Search-Based Mock Generation of External Web Service Interactions

Susruthan Seran$^{1[0000-0003-1800-060X]}$, Man Zhang$^{1[0000-0003-1204-9322]}$, and Andrea Arcuri$^{1,2[0000-0003-0799-2930]}$

[1] Kristiania University College, Norway
{susruthan.seran,man.zhang,andrea.arcuri}@kristiania.no
[2] Oslo Metropolitan University, Norway

Abstract. 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 REST APIs). Although several techniques in the research literature have been shown to be effective at generating test cases in many 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. Furthermore, external services may not be under the control of the same developers of the tested application.
In this paper, we present a novel search-based approach aimed at *fully automated* mocking 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 it automatically setups responses in those mock web server instances, aiming at maximizing code coverage and fault-finding. An empirical study on 3 open-source REST APIs from EMB, and one 1 industrial API from an industry partner, shows the effectiveness of our novel techniques, i.e., significantly improves code coverage and fault detection.

Keywords: Microservices · Automated Mock Generation · Search-Based Test Generation · Search-based Software Engineering.

1 Introduction

Enterprise software backends are typically large and complex. Microservice-based software architecture design helps to tackle the challenges in this domain [20]. In a microservice environment, distributed services are interconnected using communication protocols like HTTP(S), Remote Procedure Call (RPC), and Advanced Message Queuing Protocol (AMQP) in a synchronous or asynchronous manner, to execute a common goal [20]. Likewise, modern web and mobile applications also connect to each other for various reasons, for instance, using services for authentication purposes (e.g., OAuth) or for financial transactions (e.g., Stripe[3] and PayPal[4]). Especially during automated testing, dealing with such external dependencies can be tricky, as what executed in the tested application depends on what returned from those external APIs. 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.

A popular technique in industry to address this limitation is *mocking*. However, for *system-level testing* of web/enterprise applications, instead of using *mock objects*, use of mock external services 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, behavior of the external web service can be extended to test various scenarios (e.g., different returned HTTP status codes). Mocking an external service using a mock HTTP server is a more advanced and possibly less known case. A popular library for the JVM-based applications that can do this is WireMock[5].

In this paper, we provide novel search-based techniques to generate mock web services in a fully automated manner for white-box fuzzing, for the JVM. Using bytecode instrumentation, we can analyze all interactions with external web services during fuzzing, and automatically create mock servers where their responses will be part of the search process. We focus on JavaScript Object Notation (JSON) payloads, where the structure of the responses can be automatically inferred via taint analysis and on how parsing libraries such as Gson and Jackson are used by those tainted values.

Our novel techniques could 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), for example, RESTful APIs. For the experiments in this paper, we used our open-source fuzzer EvoMaster [11], as not only it gave the best results among fuzzer comparisons [16, 23], but also currently it seems the only fuzzer that supports white-box analyses for this kind of web/enterprise systems [13]. Our extension enables EvoMaster to generate mock external web service with no need of any manual modification of the System Under Test (SUT). Furthermore, we were able to generate self-contained test suites with integrated mocking capabilities to simulate the same production behavior without relying on any external services up and running.

Our novel developed techniques are evaluated on four RESTful APIs (3 open-source from the EMB corpus [2], and 1 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.

The novel contributions of this paper are:

[3] https://stripe.com/
[4] https://paypal.com/
[5] https://wiremock.org/

- A novel search-based approach in which HTTP mock servers are automatically instantiated when fuzzing web/enterprise applications.
- A novel use of taint-analysis to automatically infer the syntactic structure of the JSON payloads expected to be returned from these external services.
- A novel 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 6.9% on average on one of the tested APIs) and find 12 more faults.

2 Related Work

In industry, it is a common practice to use mocking techniques during unit testing to deal with dependencies [21]. The term "mocking" is typically associated in the literature with *unit testing*. For example, instead of passing as input an instance of a class that can make a call to an external service or database, a mock object can be used in which each returned value of its method calls can be configured programmatically directly in the tests (without the need to execute the original code). There are different libraries to help to instantiate and configure *mock objects*. For the JVM, popular libraries are for example Mockito,[6] EasyMock,[7] and JMock,[8] which significantly simplify the writing of mock objects.

Although mocking has several practical benefits [21], 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 needs 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 [12], which can generate mock object inputs using Mockito [7]. Similarly, Pex can setup mock objects related to the file system APIs [18].

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 major issue, especially if the CUT is doing operations on the file system (as random files could be created and deleted during the test generation, which could have disastrous 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 [5] and a virtual network [6, 15]. Still, there are major issues in creating the right data

[6] `https://site.mockito.org/`
[7] `https://easymock.org/`
[8] `https://www.jmock.org/`

with the right structure (e.g., files and HTTP responses) returned from the calls to the virtual environment [15].

For system testing, in which mock objects cannot be typically used as the SUT is executed through the user interfaces (e.g., a GUI or network calls in the case of web services), there is still the issue of how to deal with network calls to external services during testing. Instead of communicating with the actual external services, the SUT can be modified (e.g., via parameters to change the hostname/IP address of these services) to speak with a different server, on which the tester has full control on how responses are returned. Simulating/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 service or a mocked one, as it will still use the same code to make network calls (e.g., HTTP over TCP) and the same code to read and parse the responses. There are several tools and libraries available in industry which help to setup and run a mocked external web service with less effort, such as Mockoon,[9] Postman,[10] and WireMock.[11] All of them share common functionalities to create configurable mock HTTP(S) web services for testing.

Creating handwritten mock external web services can be tedious and time-consuming. Furthermore, if the third-party interactions are executed using a provided software development kit (SDK), unless the knowledge about the interactions is available or the SDK is reverse-engineered, the process becomes convoluted. A complementary approach is to use *record&replay* where each request and response is captured using a pass-through proxy by the mock server rather than configuring them manually. Later on, the captured information will be used to initiate 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, handwritten mock web services is not a scalable approach. However, record and replay would still be feasible.

The work done in [6, 15] is perhaps the closest in the literature to what we present in this paper. However, there are some significant major differences. First, we do not use a limited, artificial virtual network, but rather make actual HTTP calls using external mocked web services (using WireMock). To be able to achieve this automatically, we have to overcome several challenges, especially when dealing with SSL encryption, and when multiple external services use the same TCP port. Furthermore, to enable effective system testing, we need to automatically generate mocked responses with valid syntax (e.g., in JSON representing valid Data Transfer Objects (DTO) for the mocked service), regardless of whether any formal schema is present or not (e.g., the work in [15] requires specifying XSD files manually with the schema of the response messages). Furthermore,

[9] `https://mockoon.com/`
[10] `https://www.getpostman.com/`
[11] `https://wiremock.org/`

while the work in [15] was evaluated only on an artificial class, we show that our technique can scale on real systems (including an industrial API).

To the best of our knowledge, no technique exists in the scientific literature that can fully automate the process of mock generation for handing external web service dependencies. Moreover, fully automated mock generation of external web services for system level white-box testing is not a topic that has received much attention in the research literature, despite its importance in industry (e.g., considering all the existing popular libraries/tools such as WireMock and Postman).

EvoMaster is an open-source tool used for fuzzing web services using search-based techniques with white-box and black-box fuzzing [4, 11]. EvoMaster provides client-side drivers to enable white-box fuzzing and requires no code modification on the SUT, as instrumentation is done automatically. 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. Furthermore, EvoMaster features different search algorithms (e.g., MIO [3] extended with adaptive-hypermutation [22]) and fitness functions (e.g., using advanced testability transformations [9] and heuristics based on the SQL commands executed by the SUT [8]) to generate system level test suites.

3 HTTP Mocking

The objective of our work is to enable automated mocking and configuration of external web services during search-based test generation. We focus on HTTP services that use JSON for the response payloads, as those are the most common types of web services in industry [19]. To achieve this goal, a good deal of research, and technical challenges, need to be addressed, as discussed next in more details.

3.1 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* [14], relying on the infrastructure of EvoMaster to apply *method replacements* for common library methods [9] (e.g., the APIs of JDK itself). We extended the white-box instrumentation of EvoMaster with new replacement methods for APIs related to networking. When classes are loaded into the JVM, method calls towards those APIs are replaced with our methods. Those can track what inputs were used, and then call the original API without altering the semantic of the SUT.

We created 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.

```
 1 @GetMapping(path = ["/string"])
 2 fun getString() : ResponseEntity<String> {
 3
 4     val url = URL("http://hello.there:8123/api/string")
 5     val connection = url.openConnection()
 6     connection.setRequestProperty("accept", "application/json")
 7     val data = connection.getInputStream()
 8                          .bufferedReader()
 9                          .use(BufferedReader::readText)
10
11     return if (data == "\"HELLO THERE!!!\""){
12           ResponseEntity.ok("YES")
13     } else{
14           ResponseEntity.ok("NOPE")
15     }
16 }
```

Fig. 1: Small example of a REST endpoint written in SpringBoot with Kotlin, making an HTTP call towards an external service using a `URL` object.

Using the first call made to the external web service, we can capture (and alter) the DNS information about the external web service through the instrumented `java.net.InetAddress`. After the IP address is resolved for the given hostname, when the SUT tries to initiate the `Socket` connection we can capture the port information as well. If the connection is rather created through a `URL` object, such information can be derived directly there.

To make this discussion more clear, let us consider the example in Figure 1. This is a small artificial example of a REST endpoint written in SpringBoot making an HTTP call towards an external service using a `URL` object. 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’’`.

The `openConnection()` call executed at Line 5 will establish the connection with the remote destination (at the fictional `hello.there:8123`), and it will return a `java.net.URLConnection` to enable to send and receive data to such external endpoint. Our instrumentation will replace the call `url.openConnection()` with our custom `URLClassReplacement.openConnection(url)`. Inside this function, we can get all information about the HTTP connection, and create a new connection where instead of connecting to `hello.there:8123` we connect to an instance of WireMock we control.

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. For example, if the SUT is using a client library to connect to the external services, this approach will still work. An example in our case study (Section 4) is *catwatch*, where it uses the Java library `org.kohsuke:github-api`, which internally connects to `https://api.github.com`.

However, 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 hostname `localhost`. This is not a problem when dealing with external services running on internet (e.g., `https://api.github.com`). However, in microservice architectures, the external services are often running on the same host. This is, for example, an issue for the API *cwa-verification* used in our case study (Section 4). In this case, the SUT needs to be started with `localhost` replaced with any other hostname, possibly via configuration options. In the case of *cwa-verification*, this was easily achieved by manually overriding the option `--cwa-testresult-server.url`.

3.2 The Mock Server

Once we collect the information on hostnames, protocols, and ports of the external services, we have enough information to initiate mock servers (one per external service).

Writing a mock server that can handle HTTP requests is a major engineering effort. Considering the large amount of work that would be needed to write our own mock server, we rather decided to use an existing mock server library, with a wide use in industry. As the implementation of our techniques is targeting the JVM (e.g., for programs written for example in Java and Kotlin), we needed a mock server capable of running inside the JVM and be programmatically configurable. We chose to use WireMock[12] for this purpose, since it satisfies all our requirements.

WireMock can be configured manually (e.g., by writing all the reposes before staring the server) or programmatically (e.g., during runtime). Apart from that, WireMock supports record and replay as well to configure the mock server. For our purpose, we have used WireMock programmatically. The instrumentation allowed us to reroute the external web service subsequent requests to the respective WireMock server.

Each of the responses can be defined using stubs inside WireMock. A stub can be configured using a static URL path (e.g., `/api/v1/`) or a regular expression-based path (e.g., `/api/v1/.*`). Furthermore, a stub can be extended using various parameters to match a request. The default behavior of WireMock is to throw an exception if a request pattern is not configured. This caused problems during the search. To overcome the difficulties, 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 the respective WireMock server using instrumentation. After a test case is executed, we can query WireMock to retrieve the captured request patterns.

[12] `https://wiremock.org/`

When initiating WireMock servers, we faced difficulties regarding TCP ports. SUT may connect to various external web services using the same TCP port (e.g., HTTP 80 or HTTPS 443). During the search, we have control over the SUT through instrumentation. However, it was difficult to maintain the same behavior in the generated test cases (e.g., JUnit files) where no bytecode instrumentation is applied. This means generated test cases should behave the same way without any modification required, like during the search.

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 of the operating systems has a different default value for *TCP Time Wait*. This is a major issue for empirical research, when running several experiments on the same machine in parallel.

Using different addresses from the loopback subnet range (e.g.,*127.0.0.0/8*) is an easy and elegant solution to overcome the challenges we faced with the limitation of available ephemeral TCP ports to initiate WireMock servers. Moreover, 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. However, some operating systems might handle loopback addresses differently. For example, *macOS* does not allow to access addresses from this range besides *127.0.0.1* 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*). However, we have not seen this kind of problem when running experiments on *Linux* and *Windows*.

The decision to pick the loopback IP address to host the mock server happens automatically. However, in our tool extension it can be configured using various options. Three different initial local IP address allocation strategies have been developed. This helped to avoid conflicts during running experiments in parallel and to have more control over address allocation (e.g., in the generated test cases). Because of the space limitation, we cannot go into details about explaining each strategy.

Furthermore, in each setting, some of the loopback addresses will be skipped. This includes network addresses *127.0.0.0, 127.0.0.1* and the broadcast address of the range *127.255.255.255*, to avoid unanticipated side effects.

As mentioned earlier, to ensure the same behavior in the generated test cases as the search, we relied on *Java Reflection*, especially needed when hostnames are hard-coded in third-party libraries and cannot be easily modified by the user (e.g., like 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` library from Alibaba. 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`). But the use of hard-coded IP addresses does not seem a common practice.

3.3 Requests and Responses

As previously mentioned, the instantiated WireMocks will run with a default response HTTP 404 for all the request patterns. However, mocking the external web services with different correct responses is necessary to achieve better code coverage. Recall the example in Figure 1, 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 needed data at random. Performance improvement of the search requires better heuristics in a case like this.

Besides body payloads, there could be a possibility for SUT to check for other HTTP response parameters as well (e.g., HTTP status code and headers). To maximize code coverage in 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.

To address these issues, we have extended the fuzzer engine of EvoMaster to create mock responses. Besides evolving the parameters and payloads of the REST API calls towards the SUT, now our EvoMaster extension can also evolve the data in the mock responses.

Internally, each REST API call "action" in a test will have associated actions related to setup these mock. 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. In the subsequent steps of the search, when the test is selected again for mutation, these newly created actions will be mutated, and the respective WireMock server will have a new response for the request pattern besides the default `HTTP 404`. Initially, the genotype of these WireMock setup actions will contain mutable genes to handle the returned HTTP status code and an optional body payload (for example, treated as random strings). Throughout the search, actions will enhance the respective WireMock responses by mutating those genes, and then collecting their impact on the fitness function.

It is possible to easily randomize some parameters in a typical HTTP response while searching, such as HTTP status code (e.g., typically range is from 100 to 599). But, 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 body payload will lead to throwing an exception in the data parsing library of the SUT.

In case the schema is available in a commonly used format such as OpenAPI/Swagger, it is possible to infer it from that. However, it is not always possible. An alternative, more general solution is to use instrumentation to analyze how such body payloads are parsed by the SUT.

The `JSON` is a common choice as data transfer format for the interactions between web services [19]. Use of Data Transfer Objects (DTO) is a known practice in most of the statically typed programming languages 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 structure of the schema of these JSON messages.

On the JVM, the most popular libraries for JSON parsing are *Gson* and *Jackson* [17]. 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 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 DTOs for each of its different endpoints. Or, it could come from a database, or as input to the SUT, and have nothing to do with any of our WireMock instances. The solution here is to use *taint analysis*. In particular, we extend the current input tracking system in EvoMaster [10]. 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_`. In each of our method replacements (e.g., for `fromJson`) we check if the input string does match 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 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). If no matching request exists to the existing actions after a fitness evaluation, it will be disabled. The rest will be marked as active for further use. If an action representing a call to the SUT is deleted, then all the WireMock stubs associated with it are deleted as well.

4 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 question:

Table 1: Descriptive statistics of case studies. Note that, for each case study, #Endpoints represents several endpoints, #Classess represents numerous classes and #File LOCs represents numerous lines of code (LOC).

SUT	#Endpoints	#Classes	#File LOCs
catwatch	23	106	9636
cwa-verification	5	47	3955
genome-nexus	23	832	64339
ind1	54	115	7112
Total	105	1100	85042

What is the impact on code coverage and fault-finding of our novel search-based approach for external service mocking?

4.1 Experiment Setup

To evaluate our approach, we conducted our empirical study with three case studies from EMB [2] and one industrial case study (i.e., *ind1*). EMB is an open-source corpus composed of open-source web/enterprise APIs for scientific research [2].

To assess the effectiveness of our mock generation, we selected from EMB all the REST APIs which require to communicate with external web services for their business logic (i.e., *catwatch*, *cwa-verification*, and *genome-nexus*). Descriptive statistics of the selected case studies are reported in Table 1. In total, 105 endpoints and $85k$ lines of codes were employed in these experiments. Note that these statistics only concern code for implementing the business logic in these APIs. The code related to third-part libraries (which can be millions of lines [2]) is not counted here.

catwatch is a web application which allows fetching GitHub statistics for GitHub accounts, processes, and saves the data in a database and provides them via a RESTful 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 case study is a part of a microservice-based architecture which 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 which aggregates information from various other online services about genetic variants in cancer.

ind1 is a component in an e-commerce system. It deals with authentication using an external Auth0 server, and it processes monetary transactions using Stripe[13].

To assess our novel approach, we integrated it into our open-source, white-box fuzzer EvoMaster. Then we conducted experiments to compare the baseline and

[13] `https://stripe.com/`

our novel proposed approach, i.e., *Base* refers to EvoMaster with its default configuration, and *WM* refers to our approach which enables our handling of mock generation with EvoMaster. Considering the randomness 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 [1].

Interactions with real external services are often non-deterministic, as the external services might not be accessible and change their behavior at any time. 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 DoS attacks) to control the access to this API[14] based on IP address, e.g., up to 60 requests per hour for unauthenticated requests. Considering the network setup of our university, we may share the same IP address with all our colleagues and students (e.g., when behind a NAT router). Then, such rate limit configuration will strongly affect results on this study, e.g., 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 all the experiments with internet disabled. This also provides a way to evaluate our techniques in the cases in which the external services are not implemented yet (e.g., at the beginning of a new project) or are temporarily down.

With two settings (i.e., Base and WM) on four case studies using 1 hour as search budget, 30 repetitions of the experiments took 240 hours (10 days), i.e., $2 \times 4 \times 30 \times 1$. All experiments were run on the same machine, i.e., 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.

4.2 Experiment Results

To answer our research question, we report line coverage and number of detected faults on average (i.e., `mean`) with 30 repetitions achieved by Base and WM, as shown in Table 2. We employed a Mann–Whitney U test (i.e., p-value) and Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to perform comparison analyses between Base and WM in terms of line coverage and fault detection. With Mann–Whitney U test, if p-value is less than the standard significance level (i.e., 0.05), it indicates that the two compared groups (i.e., Base and WM) have statistically significant differences. Otherwise, there is no enough evidence to support the claim the difference is significant. Comparing WM with Base, the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) measures how likely WM can perform better than Base. $\hat{A}_{12} = 0.5$ represents no effect. If $\hat{A}_{12}$ surpasses 0.5, it indicates that WM has more chances to achieve better results than Base.

Based on the analysis results reported in Table 2, compared to Base, we found that WM achieved consistent improvements on both metrics, i.e., line

[14] `https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28`

Table 2: Results of line coverage and detected faults achieved by Base and WM. We also report pair comparison results between Base and WM using Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) and Mann–Whitney U test (i.e., p-value at significant level 0.05).

	Line Coverage %				Detected Faults %			
SUT	Base	WM	$\hat{A}_{12}$	p-value	Base	WM	$\hat{A}_{12}$	p-value
catwatch	46.4	53.3	**0.99**	< 0.001	41.5	45.4	**0.82**	< 0.001
cwa-verification	57.4	59.9	**0.95**	< 0.001	12.3	13.8	**0.87**	< 0.001
genome-nexus	27.9	30.3	**1.00**	< 0.001	21.2	22.0	**0.71**	< 0.001
ind1	14.2	15.1	**0.70**	0.002	59.7	65.1	**0.97**	< 0.001
Average	36.5	39.7	0.91		33.7	36.6	0.84	

coverage and detected faults. The results show that, for all the four selected APIs, our approach (i.e., WM) significantly outperformed Base with low p-values and high $\hat{A}_{12}$. Such results demonstrate effectiveness of our approach with white-box heuristic and taint analysis to guide mock object generation.

For *genome-nexus*, we found that most of the interactions from the SUT with external services is to fetch data, then save the data into a database (i.e., MongoDB). In this case, it is not necessary to extract the data when fetching them, and such data extraction can be performed when saving data into the database. Code snippet of an interaction and data extraction in *genome-nexus* is shown as below (complete code can be found in `BaseCachedExternalResourceFetcher.java`[15]).

```
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);
6
```

Based on the code, line 1 does fetch to get raw data. By checking the implementation of `fetchRawValue` (e.g., `BaseExternalResourceFetcher.java`[15]), a general type (such as `BasicDBList`[16]) is provided. With such type info, we can only know it is a list. The actual data extraction based on the raw value is performed later in Line 5 before saving it into database. Our current handling does not support such response schema extraction yet. It can be considered as a future work.

Unfortunately, for reasons of space, we cannot discuss in more details the results obtained on the other APIs.

[15] `https://github.com/EMResearch/EMB`

[16] `https://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBList.html`

***RQ**: Our approach with white-box heuristics and taint analysis demonstrates its effectiveness to guide mock object generation, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e., EvoMaster) are statistically significant on all the four selected APIs.*

5 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 [11], using WireMock for the mocked external services. Experiments on 3 open-source and 1 industrial APIs show the effectiveness of our novel techniques, both in terms of code coverage and fault detection.

To the best of our knowledge, this is the first work in the literature that provides a working solution for this important problem. 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).

Our tool extension of EvoMaster is released as open-source, with a replication package for this study currently available on GitHub[17].

Acknowledgment

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

References

1. Arcuri, A., Briand, L.: A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering **24**(3), 219--250 (2014)
2. Arcuri, A., Zhang, M., Golmohammadi, A., Belhadi, A., Galeotti, J.P., Marculescu, B., Susruthan, S.: Emb: A curated corpus of web/enterprise applications and library support for software testing research. IEEE (2023)
3. Arcuri, A.: Test suite generation with the Many Independent Objective (MIO) algorithm. Information and Software Technology **104**, 195--206 (2018)
4. Arcuri, A.: Automated black-and white-box testing of restful apis with evomaster. IEEE Software **38**(3), 72--78 (2020)
5. Arcuri, A., Fraser, G., Galeotti, J.P.: Automated unit test generation for classes with environment dependencies. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 79--90 (2014)
6. Arcuri, A., Fraser, G., Galeotti, J.P.: Generating tcp/udp network data for automated unit test generation. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 155--165 (2015)

[17] `https://github.com/researcher-for/es-mocking`

7. Arcuri, A., Fraser, G., Just, R.: Private api access and functional mocking in automated unit test generation. In: 2017 IEEE international conference on software testing, verification and validation (ICST). pp. 126--137. IEEE (2017)
8. Arcuri, A., Galeotti, J.P.: Handling sql databases in automated system test generation **29**(4), 1--31 (2020)
9. Arcuri, A., Galeotti, J.P.: Enhancing Search-based Testing with Testability Transformations for Existing APIs. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1--34 (2021)
10. Arcuri, A., Galeotti, J.P.: Enhancing search-based testing with testability transformations for existing apis. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1--34 (2021)
11. Arcuri, A., Galeotti, J.P., Marculescu, B., Zhang, M.: Evomaster: A search-based system test generation tool. Journal of Open Source Software **6**(57), 2153 (2021)
12. Fraser, G., Arcuri, A.: 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. pp. 416--419 (2011)
13. Golmohammadi, A., Zhang, M., Arcuri, A.: Testing restful apis: A survey (2023)
14. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering **30**(1), 3--16 (2004)
15. Havrikov, N., Gambi, A., Zeller, A., Arcuri, A., Galeotti, J.P.: Generating unit tests with structured system interactions. In: 2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST). pp. 30--33. IEEE (2017)
16. Kim, M., Xin, Q., Sinha, S., Orso, A.: 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. p. 289–301. ISSTA 2022, Association for Computing Machinery, New York, NY, USA (2022). `https://doi.org/10.1145/3533767.3534401`, `https://doi.org/10.1145/3533767.3534401`
17. Maeda, K.: Performance evaluation of object serialization libraries in xml, json and binary formats. In: 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP). pp. 177--182. IEEE (2012)
18. Marri, M.R., Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: An empirical study of testing file-system-dependent software with mock objects. In: Automation of Software Test, 2009. AST'09. ICSE Workshop on. pp. 149--153 (2009)
19. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public rest web service apis. IEEE Transactions on Services Computing (2018)
20. Newman, S.: Building microservices. " O'Reilly Media, Inc." (2021)
21. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: To mock or not to mock? an empirical study on mocking practices. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 402--412. IEEE (2017)
22. Zhang, M., Arcuri, A.: Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1) (2021)
23. Zhang, M., Arcuri, A.: Open problems in fuzzing restful apis: A comparison of tools (may 2023). `https://doi.org/10.1145/3597205`, `https://doi.org/10.1145/3597205`, just Accepted