in it (e.g., adding a new HTTP call in a test case), or modifying the genotype of the actions (e.g., mutating the string value of a query parameter).

Each of the interactions with external services inside the SUT will have a corresponding action inside EvoMASTER, to represent how to set the response from the contacted WireMock server. As shown in Figure 10, each of the interactions will be represented by a child of `ApiExternalServiceAction`, and depending on the protocol, it will have its child action type. In this case, it will be represented using `HttpExternalServiceAction`. All the sub-actions are grouped into an `EnterpriseActionGroup` which will be a child element to the corresponding parent of a `RestCallAction` which defines the API call. `EnterpriseActionGroup` includes all the requests made to the mock server and their respective responses as `HttpExternalServiceAction`. During mutation, if a `RestCallAction` is deleted, all the relevant `HttpExternalServiceAction` inside the `EnterpriseActionGroup` will be deleted along with that.

During the search, once a test is executed, and its fitness evaluation is computed, through the instrumentation we gather information about the interactions the SUT had with the external services from the respective WireMock servers. As shown in Figure 11, after `getAllServedRequests()`, the collected heuristics from the mock server, about the served HTTP requests will be used to create `HttpExternalServiceAction` inside the `EnterpriseActionGroup` for each unique request.

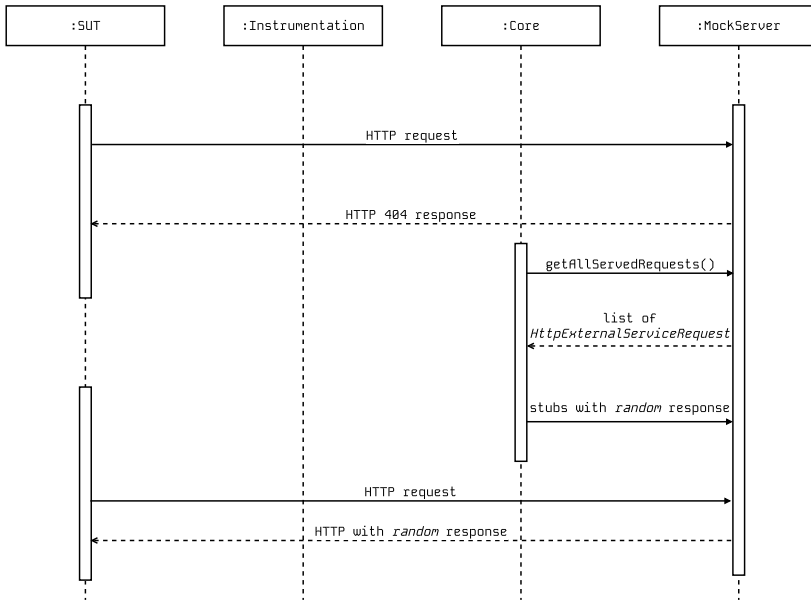


Fig. 11. The sequence diagram which illustrates how a mock server is set up.

```

1 val text = BufferedReader(connection.getInputStream().reader()).readText()
2 val tree = gson.fromJson(text, Map::class.java)

```

Fig. 12. REST endpoint code block is taken from our artificial case study to illustrate taint analysis flow.

Each HTTP request will be identified as unique by the URL path when creating `HttpExternalServiceAction` and duplicates will be omitted. For example, for `/api/v1/` there will be only one `HttpExternalServiceAction` representing the request, regardless of how many requests are made to the external web service.

At the point of initiation, an instance of `HttpExternalServiceAction` will have information about the request made to the external web service such as hostname, protocol, URL path, headers, and body. In the following steps of the search, when the test is selected again for mutation, these newly created actions will be mutated. Therefore, the respective WireMock server will have a new response for the request pattern besides the default HTTP 404. Initially, the genotype of the `HttpExternalServiceAction` will contain mutable genes to handle the returned HTTP status code and an optional body payload (for example, treated as random strings). As a result, a new mock server response will be made using randomly picked predefined values of HTTP status in the `HttpWsResponseParam`. The response body will be decided based on the selected HTTP status code. For HTTP status codes such as all 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include content. For other HTTP status codes, the response body will be an empty string at the initiation. Similarly, throughout the search, actions will enhance the corresponding WireMock stubs by mutating those genes and then collecting their impact on the fitness function.

4.5 Schema Inference

It is possible to easily randomize some parameters (e.g., HTTP status codes) in a typical HTTP response during the search. However, creating correct values in parameters like headers and response bodies is not a straightforward task. With no information available at the beginning of the search, it is necessary to find a way to gather this information to create a response schema. For example, when the SUT expects a specific JSON schema as a response, sending a random string as the body payload will lead to throwing an exception in the data parsing library of the SUT.

On the other hand, the information about the external web service may be available in a commonly used documentation format such as OpenAPI/Swagger. In theory such information could be used to create syntactically valid responses. However, not only such formal schemas could be missing, but also automatically locating them might not always be possible unless the user provides them manually. As an alternative, more general solution would be to use bytecode instrumentation to analyze how such body payloads are parsed by the SUT [15]. Furthermore, this approach allows us to extract the essential schema information needed to generate the mock web services.

The JSON is a common choice as a data transfer format for the interactions between web services [41]. Usage of Data Transfer Objects (DTO) is a known practice in most of the statically typed programming languages (e.g., Java, C, C++, Rust) to represent the schema in code. When a JSON payload is received from an HTTP request, a library can be used to parse such text data into a DTO object. By instrumenting the relevant libraries at runtime during the test evaluation, we can analyze these DTOs to infer the schema structure of these JSON messages.

On the JVM, the most popular libraries for JSON parsing are *Gson* and *Jackson* [33]. For these two libraries, we provide method replacements for their main entry points related to parsing DTOs, for example, `<T> T fromJson(String json, Class<T> classOfT)`. In these method replacements, we can see how any JSON string data is parsed to DTOs (analyzing as well as all the different Java annotations in these libraries used to customize the parsing, for example, `@Ignored` and `@JsonProperty`). This information can then be fed back to the search engine: instead of evolving random strings, *EvOMASTER* can then evolve JSON objects matching those DTO structures.

There is one further challenge that needs to be addressed here: how to trace a specific JSON text input to the source it comes from. It can come from an external web service we are mocking, but that could use different DTO for each of its different endpoints. Alternatively, the data may come from a database or may be provided as input to the SUT. In those cases, the data has nothing to do with *WireMock* instances. Therefore, we need to distinguish each case.

In order to tackle such a challenge, we use *taint analysis*. Figure 13 explains the flow of how the taint analysis is implemented to achieve this goal. In particular, we extend the current input tracking system in *EvOMASTER* [12]. Each time a string input is used as body payload in the mocked responses, it will not be a random string but rather a specific tainted value matching the regex `_EM_d+_XYZ_`, e.g., `_EM_0_XYZ_`.

Figure 12 contains two lines of code obtained from one of our artificial case studies to provide a practical example of taint analysis. In Line 2, the SUT reads the response from the external service call made using *BufferedReader*. Afterwards, using `fromJson` from *Gson* library, the response JSON string is converted into a *Map* of strings to process further. In this case, during the input generation, by including a tainted value (e.g., `_EM_0_XYZ_`) in the JSON string it is plausible to trace the process of input handling throughout the execution. Such a mechanism allows us to trace and identify the DTO used. In the next step, the identified DTO can be used further as the response schema in the mock server for the request.

Similar to the provided example in Figure 12, in each of our method replacements (e.g., for `fromJson`), we check if the input string matches that regular expression. If so, we can check the genotype of the executed test case to see where that value is coming from. The gene representing

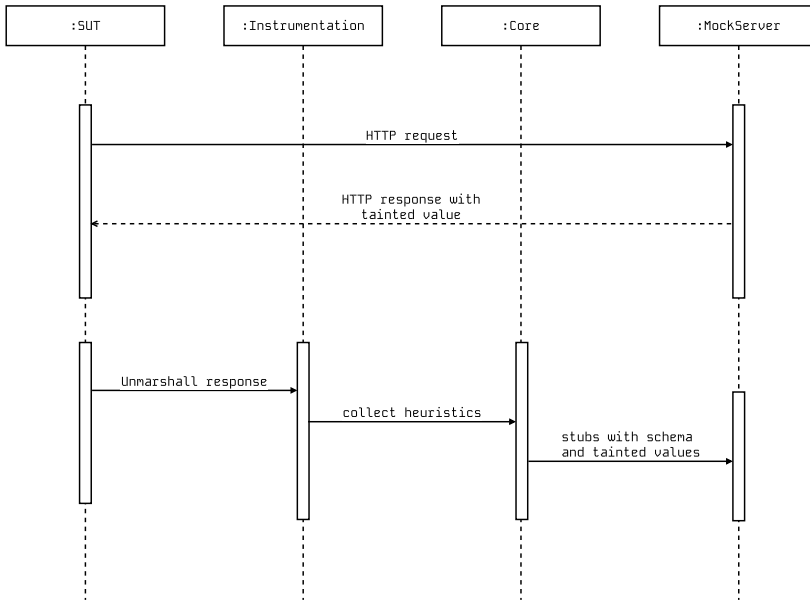


Fig. 13. The sequence diagram demonstrating the flow of taint analysis.

that value is then modified in the next mutation operation to rather represent a JSON object valid for that DTO.

During the search, modifications/mutations to an evolving test case might lead the SUT to connect to new web services, or not connect anymore to any (e.g., if a mutation leads the test case to fail input validation, and then the SUT directly returns a 400 status code without executing any business logic). During the fitness evaluation all the mock servers will be reset to the default state before computing the fitness (i.e., with the HTTP 404 for all request patterns). Thereafter, the active state of the action will be set based on the “used” parameter for all existing actions. Active actions will rebuild the WireMock stubs based on the active index for each HTTP parameter. After the test execution, if no matching request exists to the existing actions after a fitness evaluation, it will be marked as not used (i.e., “used” will be set to false). The rest will be marked as used for further use.

4.6 Harvesting Responses

Inferring the expected structure of the JSON responses can improve the search performance (as we will show in Section 6). However, it might not always be possible to infer these schemas. Furthermore, even when possible, for the search to generate a useful response to mock the external web service, it might take a significant amount of time.

To expedite this process, we have implemented an approach to harvest the actual responses from the original service in parallel to the search, if it is available. This means that, when the SUT makes a call to a WireMock server, from our EvoMASTER extension, we still do the same call as well towards the real external web service. We created a multithreaded harvester to make these requests and fetch the responses with all their relevant information, such as headers. Then, each time a test case is mutated, it will refer to the harvester for available actual response payloads to set up the WireMock stubs, as illustrated in Figure 14. With a given probability, one of these responses

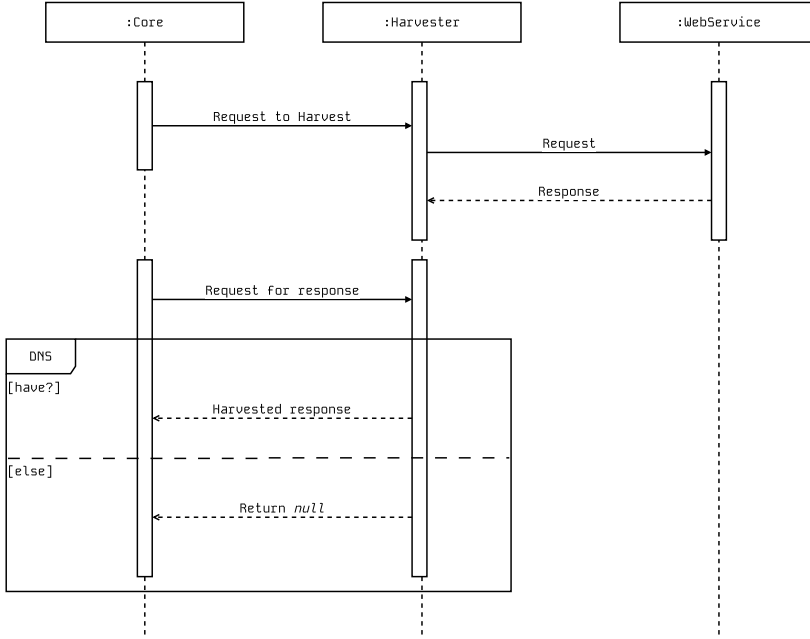


Fig. 14. Sequence diagram explaining the execution flow of harvesting responses from the actual external web services.

(for the matching called endpoint) will replace the current genotype of the WireMock action setup. Then, these payloads can still be further mutated and evolved during the search.

Anyhow, accessing the actual service is not always possible (e.g., for the *cwa-verification* case the service was not publicly available) and accessible. Sporadically, we faced issues with the availability of the service, and sometimes rate-limited access. Apart from this, occasional response timeouts and network performance degradation caused issues. This had an impact on the search performance.

To overcome these hurdles, we developed three strategies to select the harvested actual responses, if available. The first option selects the exact matching response to the request URL, the second option selects the closest match to the request URL, and the third option selects a random response from the harvested requests. The selection is a continuing process until the search finds a satisfying response to the request. We evaluated each option's performance separately. In some preliminary experiments (not reported in this paper), we did not see much differences among these strategies, so we left on the default random strategy.

5 Generated Test Cases

During the search, we were able to control the external web service connections using instrumentation. Instrumentation helped to manipulate the URL information to connect the HTTP mock servers instead of the external web services. Unfortunately, replicating the same behavior within the generated JUnit test cases has become another challenge. To do the same inside the generated test cases, we have to rely on the DNS cache of the JVM.

The JVM uses DNS caching to improve the network performance. Each first network access to a remote location made by the application will cache the DNS information for further use. So, any subsequent operation will read from the previously cached information. The cached DNS

```

1 @Test @Timeout(60)
2 fun test_7() {
3     DnsCacheManipulator.setDnsCache("example.test", "127.0.0.92")
4     assertNotNull(wireMock__http__hello_there__8123)
5     wireMock__http__hello_there__8123.stubFor(
6         get(urlEqualTo("/api/string"))
7         .atPriority(1)
8         .willReturn(
9             aResponse()
10                .withHeader("Connection","close")
11                .withHeader("Content-Type","application/json")
12                .withStatus(201)
13                .withBody("\nHELLO THERE!!!\n")
14            )
15     )
16
17     given().accept("*/*")
18         .header("x-EMextraHeader123", "")
19         .get("${baseUrlOfSut}/api/wm/urlopen/string")
20         .then()
21         .statusCode(200)
22         .assertThat()
23         .contentType("text/plain")
24         .body(containsString("YES"))
25
26
27     wireMock__http__hello_there__8123.resetAll()
28 }

```

Fig. 15. The code block taken from a generated test specified with JUnit and Kotlin for the synthetic example shown in Figure 5.

information inside the JVM can be programmatically altered if it is necessary to reroute the traffic to a different destination for the same hostname.

In our case, before each call to external web services, we are able to manipulate the DNS cache information using a library called *DNS cache manipulator*. *DNS cache manipulator* helped to reduce the amount of work that needs to be done to achieve the same results [19].

At the end of the search, EvoMASTER produces self-contained test suites as JUnit test files, written in either Java or Kotlin. An individual test suite will contain all the necessary configurations related to WireMock, and this allows the test to run independently without any additional configurations. Figure 15 has a single test taken from the generated test suite for the synthetic example in Figure 5. After the function definition, `DnsCacheManipulator.setDnsCache()` configures the JVM DNS cache to redirect all the DNS queries to a local address where the WireMock server is running. With the successful assertion on the presence of WireMock, the test further sets the stub with all the necessary parameters for the request pattern for a successful response. This stub represents the response from the external web service with which the SUT is supposed to communicate. Further down, the test checks for the successful response from the SUT. This is only achievable when the external web service responds with a "HELLO THERE!!!" text, as shown in Figure 5.

Table 1. Experiment settings

RQ	Internet	Techniques	SUTs
RQ1	On	<i>Mocking</i>	Artificial
RQ2	On	<i>Base, Mocking</i>	Real-World
RQ3	Off	<i>Base, Mocking</i>	Real-World

Table 2. Descriptive statistics of the employed artificial SUTs. For each SUT, #Endpoints represents the number of declared HTTP endpoints in those APIs. #Path represents the source code path in the E2E test folder of EvoMASTER.⁹

SUT	#Endpoints	#Path
<i>Auth0</i>	1	com/foo/rest/examples/spring/openapi/v3/wiremock/auth0
<i>Inet</i>	1	com/foo/rest/examples/spring/openapi/v3/wiremock/hostnameaction
<i>Socket</i>	3	com/foo/rest/examples/spring/openapi/v3/wiremock/socket
<i>URL</i>	3	com/foo/rest/examples/spring/openapi/v3/wiremock/urlopen
<i>OkHttp</i>	3	com/foo/rest/examples/spring/openapi/v3/wiremock/okhttp
<i>OkHttp3</i>	3	com/foo/rest/examples/spring/openapi/v3/wiremock/okhttp3
<i>Harvester</i>	4	com/foo/rest/examples/spring/openapi/v3/wiremock/harvestresponse

6 Empirical Study

To evaluate the effectiveness of our novel techniques presented in this paper, we carried out an empirical study to answer the following research questions:

RQ1: Is our novel approach capable of automatically generating mock external web services for artificial, hand-crafted examples?

RQ2: Compared to the performance of existing white-box fuzzing techniques, how much improvement can our mock external web service generation achieve in terms of code coverage and fault detection on real-world applications?

RQ3: When external services are inaccessible, how effective is our approach in generating mock external web services?

6.1 Experiment Setup

To answer our three research questions, we carried out three different sets of experiments, one per research question, on two different sets of SUTs, as shown in Table 1. We consider two different variables: (1) the *internet* (either *On*, or *Off*) represents whether the internet connection is enabled or not while running the experiments; (2) then we conducted experiments to explore that the *technique* represents the baseline and our proposed approach, i.e., *Base* refers to EvoMASTER with its default configuration, and *Mocking* refers to our approach, which enables our handling of mock external web service generation with EvoMASTER.

6.1.1 Artificial APIs. To answer **RQ1**, the first set of SUTs consists of seven artificial APIs, developed by us, as listed in Table 2. This includes REST APIs covering different libraries and various real-world scenarios. The list includes third-party libraries such as Auth0²⁰ and OkHttp, as well as JDK classes such as InetAddress, Socket, and URL. Additionally, this set contains as well an API to cover the *Harvester* (i.e., covering the technique presented in Section 4.6) using a few real-world

²⁰<https://auth0.com/>


```

1 @GetMapping(path = [""])
2 fun get() : ResponseEntity<String> {
3
4     try {
5         val domain = "www.doesnotexistfoo.test:6789"
6         val audience = String.format("https://%s/api/v2/", domain)
7         val authClient = AuthAPI(domain, "foo", "123")
8
9         val tokenHolder = authClient.requestToken(audience).execute()
10        return ResponseEntity.ok("OK")
11    } catch (e: Exception){
12        return ResponseEntity.status(400).build()
13    }
14 }

```

Fig. 16. REST endpoint code block which is taken from our artificial API *Auth0* using the Auth0 SDK.

APIs as external web services. Besides, we use a popular JSON parser, namely *Jackson*,⁷ for JSON handling inside the artificial APIs. The artificial examples were developed using SpringBoot¹⁷ with Kotlin.

Fuzzing a REST API involves addressing several different challenges [54]. Our novel techniques presented in this paper are aimed at one specific challenge, which is dealing with interactions with external services. To obtain sound empirical evidence, empirical evaluations should be based on real-world applications. However, lacks of performance improvements on some APIs might be independent of the quality and effectiveness of a novel technique, and rather they could be related to other challenges that are not fully solved yet. For example, no communications with external services would be done if a fuzzer is not able to generate test cases that can bypass the first layer of input validation, and so the SUT would return immediately with a 4xx user-error status code. Such an API would not be good for studying the effectiveness of mocking external services, as no call to external services would be done.

To address these potential issues, we have designed artificial examples in which dealing with external services is the main (and possibly only) challenge. This enables us to study the effectiveness of our novel techniques without any confounding factors. Furthermore, this also enables us to have a larger case study, as finding more real-world APIs for experimentation in this context is challenging. Still, experiments only on artificial examples are not sufficient. Real-world applications must be used as well.

The full source code of these artificial examples is available in the code repository of EvOMASTER, as they are currently used as end-to-end (E2E) tests for it [14]. Table 2 provides the package paths for each example. In the reminder of this section, for sake of clarity, we also provide code snippets for some of the endpoints of these artificial APIs.

Auth0. Figure 16 contains the REST endpoint written in SpringBoot.¹⁷ As shown, we initiate the Auth0 SDK with a fake hostname and port. This library will try to connect to that remote server, assuming it is an Auth0 server, where it will try to authenticate the user called *foo*, with password 123. If the connection is successful through the SDK, our REST API will respond with an HTTP 200 status code, and with the body "OK". Otherwise, it will answer with a user error HTTP 400 status code.

Inet. Figure 17 contains the code of the REST endpoint from the artificial case study developed to test the usage of `java.net.InetAddress`. First, the code initiates an `InetAddress` with a fake

```

1 @GetMapping(path = ["/resolve"])
2 fun exp(): ResponseEntity<String> {
3     val address = InetAddress.getByName("imaginary-second.local")
4
5     return try {
6         val socket = Socket()
7
8         val a = InetSocketAddress(address, 10000)
9         socket.connect(a, 10000)
10        socket.close()
11
12        ResponseEntity.ok("OK")
13    } catch (e: Exception) {
14        ResponseEntity.status(400).build()
15    }
16 }

```

Fig. 17. REST endpoint code block which is taken from our artificial API *Inet* for `java.net.InetAddress`.

```

1 @GetMapping(path = ["/resolve"])
2 fun exp(): ResponseEntity<String> {
3     val a = InetSocketAddress("imaginary-socket-evomaster.com", 12345)
4
5     return try {
6
7         val socket = Socket()
8
9         socket.connect(a)
10        socket.close()
11
12        ResponseEntity.ok("OK")
13    } catch (e: Exception) {
14        ResponseEntity.status(400).build()
15    }
16 }

```

Fig. 18. REST endpoint code block taken from our artificial case study for `java.net.Socket`.

hostname and uses it to create an `InetSocketAddress` (Lines 3-- 8). After that, the SUT tries to connect to the remote host using a `Socket` (Line 9). Once the connection is established successfully, the code will respond with HTTP 200 and body "OK", otherwise with HTTP 400. Although this implementation uses `java.net.Socket`, as shown in Figure 18, we have created separate APIs to cover scenarios related to the `Socket` class.

Socket. Similar to the *Inet* API, in the *Socket* API the endpoint tries to resolve a fake hostname and port using `java.net.InetSocketAddress` instead of a `java.net.InetAddress`, and then establishes a connection, as shown in Figure 18. On success, the endpoint will respond with HTTP 200 and body "OK" (Line 12), otherwise with HTTP 400 (Line 14).

URL. Unlike *Inet* and *Socket*, for `java.net.URL` we have three different cases. The code block in Figure 19 is taken from the API developed to study `java.net.URL`. Each case uses `java.net.URL`

```

1 ...
2 val url = URL("http://example.test:8123/api/string")
3 val connection = url.openConnection()
4 connection.setRequestProperty("accept", "application/json")
5 ...

```

Fig. 19. A code block taken from the API *URL* developed for `java.net.URL`.

```

1 ...
2 val url = URL("$protocol://$host:5555/api/string")
3 val request = Request.Builder().url(url).build()
4
5 try {
6     val data = client.newCall(request).execute()
7     val body = data.body()?.string()
8     val code = data.code()
9     return if (code in 200..299){
10         if (body == "\"HELLO THERE!!!\""){
11             ResponseEntity.ok("Hello There")
12         } else {
13             ResponseEntity.ok("OK")
14         }
15     } else if (code in 300..499){
16         ResponseEntity.status(400).build()
17     } else {
18         ResponseEntity.status(418).build()
19     }
20 } catch (e: Exception){
21     return ResponseEntity.status(500).build()
22 }
23 ...

```

Fig. 20. A code block from the API *OkHttp*.

to establish a connection to a fake external web service (Lines 2-- 3). We developed two identical endpoints to test HTTP and HTTPS. Additionally, we created one more endpoint to use `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson*.

Under each endpoint, once the connection to an external web service is successful, each uses a different means to unmarshal the received response. In two of the endpoints, we used the *URL* built-in buffer reader, while in one we used `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson* to unmarshal the response. In the end, in two of the endpoints, we return "OK" with HTTP 200 if the external service responds with "HELLO THERE!!!", otherwise with HTTP 500. Meanwhile, in one that uses the *ObjectMapper*, we return "OK" with HTTP 200 if the integer parameter (i.e., *x*) in the response is greater than 0.

OkHttp and *OkHttp3*. Similar to *URL*, we have three different endpoints for `com.squareup.okhttp` and `okhttp3`. The code snippet in Figure 20 is taken from the API created to test the `com.squareup.okhttp`. First, `com.squareup.okhttp.Request` is used to establish a connection to a remote host using different ports and protocols, such as HTTP and HTTPS (Lines 2 and 3). In this example, HTTP protocol is used for connecting to `github.com` in port 5555. Likewise, we have developed a

Table 3. Descriptive statistics of the employed SUTs. For each SUT, #SourceFiles represents the number of files (i.e., the number of public Java classes) composing such SUT, where #LOCs represents their total number of lines of code (LOC). Finally, #Endpoints represents the number of declared HTTP endpoints in those APIs.

SUT	#SourceFiles	#LOCs	#Endpoints
<i>catwatch</i>	106	9636	14
<i>cwa-verification</i>	47	3955	5
<i>genome-nexus</i>	405	30004	23
<i>ind1</i>	163	15240	53
<i>pay-publicapi</i>	377	34576	10
Total 5	1098	93411	105

separate method to use HTTPS as well. Furthermore, similar to the *URL*, we have one more method to test the use of `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson*.

Harvester. To test the our Harvester technique (recall Section 4.6), we created a dedicated API with four endpoints to cover different scenarios. Compared with previous examples, in *Harvester* instead of using fake hosts, we used one public API used in *genome-nexus* and an API with a JSON schema list created in *Listly*²¹ with synthetic data. Furthermore, in the *Harvester* API, we used all the libraries from previous examples to make connections and to read responses (i.e., *URL*, *Jackson*, and *OkHttp*).

6.1.2 Real-World APIs. To answer **RQ2** and **RQ3**, we decided to evaluate our approach using real-world applications. Therefore, our second sets of SUTs consist of four APIs from EMB [16] and one industrial API (i.e., *ind1*). EMB is an open-source corpus composed of open-source web/enterprise APIs for scientific research [16]. This corpus has been utilized by different research groups for performing experiments related to Web API fuzzing techniques [30, 47, 50, 57].

To assess the effectiveness of our mock generation, from EMB, we selected all the REST APIs which require communicating with external web services for their business logic (i.e., *catwatch*, *cwa-verification*, *genome-nexus*, and *pay-publicapi*). Descriptive statistics of the selected APIs are reported in Table 3. In total, 105 endpoints and 93k lines of codes were employed in these experiments. Note that these statistics only concern code for implementing the business logic of these APIs. The code related to third-party libraries (which can be millions of lines [16]) is not counted here.

Compared to other studies in the literature on fuzzing REST APIs, using only four open-source projects can be considered limited. For example, in the study introducing ARAT-RL [29] 10 open-source APIs were employed, and 11 were employed in the study introducing DeepREST [18]. In our own previous work on comparing REST API fuzzers, we used 18 open-source APIs [54]. As our novel techniques are white-box based, we are limited in our selection of APIs written in the programming languages we can support, i.e., Java and Kotlin. Furthermore, although intra-API communications are common in industrial APIs, especially when large systems are developed with a microservice architecture, they are less so in open-source projects. We are aware of no other Java/Kotlin API used in previous research studies in the literature that we could have added for our empirical study in this paper. This is a major reason why we made sure to employ in our study at least one closed-source industrial API, and developed seven artificial APIs.

²¹<https://list.ly>

catwatch is a web application that retrieves statistics from GitHub for user accounts, processes and stores this data in a database, and subsequently provides access to the data through a REST API. The application heavily relies on GitHub APIs to perform its core functions.

cwa-verification is a component of the official Corona-Warn-App for Germany. This API is part of a microservice-based architecture that contains various other services, including mobile and web apps. The tested backend server relies on other services in the microservice architecture to complete its tasks.

genome-nexus is an interpretation tool that allows gathering information about genetic variants in cancer. This API gathers and integrates information from various external online sources. These include web services that convert DNA changes to protein changes, predict the functional effects of protein mutations, and contain information about mutation frequencies, gene function, variant effects, and clinical actionability.

pay-publicapi is an open-source API from the government of the United Kingdom for government or public sector organizations that want to take payments. The API provides the ability to take payments, issue refunds, and run reports on all payments. This API is currently being used by partners from across the public sector, including the NHS, MOJ, police forces, and local authorities.

ind1 is closed-source API, part of an e-commerce system. It deals with authentication using an external Auth0 server, and it processes monetary transactions using Stripe.²

Interactions with real external web services are often non-deterministic, as the external web services might not be accessible and change their behavior at any time. This is a major issue for software testing, as any generated test could become flaky [44]. In this study, to handle this issue, we studied the performances of the compared techniques with two internet settings (i.e., *On* and *Off*). The interactions are deterministic with the disabled internet setting (i.e., *Off*), while, with an enabled setting (i.e., *On*), results might be negatively affected by the variance in the external web services. For instance, with a preliminary study, we found that *catwatch* communicates with GitHub API, but there exists a rate limiter (a typical method to prevent denial-of-service attacks) to control the access to this API²² based on IP address, e.g., up to 60 requests per hour for unauthenticated requests. As the network setup of our university, we may share the same IP address with all of our colleagues and students (e.g., when behind a NAT router). Due to this, such a rate limit configuration will strongly impact the results of the experiments. For example, depending on the time of the day in which the experiments are running, in some experiments we might be able to fetch data from GitHub, but not in others. As this can lead to completely unreliable results when comparing techniques, we decided to run experiments without internet connection as well. Therefore, to answer **RQ3**, we repeated the same experiments as **RQ2**, but with the machine physically and wirelessly disconnected from the internet. This also provides a way to evaluate our techniques in cases in which the external web services are not implemented yet (e.g., at the beginning of a new project) or are temporarily down.

Considering the random nature of search algorithms, we repeated each setting 30 times using the same termination criterion (i.e., 1 hour), by following common guidelines for assessing randomized techniques in software engineering research [6]. With two settings (i.e., *Base* and *Mocking*) on five SUTs using 1 hour as search budget, 30 repetitions of the experiments took 300 hours (12.5 days), i.e., $2 \times 5 \times 30 \times 1$, for **RQ2**, and another 300 hours for **RQ3**, for a total of 25 days.

All the experiments were run on the same machine, i.e., an HP Z6 G4 Workstation with Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz, 192 GB RAM, and 64-bit Windows 10 OS.

To take the randomness of these algorithms into account when analyzing their results, we used common statistical tests recommended in the literature [6]. In particular, we employed a

²²<https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28>

Mann–Whitney U test (i.e., p -value) and Vargha-Delaney effect size (i.e., \hat{A}_{12}) to perform comparison between *Base* and *Mocking* in terms of line coverage and fault detection. With the Mann–Whitney U test, if p -value is less than a significant level (i.e., 0.05), it indicates that the two compared groups (i.e., *Base* and *Mocking*) have statistically significant differences. Otherwise, there is not enough evidence to support the claim the difference is significant. Comparing *Mocking* with *Base*, the Vargha-Delaney effect size (i.e., \hat{A}_{12}) measures how likely *Mocking* can perform better than *Base*. $\hat{A}_{12} = 0.5$ represents no effect. If \hat{A}_{12} surpasses 0.5, it indicates that *Mocking* has more chances to achieve better results than *Base*.

6.2 Results for RQ1

In this section, we discuss the details of the experiments to be able to answer **RQ1**. Due to space constraints, we only discuss the experiment steps for *Auth0* in detail. For the other APIs, the experiment steps were identical.

Each artificial API is designed in a way in which it makes a call to an external service (that does not exist), fetches some data, parses it, and then returns a success response if no error was encountered. If our novel techniques are successful, then it should be possible to get a success response on each of these artificial APIs. We do not need to empirically compare with *Base* technique here, as by construction all test executions will crash when trying to connect to the non-existent external services. As such, we already know it would be impossible to get any success response with *Base*.

In a preliminary study, we successfully generated self-contained test suites with mock external web services for all the artificial APIs presented in Table 2. Furthermore, we were able to handle multiple mock external web services at once without any conflicts, as discussed in Section 4.3. All these artificial APIs can now be fully covered by EvoMASTER, using a small search budget, reliably. Thanks to this, these artificial APIs have been added as part of the E2E tests for EvoMASTER [14]. This means that, at each build of EvoMASTER on its Continuous Integration server (currently GitHub Actions), EvoMASTER is run on these APIs, and the build fails if any API cannot be covered. To achieve this, we wrote embedded drivers and E2E tests for each of these APIs.

In the following steps, we discuss the details of the tests and the outcomes in detail only for *Auth0*, as an example. The other APIs would have similar discussions.

As shown in Figure 21, we wrote an EvoMASTER E2E test to evaluate our artificial API with *Auth0* SDK. This test uses the E2E scaffolding of EvoMASTER to run the API, run EvoMASTER on it (with a search budget of 500 fitness evaluations in this case), verify the defined assertions (e.g., Line 20), compile the generated JUnit test cases, and run those later as well. If the E2E test takes more than 3 minutes (specified on Line 22), it will fail. If the test is flaky, it is repeated up to N times with different incremental seeds. The full details of this E2E test infrastructure can be found in [14].

In this E2E test, we evaluate whether we are able to reach the line in the SUT which gives the response of HTTP 200 and with the body "OK". To be able to reach the line of code that returns the HTTP 200 response, the implementation of our approach of automatically generating external web services must work successfully. Novel, experimental features are not on by default, until they are stable and properly analyzed. In this case, our novel techniques need to be activated via commandline properties, as set in this E2E test on Line 14. If we manage to reach the line of code within the given budget for search, the test will pass (i.e., as the assertion on Line 20 will be satisfied). Finally, the test will generate a self-contained JUnit test suite after completion. Below is a detailed analysis of these generated tests.

```

1 @Test
2 fun testRunEM() {
3     runTestHandlingFlakyAndCompilation(
4         "WmAuth0EM",
5         "org.foo.WmAuth0EM",
6         500,
7         true,
8         { args: MutableList<String> ->
9
10            args.add("--externalServiceIPSelectionStrategy")
11            args.add("USER")
12            args.add("--externalServiceIP")
13            args.add("127.0.0.22")
14            args.add("--instrumentMR_NET")
15            args.add("true")
16
17            val solution = initAndRun(args)
18
19            assertTrue(solution.individuals.size >= 1)
20            assertHasAtLeastOne(solution, HttpVerb.GET, 200, "/api/wm/auth0", "OK")
21        },
22        3
23    )
24 }

```

Fig. 21. E2E test case (in the class named WmAuth0EMTest) of EvoMASTER for the *Auth0* API.

```

1 private val controller : SutHandler = com.foo.rest.examples.spring.openapi.v3.
    wiremock.auth0.WmAuth0Controller()
2 private lateinit var baseUrlOfSut: String
3 private lateinit var wireMock__https__www_doesnotexistfoo_test__6789:
    WireMockServer

```

Fig. 22. Variable declarations from the generated suite for the SUT (i.e., *Auth0*) shown in Figure 16.

As seen in Figure 22, the beginning of a test suite contains the necessary variables to configure the mock servers and other parameters needed throughout the test. Of particular importance is the instantiation of the embedded driver (class WmAuth0Controller on Line 1). Figure 23 shows the BeforeAll section of the generated test suite. Under the BeforeAll section, the test suite will use the driver to initiate the SUT (Line 5). After the successful start of the SUT, the test suite will set up the mock server(s) and start them (Line 18). Each test is self-contained in the sense that the necessary configuration to run the test successfully will be configured under the test, and everything will be reset to default at the end.

Figure 24 contains the BeforeEach section of the generated test suite. Before executing each test, we ensure that the mock server is always set to the default state, similar to the behavior during the search, as discussed earlier in Section 4.3. Additionally, we reset the DNS cache to the default configuration using DnsCacheManipulator.

Figure 25 contains a unit test taken from the generated test suite for the artificial API (i.e., *Auth0* in Figure 16). The generated test suite contains all the necessary elements to execute the tests

```

1 @BeforeAll
2 @JvmStatic
3 fun initClass() {
4     controller.setupForGeneratedTest()
5     baseUrlOfSut = controller.startSut()
6     controller.registerOrExecuteInitSqlCommandsIfNeeded()
7     assertNotNull(baseUrlOfSut)
8     RestAssured.enableLoggingOfRequestAndResponseIfValidationFails()
9     RestAssured.useRelaxedHTTPSValidation()
10    RestAssured.urlEncodingEnabled = false
11    RestAssured.config = RestAssured.config()
12        .jsonConfig(JsonConfig().numberReturnType(JsonPathConfig.
13            NumberReturnType.DOUBLE))
13        .redirect(redirectConfig().followRedirects(false))
14    wireMock__https__www_doesnotexistfoo_test__6789 = WireMockServer(
15        WireMockConfiguration()
16        .bindAddress("127.0.0.22")
17        .httpsPort(6789)
18        .extensions(ResponseTemplateTransformer(false)))
19    wireMock__https__www_doesnotexistfoo_test__6789.start()
20 }

```

Fig. 23. BeforeAll of the generated suite for the SUT (i.e., *Auth0*) in Figure 16.

```

1 @BeforeEach
2 fun initTest() {
3     controller.resetStateOfSUT()
4     wireMock__https__www_doesnotexistfoo_test__6789.resetAll()
5     wireMock__https__www_doesnotexistfoo_test__6789.stubFor(
6         any(anyUrl())
7         .atPriority(100)
8         .willReturn(
9             aResponse()
10                .withHeader("Connection", "close")
11                .withHeader("Content-Type", "text/plain")
12                .withStatus(404)
13                .withBody("Not Found")
14            )
15     )
16    DnsCacheManipulator.clearDnsCache()
17 }

```

Fig. 24. BeforeEach of the generated suite for the SUT (i.e., *Auth0*) in Figure 16.

without depending on any external dependencies, as mentioned in Section 5. In this particular example, the test suite will contain all the necessary configurations to run a mock server successfully without any user interactions. Such generated test suites can be used directly in the development pipelines without any additional steps.


```

1 @Test @Timeout(60)
2 fun test_1() {
3     DnsCacheManipulator.setDnsCache("www.doesnotexistfoo.test", "127.0.0.22")
4     assertNotNull(wireMock__https__www_doesnotexistfoo_test__6789)
5     wireMock__https__www_doesnotexistfoo_test__6789.stubFor(
6         post(urlEqualTo("/oauth/token"))
7         .atPriority(1)
8         .willReturn(
9             aResponse()
10                .withHeader("Connection", "close")
11                .withHeader("Content-Type", "application/json")
12                .withStatus(201)
13                .withBody("{ \"id_token\": \"W7\", \" +
14                \"refresh_token\": \"KW\", \" +
15                \"expires_in\": -7023511200453181075\" +
16                \"}\"
17            )
18        )
19    )
20 }
21
22
23 given().accept("*/")
24     .header("x-EMextraHeader123", "")
25     .get("${baseUrlOfSut}/api/wm/auth0")
26     .then()
27     .statusCode(200)
28     .assertThat()
29     .contentType("text/plain")
30     .body(containsString("OK"))
31
32
33 wireMock__https__www_doesnotexistfoo_test__6789.resetAll()
34 }

```

Fig. 25. One unit test taken from the generated test suit for the SUT *Auth0* in Figure 16.

As shown in Figure 25, the code initially configures the required DNS information using `DnsCacheManipulator` to redirect the traffic to our mock server (Line 3). Secondly, after the successful assertion of an active mock server, the relevant stubs related to the successful completion of the test will be configured on the respective mock server. In this case, to reach the particular line that returns HTTP 200 (Line 27) with the body "OK" (Line 30). Once the necessary assertions are executed, the test will reset the mock server to its default state. Finally, at the end of all test case executions, we make sure that everything shuts down gracefully in a JUnit `AfterAll` call, as shown in Figure 26.

For all these seven artificial APIs, during our experiments, we were able to produce usable test suites. Through our approach, we were able to generate test suites that can cover all the special cases we designed our artificial examples for. With our mocking techniques, full coverage is achieved on all these artificial APIs in a matter of seconds. As all those APIs can now be reliably solved with

```

1 @AfterAll
2 @JvmStatic
3 fun tearDown() {
4     controller.stopSut()
5     wireMock__https__www_doesnotexistfoo_test__6789.stop()
6     DnsCacheManipulator.clearDnsCache()
7 }

```

Fig. 26. AfterAll of the generated suite for the SUT (i.e., *Auth0*) in Figure 16.

Table 4. Results of the line coverage and detected faults achieved by *Base* and *Mocking* with internet *On*. We also report pair comparison results between *Base* and *Mocking* using Vargha-Delaney effect size (i.e., \hat{A}_{12}) and Mann-Whitney U test (i.e., p -value at significant level 0.05). The number of executed HTTP calls during the search is reported as well.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	Mocking	\hat{A}_{12}	p -value	Base	Mocking	\hat{A}_{12}	p -value	Base	Mocking	Ratio
<i>catwatch</i>	49.0	47.5	0.40	0.206	25.4	47.0	1.00	< 0.001	9412	4047	42.99
<i>cwa-verification</i>	55.6	56.6	0.78	< 0.001	9.1	11.1	0.83	< 0.001	111346	116944	105.03
<i>genome-nexus</i>	36.9	36.7	0.31	0.013	17.3	20.8	0.98	< 0.001	17106	27350	159.89
<i>ind1</i>	13.3	13.4	0.58	0.335	57.6	59.3	0.66	0.045	109156	108466	99.37
<i>pay-publicapi</i>	11.1	31.8	1.00	< 0.001	26.6	40.9	0.93	< 0.001	145104	25603	17.64
Average	33.2	37.2	0.61		27.2	35.8	0.88		78425	56482	84.98
Median	36.9	36.7	0.58		25.4	40.9	0.93		109156	27350	99.37

our novel techniques in a short amount of time, those have been added with E2E tests in the CI build of EvoMASTER. All these artificial APIs can be found in the EvoMASTER's code repository.⁹

RQ1: Our approach, using white-box heuristics and taint analysis, demonstrates its successful capability to guide automatic mock external web service generation for artificial APIs.

6.3 Results for RQ2

To answer **RQ2**, we report line coverage and number of detected faults on average (i.e., *mean*) with 30 repetitions achieved by *Base* and *Mocking*, as shown in Table 4. To further demonstrate how the lines are covered throughout the search, we plot the number of covered lines at every 5% intervals of the search budget for each setting in each SUT, as shown in Figure 27. The same kinds of graphs for the fault detection criterion are reported in Figure 28.

Line coverage is measured with EvoMASTER's own bytecode instrumentation. Fault detection is based on the automated oracles used in EvoMASTER to detect faults. At the time of these experiments, we used returned 500 HTTP status codes (i.e., server errors) per endpoint as automated oracle. This is a common practice in the research literature on fuzzing REST APIs [25]. No artificial fault was injected in the SUTs. The SUTs used in the experiments are "as-they-are", without modification. All detected faults are actual faults, albeit different faults have different severity and importance [35].

Figure 29 contains a test obtained from the generated test suite for *catwatch* to illustrate the reported faults. The selected test simulates a situation in which the external service is available. However, the response body is invalid. In Line 2, using *DnsCacheManipulator*, the traffic to *api.github.com* will be redirected to a loopback address where the respective mock server is running. Similarly, in Line 3, the traffic to *info.services.auth.zalando.com* will be redirected to

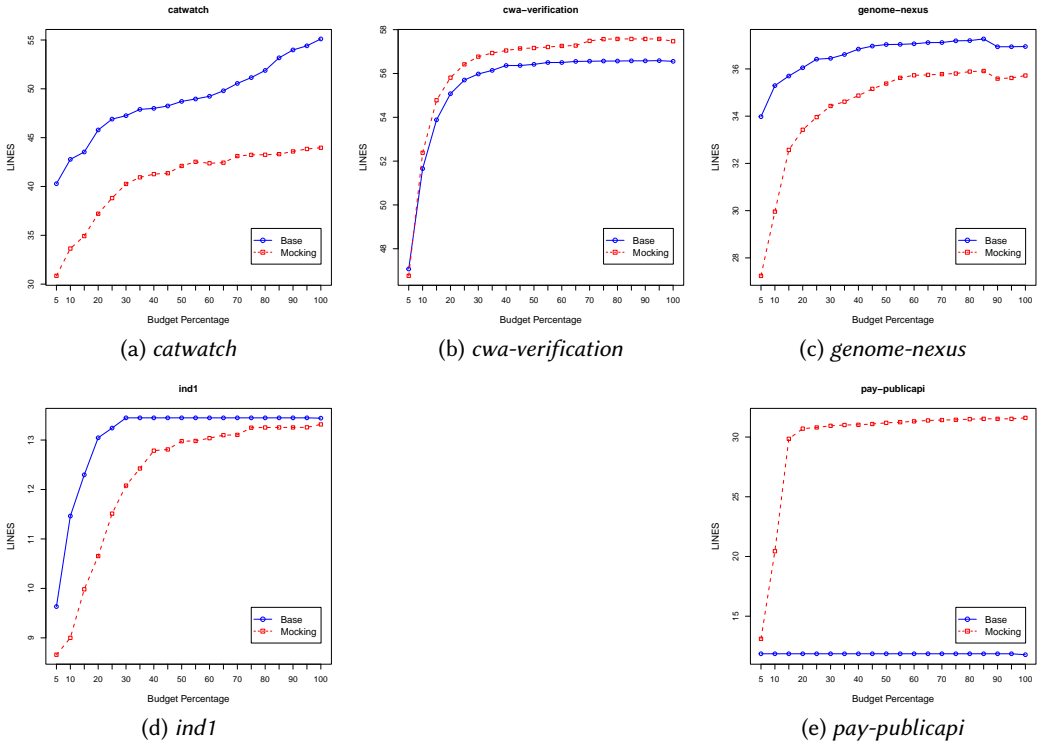


Fig. 27. The average covered lines (y-axis) with 30 runs achieved by On-Base and On-Mocking (RQ2), reported at 5% intervals of the budget allocated for the search (x-axis).

another mock server. Line 5 assigns the stub to the mock server to return HTTP 200 with an invalid body as the response. Such a response will trigger the application to crash with the error message related to the exception, as seen in Line 33. HTTP 500 is a generic server error message, which is usually returned when there is a crash (e.g., an uncaught thrown exception) in the business logic of the executed endpoint due to a software bug. Generic crashes can be considered as indicative of a fault, as in this case. If the external services an endpoint relies on are either currently unavailable or return invalid data, the correct response statuses should either be 503 (Service Unavailable) or 424 (Failed Dependency).

Detailed analyses based on these results are presented next. Before going into those details though, one thing that is important to notice is that the achieved line coverage values reported in Table 4 might be lower than what reported in Figure 27. This is not a mistake, and it is actually expected. The values presented in Table 4 are after the search is completed, and after a test *minimization* phase is executed. In such a minimization phase, redundant HTTP calls that do not contribute to the achieved coverage are removed. To check this property, test cases must be re-evaluated. Information about the code that is executed only once would be lost, though. This may be due to the fact that static initializers are not considered for coverage metrics in the instrumentation of EvoMASTER. However, coverage information about constructors executed in singleton classes (a typical case in Spring and JEE service beans) would be lost as well after this minimization phase. Unfortunately, automatically determining if a constructor belongs to a singleton class is not straightforward. As these issues might only affect reported values in scientific articles, and would have little to no effect

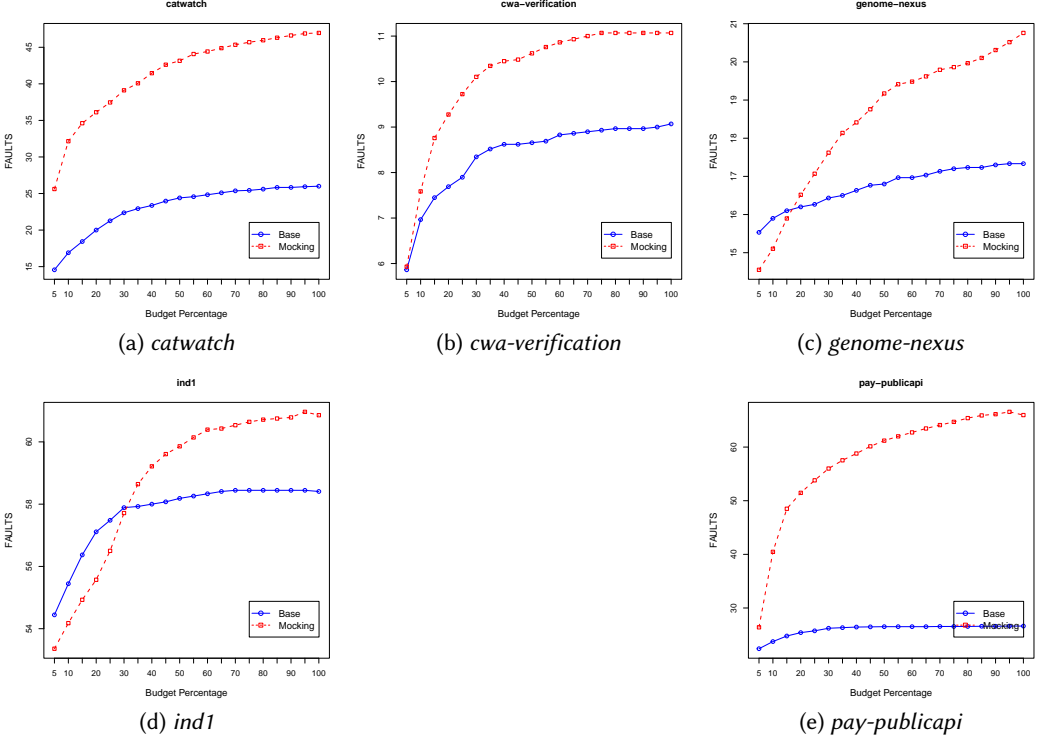


Fig. 28. The average number of detected faults (y-axis) with 30 runs achieved by On-Base and On-Mocking (RQ2), reported at 5% intervals of the budget allocated for the search (x-axis).

to practitioners using EVOMASTER, they are not a major concern (a relatively simple fix could be to collect final coverage metrics before the minimization phase).

Based on the analysis of the results reported in Table 4, compared to *Base*, we found that *Mocking* achieved significant improvements on both metrics, i.e., line coverage and detected faults. On the one hand, the results show that, regarding fault detection, for all the five selected APIs, our approach (i.e., *Mocking*) significantly outperformed *Base* with low p -values and high \hat{A}_{12} . On the other hand, for line coverage, statistically significant improvements were obtained only on two APIs (i.e., *cwa-verification* and *pay-publicapi*), while worse on one (i.e., *genome-nexus*). Those results demonstrate the effectiveness of our approach, which uses white-box heuristics and taint analysis for generating mock external web services.

As we mentioned earlier in Section 6.1.2, *catwatch* relies heavily on GitHub APIs. Due to the rate limiting, we observed impacts on the final results in the line coverage as shown in Figure 27. The results for *catwatch* are lower than *Base* (i.e., 49.0%) for *Mocking* (i.e., 47.5%) in line coverage, although such difference is not statistically significant. At the same time, when comparing detected faults, *catwatch* performed better in *Mocking* (i.e., 47.0 faults detected on average) compared to *Base* (i.e., 25.4).

Regarding the *cwa-verification* API, in absolute terms, there is not much difference, only $56.8\% - 55.8\% = 1.0\%$ in line coverage. However, such difference is statistically significant (i.e., p -value < 0.001), with a high value effect-size (i.e., 0.78). We observed similar patterns in the results obtained for RQ3, which will be presented in Table 5. This is because the *cwa-verification* is part of

```

1 ...
2 DnsCacheManipulator.setDnsCache("api.github.com", "127.0.0.5");
3 DnsCacheManipulator.setDnsCache("info.services.auth.zalando.com", "127.0.0.6");
4 assertNotNull(wireMock__https__info_services_auth_zalando_com__443);
5 wireMock__https__info_services_auth_zalando_com__443.stubFor(
6     get(urlEqualTo("/oauth2/tokeninfo"))
7     .atPriority(1)
8     .willReturn(
9         aResponse()
10            .withHeader("Connection", "close")
11            .withHeader("Content-Type", "application/json")
12            .withStatus(200)
13            .withBody("i6W_KU5cGJM")
14    )
15);
16
17
18 // Fault100. HTTP Status 500. org/zalando/stups/oauth2/spring/server/
19 // TokenInfoResourceServerTokenServices_102_getMap GET:/languages
20 // Fault200. Received A Response From API That Is Not Valid According To Its
21 // Schema. GET:/languages -> Response status 500 not defined for path '/languages
22 // '.
23 given().accept("application/json")
24 .header("x-EMextraHeader123", "")
25 .get(baseUrlOfSut + "/languages?" +
26     "organizations=_EM_153_XYZ_" +
27     "limit=341" +
28     "offset=367" +
29     "q=_EM_152_XYZ_" +
30     "access_token=Gt")
31 .then()
32 .statusCode(500) // org/zalando/stups/oauth2/spring/server/
33 // TokenInfoResourceServerTokenServices_102_getMap
34 .assertThat()
35 .contentType("application/json")
36 .body("'status'", numberMatches(500.0))
37 .body("'error'", containsString("Internal Server Error"))
38 ...

```

Fig. 29. A generated test for *catwatch*, simulating an application crash scenario, relies on an external web service.

a microservice architecture, and the interacted service is not up and running on the local machine we used for these experiments. Thus, the two settings for this case study are practically the same.

To get more insight, Figure 30 shows a part of the code of *cwa-verification* (code snippet of lines 117--124 in *ExternalTestStateController*²³). This code can be covered by *Mocking*, but never by *Base* within its 30 repetitions. The code in Figure 30 is used to interact with an external web service, to fetch some specific information. Such information will be validated first. Only valid information can be used to implement the business logic of POST */version/v1/testresult* endpoint in the *cwa-verification* (Lines 4 -- 7). As the datatype is clear, when parsing the response

```

1 TestResult testResultDob = testResultServerService.result(hashDob);
2
3 // TRS will always respond with a TestResult so we have to check if both results
  are equal
4 if (testResultDob.getTestResult() != testResult.getTestResult()) {
5     // given DOB Hash is invalid
6     throw new VerificationServerException(HttpStatus.FORBIDDEN,
7     "TestResult of dob hash does not equal to TestResult of hash");
8 }

```

Fig. 30. Code snippet taken from the *cwa-verification*.

```

1 rawValue = this.fetcher.fetchRawValue(this.buildRequestBody(subSet));
2 ...
3 rawValue = this.normalizeResponse(rawValue);
4 ...
5 List<T> fetched = this.transformer.transform(rawValue, this.type);

```

Fig. 31. Code snippet taken from the *genome-nexus* demonstrating external web service interaction.

(i.e., `TestResult.java`²³), with our approach (i.e., *Mocking*) the structure of the responses can be easily inferred. During fuzzing, the search can update valid values in the responses to cover more cases in the business logic of the API.

For the *genome-nexus*, we found that most interactions between the SUT and external web services are to fetch data, then saving into a database (i.e., MongoDB). In this case, it is not necessary to extract the data when fetching it from the external web service, rather such data or schema extraction can also be performed when saving data into the database. The code snippet of an external web service interaction and data extraction in *genome-nexus* is shown in Figure 31 (the complete code can be found in `BaseCachedExternalResourceFetcher.java`²³).

Based on the code in Figure 31, a fetch operation is performed initially to get raw data (Line 1). By checking the implementation of `fetchRawValue` (e.g., `BaseExternalResourceFetcher.java`²³), a general type (such as `BasicDBObject`²⁴) is provided. With such *Type* information, we can only know it is a list. After that, the actual data extraction based on raw value if performed before saving it into the database (Line 5). Our current handling does not support such multi-level schema extraction of responses yet. This is going to be an important future work. Due to this issue, the line coverage performance of the *genome-nexus* case study shows no improvements, but rather a small, albeit statistically significant, performance loss of 0.2%. On the other hand, we observed considerable performance improvements in detected faults (i.e., 17.3% vs. 20.8%). This is likely due to mocking technique enabling testing different error scenarios that are not properly handled by the SUT.

ind1 shows little difference in both metrics. When compared to *Base* (i.e., 13.3%), *Mocking* (i.e., 13.4%) showed a marginal improvement in terms of line coverage. Correspondingly, in detected faults, we can see a small improvement in *Mocking* (i.e., 59.3) compared to *Base* (i.e., 57.6). Due to confidentiality, we are not allowed to share the code of this API. However, it is still important to understand why these kinds of results were obtained. By studying its source code, it seems that most of its endpoints start their execution by calling an external service. But, then, the computation

²³<https://github.com/EMResearch/EMB>

²⁴<https://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBObject.html>

Table 5. Results of the line coverage and detected faults achieved by *Base* and *Mocking* with internet *Off*. We also report pair comparison results between *Base* and *Mocking* using Vargha-Delaney effect size (i.e., \hat{A}_{12}) and Mann-Whitney U test (i.e., p -value at significant level 0.05). The number of executed HTTP calls during the search is reported as well.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	Mocking	\hat{A}_{12}	p -value	Base	Mocking	\hat{A}_{12}	p -value	Base	Mocking	Ratio
<i>catwatch</i>	46.3	49.6	0.73	0.003	41.3	54.8	1.00	< 0.001	26474	25274	95.47
<i>cwa-verification</i>	55.8	56.8	0.79	< 0.001	9.0	10.8	0.86	< 0.001	113984	121780	106.84
<i>genome-nexus</i>	28.5	30.0	0.83	< 0.001	20.6	20.6	0.48	0.761	98623	40852	41.42
<i>ind1</i>	13.9	14.4	0.70	0.007	58.2	60.0	0.79	< 0.001	124362	102088	82.09
<i>pay-publicapi</i>	11.1	31.6	1.00	< 0.001	27.2	41.0	0.84	< 0.001	150263	28965	19.28
Average	31.1	36.5	0.81		31.3	37.4	0.79		102741	63792	69.02
Median	28.5	31.6	0.79		27.2	41.0	0.84		113984	40852	82.09

soon crashes due to a null pointer exception (NPE). This is based on a query by resource id to the API's SQL database. The SQL taint analysis used by EvoMASTER [10] correctly infers the id of one of the resources used in the provided test data of the API. But, such existing data has some empty columns, whose data will lead to a NPE when used in the code of the SUT. However, no search operators are currently defined in EvoMASTER to modify existing SQL test data (can only add new data [10]), nor there is any heuristic in the fitness function to reward queries that return non-null columns. Addressing interactions with external services is only one of the many challenges in fuzzing REST APIs. Other challenges like dealing with SQL databases, although addressed [10], are not fully solved yet, as the case of *ind1* shows. In an API, all these challenges can have unexpected interactions and side-effects.

The *pay-publicapi* performed better in both metrics, as the majority of its functionalities are dependent on external web services. When compared with *Base* (i.e., 11.1%), *pay-publicapi* shows a drastic improvement in *Mocking* (i.e., 31.8%, which is more than 20%) in terms of line coverage. Likewise, we observed a similar large improvement in the number of detected faults when comparing *Base* (i.e., 26.6) with *Mocking* (i.e., 40.9).

Overall, compared to *Base*, *Mocking* helped to achieve better results in both metrics as our approach successfully generated mock external web services. On the one hand, the effect is stronger for fault detection, as the use of mocking enables to test some error scenarios that would not be feasible to test with only the real services. On the other hand, the exploration of error scenarios could take time from the search for code coverage, giving some minor side-effects in some cases. But, theoretically, this could change with larger search-budgets (e.g., when fuzzing for more than one hour).

RQ2: Our approach with white-box heuristics and taint analysis demonstrates its effectiveness in guiding mock external web service generation, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e., EvoMASTER) are statistically significant in most of the five selected APIs. On one API, line coverage improvements were up by +20%.

6.4 Results for RQ3

To answer **RQ3**, we ran experiments with 30 repetitions as reported in the Table 5. Similar to **RQ2**, we report line coverage and detected faults on average for **RQ3**. We plotted the line coverage throughout the search with 5% intervals for the given budget in Figure 32. The same way, we

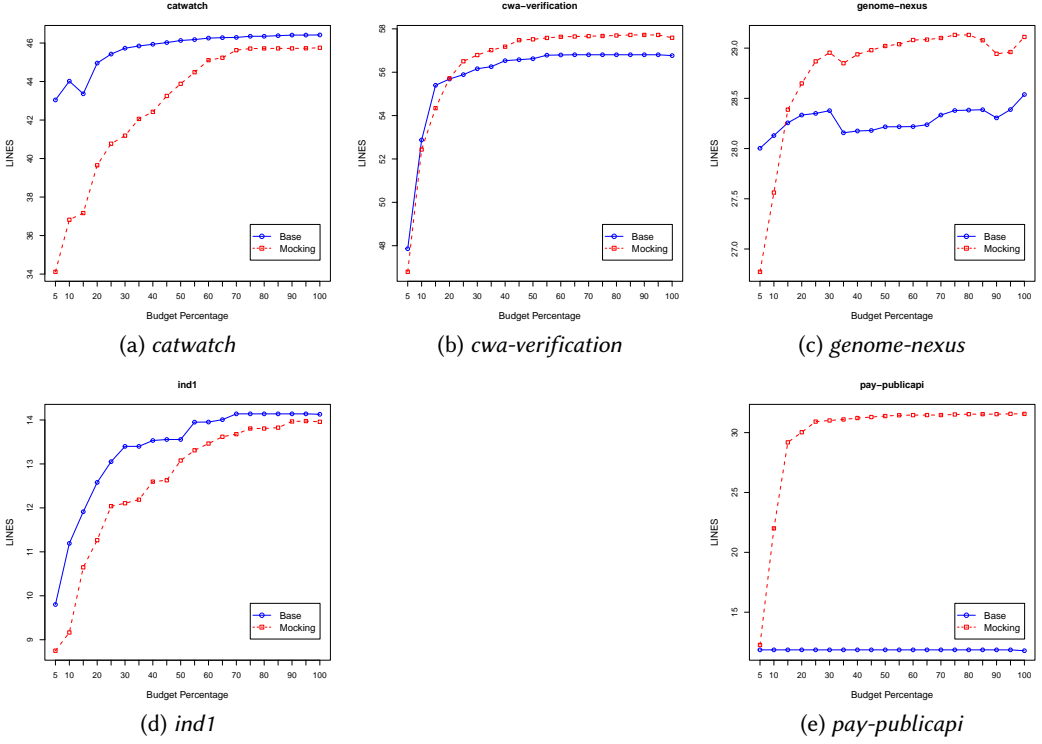


Fig. 32. Average covered lines (y-axis) with 30 runs achieved by Off-Base and Off-Mocking (RQ3), reported at 5% intervals of the budget allocated for the search (x-axis).

reported the fault detection criterion as graphs in Figure 33. Further, we conducted an extended analysis of the results as follows.

Based on the experiment results reported in Table 5, compared with *Base*, it can be seen *Mocking* achieved consistent improvements on both metrics, i.e., line coverage and fault detection. Whereas with Internet *On* improvements with line coverage were obtained on only two APIs (recall Table 4), here with Internet *Off* line coverage improvements are obtained for all APIs. Interestingly though, in contrast to what is reported in Table 4, no improvements for fault detection are detected here in Table 5 for *genome-nexus*.

Similar to the results for **RQ2** in Section 6.3, with the *catwatch*, we observed significant impacts on the final results of line coverage, as shown as well in Figure 27. Line coverage is improved by more than 3% (Table 5), and on average 13 more faults are detected. As previously discussed in Section 6.1.2, results for *catwatch* with Internet *On* in Table 4 are not reliable, due to the rate limiter imposed by GitHub. Such a rate limiter has no impact on the results reported in Table 5.

Regarding *cwa-verification*, such API makes no connections to web services on the internet. Therefore, one would not expect any difference between the results reported in Table 4 and in Table 5. All the reported small differences can be explained due to the stochastic nature of repeating experiments 30 times.

Regarding the *genome-nexus*, we observe performance improvements (+1.5%) in line coverage compared to the results in Table 4 from **RQ2**. Still, results for *Mocking* are better with Internet *On* (i.e., 36.7 vs. 30.0). This shows the importance of using techniques like Harvester (recall Section 4.6) to exploit existing, real data. Regarding the number detected faults, as previously mentioned, there

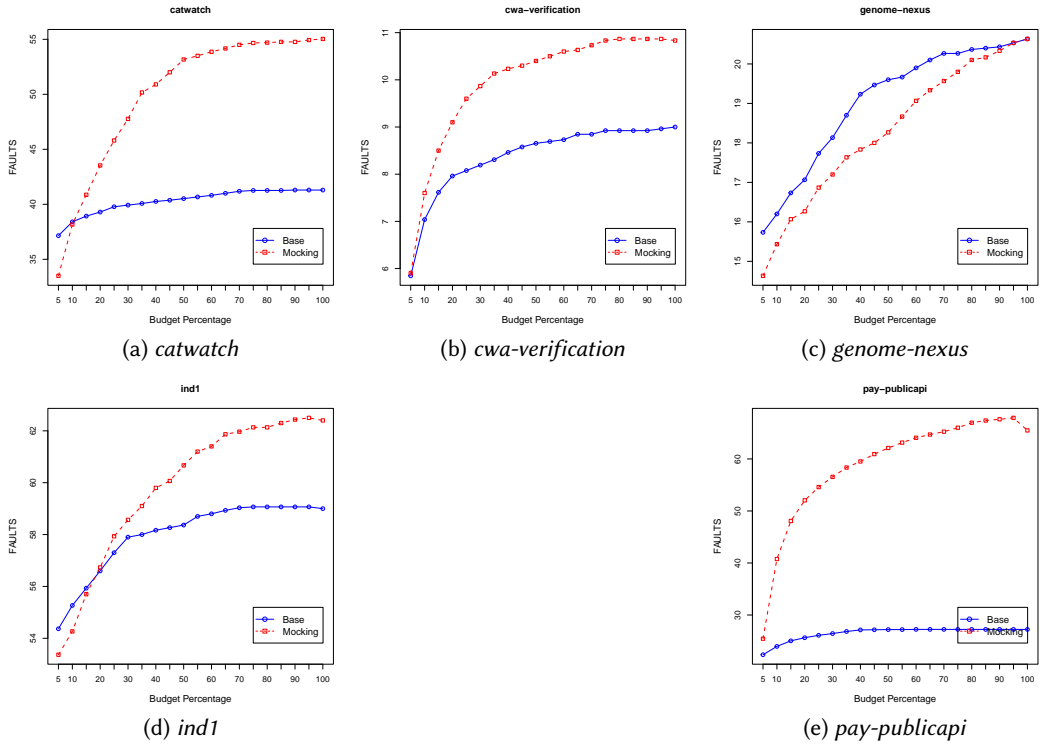


Fig. 33. The average number of detected faults (y-axis) with 30 runs achieved by Off-Base and Off-Mocking (RQ3), reported at 5% intervals of the budget allocated for the search (x-axis).

is no difference between *Base* (i.e., 20.6%) and *Mocking* (i.e., 20.6%). This could be explained if all the new detected faults are related to an improper handling of missing connections.

Similar to the results for **RQ2** in Table 4, *ind1* shows marginal improvements in both metrics. In line coverage, we observe a slight, but statistically significant, improvement (i.e., 0.5%) in the performance for *Mocking* (i.e., 14.4%) compared with *Base* (i.e., 13.9%). In the same manner, in detected faults *Mocking* (i.e., 60.0%) shows a small increase when compared to *Base* (i.e., 58.2%). For both approaches, we see slightly higher metrics for Internet *Off*. At this point in time, we are not able to provide plausible conjectures to explain this phenomenon.

Similarly, the *pay-publicapi* performed well under both metrics both in internet-enabled experiments (Table 4) and internet-disabled experiments (Table 5). We observed *Mocking* (i.e., 31.6%) performing well compared to *Base* (i.e., 11.1%) in terms of both line coverage and detected faults (i.e., *Mocking* 41.0% vs. *Base* 27.2%).

It is evident that *Mocking* helped to achieve better results in both metrics compared to *Base*. To summarize, our methodology has successfully generated mock external web services without the necessity of an internet connection.

RQ3: Our approach has demonstrated effectiveness in generating mock external web services without an internet connection, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e., EvOMASTER) are statistically significant on all the five selected APIs.

6.5 Discussion

Overall, based on the results of our experiments, it is clear that our approach performs well in all seven artificial APIs and in all five selected real-world APIs.

All artificial APIs introduced can be fully covered, reliably, by our novel techniques. As those APIs would be impossible to fully cover without manipulating the external services they connect to, no existing black-box approach [25] would be able to handle them. Therefore, our novel techniques push forward the state-of-the-art in fuzzing Web APIs.

However, results on artificial examples might not necessarily apply to real-world APIs. Our experiments provide scientific evidence to support our claim that our novel techniques provide useful improvements for real-world APIs as well. Results for fault detection were stronger than for line coverage. This can be explained by the fact that, with mocking, we can efficiently test several different error scenarios.

How much improvement can be achieved strongly depends on how reliant the SUT is on communications with external services. If a large API is doing calls to an external service only in one of its endpoints, no many improvements “on average” over the whole API would be achieved, regardless of how good such a case is handled. On the other hand, if the SUT strongly relies on external services, like in the case of *pay-publicapi*, large and substantial improvements can be achieved.

Not all APIs are the same. Handling external services is only one of the current main open problems in fuzzing REST APIs (as identified in [54]). This paper addressed one of the major challenges identified in [54], by providing an automated working solution, implemented in an open-source, state-of-the-art fuzzer, i.e., EvoMASTER. Still, there are several research challenges that need to be addressed to further improve the obtained results.

White-box heuristics require instrumentation in the code. This has a computational cost, as more code is executed when evaluating a test case. This is particularly the case when dealing with mocked servers, as those can introduce further computational overheads in their handling. The longer a test case takes to be executed, the fewer number of tests can be evaluated within the same amount of time (e.g., 1-hour fuzzing sessions). Executing fewer test cases means exploring a smaller subset of the search space, which unfortunately could lead to worse results in the end. This is a tradeoff between the quality of a fitness function (e.g., measured in how good it is in “smoothing” the search landscape) and its computational cost (which would reduce the number of fitness evaluations within the same amount of time). Whether it pays-off in the end is not guaranteed.

Our novel techniques presented in this paper are composed of many components that impact the computational cost of each fitness evaluation. Measuring the computational cost of each of them individually would not be viable. However, an indirect measure for performance cost could be to check how many HTTP calls are made during a fuzzing session, given the same amount of time. The higher the computational cost, the fewer calls would be made within the same amount of time. Table 4 and Table 5 show such data. In cases like *catwatch*, *genome-nexus* and *pay-publicapi*, there is a major drop in number of HTTP calls that are made within an hour. Unfortunately, though, this might not be directly explainable only based on heuristics overhead, as the cost of each fitness function in our context is not constant. For example, a call that returns immediately with a 4xx status code due to invalid inputs would be much quicker to execute compared to a 2xx call that executes large parts of the SUT’s business logic, including communications with databases and external services. A novel technique that achieves higher code coverage could evolve test cases that are more computationally expensive to run.

In the end, in our empirical study, this tradeoff paid-off in the end, as even with lower number of HTTP calls, higher code coverage and fault detection were achieved. However, how this would scale to APIs that use a larger number of external communications is something that would be up to empirical investigation.

7 Threats to Validity

Internal validity. Our experiment results are derived using a software tool. This can lead to threats to internal validity, due to possible faults in our implemented software. During its development, the software tool was tested using several unit and end-to-end tests (the latter using the existing infrastructure provided by EvoMASTER [14]). By contrast, we cannot guarantee that our implementation is fault-free. The tool⁹ and the chosen SUTs⁸ used in the experiments are accessible as open-source code on GitHub. Anyone can review them.

Moreover, our experiments are based on a fuzzer that employs some randomness components as part of its internal engine. To take this into account, experiments were repeated 30 times with different initializing random seeds. The results were then analyzed using the appropriate statistical tests, using common practices in the software engineering research literature [6].

In addition, interaction with live web services on the internet are problematic from a research standpoint. They can add noise to the results, especially if they have high variability in the returned HTTP responses, given the same inputs. Although repeating the experiments 30 times helps to reduce the negative impact of such noise on our conclusions, we cannot guarantee that our results can be fully replicated in the future. For example, between the time the experiments were run and the time of reading this text, those web services might no longer exist.

External validity. Our experiments were conducted on five (four open-source and one industrial) selected REST APIs. The four open-source APIs are all the APIs in EMB [16] that deal with connections with external web services. Although enterprise applications typically use external web services, only a few can be found in open-source repositories. Furthermore, running experiments on system-level testing is time-consuming. This limits the number of APIs that can be used for this kind of empirical analysis. Although we used both open-source and industrial APIs, the results of our empirical analysis cannot be generalized at this stage. Therefore, there is a need to investigate more APIs in upcoming studies.

Our novel white-box techniques are implemented for APIs running on the JVM, written for example in either Java or Kotlin. However, there are other popular languages used to build Web APIs, such as JavaScript/TypeScript, C#, Python, Ruby, PHP and Go. Whether the application of our novel techniques on these other languages would require just technical work, or novel research, is something we do not know at the moment. Our previous work on supporting white-box testing in other programming languages would suggest that it should be rather straightforward when dealing with statically-typed languages (e.g., C# [24]), but there could be few research challenges to address when dealing with dynamically-typed languages (e.g., JavaScript [56]). Regardless, even if our novel techniques would only work on the JVM, this latter is so popular in industry to build enterprise applications (e.g., considering the popularity of frameworks such as SpringBoot²⁵) that it could still have meaningful impact on industrial practice.

Our novel techniques have been implemented as an extension to the state-of-the-art white-box fuzzer EvoMASTER. Integrating our novel techniques in another evolutionary-based fuzzer should be rather straightforward. However, how to integrate them in other different types of fuzzers is an open research question. Since its inception in 2016, at the time of writing in 2025 EvoMASTER is still the only fuzzer for Web APIs (including GraphQL and RPC) that supports white-box fuzzing,

²⁵<https://theirstack.com/en/technology/spring-boot>

albeit the large interest in the research community about fuzzing REST APIs [25] (which so far it has focused on black-box testing). We cannot speculate on how our novel techniques could be integrated in other fuzzers that do not exist, yet.

8 Conclusion

In this paper, we have provided a novel search-based approach to enhance white-box fuzzing with automated mocking of external web services. Our techniques have been implemented as an extension of the fuzzer EvoMASTER [13], using WireMock for the mocked external services.⁹ Experiments on four open-source and one industrial APIs, along with seven artificial APIs show the effectiveness of our novel techniques, both in terms of code coverage and fault detection. On one real-world API (i.e., *pay-publicapi*), line coverage improvements were more than +20%.

To the best of our knowledge, this is the first work in the literature to offer a working solution for tackling this important problem, i.e., how deal with external services in fuzzing of Web APIs. Therefore, there are several avenues for further enhancements of our techniques in future work. An example is how to effectively deal with APIs that have a 2-phase parsing of JSON payloads (like in the case of *genome-nexus* in our case study).

Acknowledgment

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (EAST project, grant agreement No. 864972). Man Zhang is supported by the State Key Laboratory of Complex & Critical Software Environment (CCSE-2024ZX-01).

References

- [1] [n.d.]. AFL. <https://github.com/google/AFL>.
- [2] 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.
- [3] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195--206.
- [4] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [5] Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72--78.
- [6] 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.
- [7] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 79--90.
- [8] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 155--165.
- [9] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private api access and functional mocking in automated unit test generation. In *2017 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 126--137.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. 2023. Building an open-source system test generation tool: lessons learned and empirical analyses

- with EvoMaster. *Software Quality Journal* (2023), 1--44.
- [15] Andrea Arcuri, Man Zhang, and Juan Pablo Galeotti. 2024. Advanced White-Box Heuristics for Search-Based Fuzzing of REST APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024). <https://doi.org/10.1145/3652157>
 - [16] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P Galeotti, Bogdan Marculescu, and Susruthan Seran. 2023. EMB: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 433--442.
 - [17] Andrea Arcuri, Man Zhang, Susruthan Seran, Juan Pablo Galeotti, Amid Golmohammadi, Onur Duman, Agustina Aldasoro, and Hernan Ghianni. 2025. Tool report: EvoMaster—black and white box search-based fuzzing for REST, GraphQL and RPC APIs. *Automated Software Engineering* 32, 1 (2025), 1--11.
 - [18] Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. 2024. DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1383--1394.
 - [19] DNS Cache Manipulator [n.d.]. DNS Cache Manipulator. <https://github.com/alibaba/java-dns-cache-manipulator>.
 - [20] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7--8, 2020, Proceedings 12*. Springer, 105--120.
 - [21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416--419.
 - [22] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70--76.
 - [23] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40--44.
 - [24] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. .NET/C# instrumentation for search-based software testing. *Software Quality Journal* (2023), 1--27.
 - [25] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023). <https://doi.org/10.1145/3617175>
 - [26] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 172--183.
 - [27] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. 2004. Testability Transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3--16.
 - [28] Nikolas Havrikov, Alessio Gambi, Andreas Zeller, Andrea Arcuri, and Juan Pablo Galeotti. 2017. Generating unit tests with structured system interactions. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. IEEE, 30--33.
 - [29] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive rest api testing with reinforcement learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 446--458.
 - [30] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289--301. <https://doi.org/10.1145/3533767.3534401>
 - [31] Jiangchao Liu, Jierui Liu, Peng Di, Alex X Liu, and Zexin Zhong. 2022. Record and replay of online traffic for microservices with automatic mocking point identification. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 221--230.
 - [32] Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-testing: unit testing with mock objects. *Extreme programming examined* (2000), 287--301.
 - [33] Kazuaki Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*. IEEE, 177--182.
 - [34] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312--2331.
 - [35] 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.
 - [36] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*. 149--153.

- [37] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231--241.
- [38] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1393--1403.
- [39] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32--44. <https://doi.org/10.1145/96267.96279>
- [40] Shaikh Mostafa and Xiaoyin Wang. 2014. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th international conference on quality software*. IEEE, 127--132.
- [41] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An analysis of public REST web service APIs. *IEEE Transactions on Services Computing* (2018).
- [42] Sam Newman. 2021. *Building microservices*. " O'Reilly Media, Inc."
- [43] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1224--1228.
- [44] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1--74.
- [45] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460--465.
- [46] Carlos Rodriguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering*. Springer, 21--39.
- [47] Omur Sahin and Bahriye Akay. 2021. A Discrete Dynamic Artificial Bee Colony with Hyper-Scout for RESTful web service API test suite generation. *Applied Soft Computing* 104 (2021), 107246.
- [48] Susruthan Seran, Man Zhang, and Andrea Arcuri. 2023. Search-Based Mock Generation of External Web Service Interactions. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer.
- [49] Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 402--412.
- [50] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving test case generation for REST APIs through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 117--128.
- [51] Dave Thomas and Andy Hunt. 2002. Mock objects. *IEEE Software* 19, 3 (2002), 22--24.
- [52] LS Veldkamp, Mitchell Olsthoorn, and A Panichella. 2023. Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs. In *The 16th International Workshop on Search-Based and Fuzz Testing*. IEEE/ACM.
- [53] 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).
- [54] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023). <https://doi.org/10.1145/3597205>
- [55] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1--38.
- [56] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2023. JavaScript SBST Heuristics To Enable Effective Fuzzing of NodeJS Web APIs. *ACM Transactions on Software Engineering and Methodology* (2023).
- [57] 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.
- [58] 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.
- [59] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s, Article 230 (sep 2022), 36 pages. <https://doi.org/10.1145/3512345>
- [60] Hubert Zimmermann. 1980. OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications* 28, 4 (1980), 425--432.