Multi-Phase Taint Analysis for JSON Inference in Search-Based Fuzzing

Susruthan Seran* Onur Duman*, Andrea Arcuri*†,
*Kristiania University of Applied Sciences, Oslo, Norway
†Oslo Metropolitan University, Oslo, Norway

***Abstract*—As software applications grow increasingly complex, particularly in their input formats, testing these applications becomes a challenging endeavour. Automated testing techniques, such as search-based white-box fuzzing, have shown promise in addressing these challenges. However, generating well-formed inputs for fuzzing remains a significant obstacle. In this paper, we present novel techniques as an academic proof-of-concept for automatically inferring JSON-based schemas to enhance search-based white-box fuzzing, focusing on Java and Kotlin applications. Our work offers an alternative approach to black-box grammar-based fuzzing.**

***Index Terms*—SBST, fuzzing, JSON, Taint, REST API**

I. Introduction

Software applications are becoming more and more complex in terms of both their functionality and input formats (e.g., configuration files). Apart from the complex input formats, modern-day web and mobile applications rely on other external web services for several purposes such as payments (e.g., Stripe[1]) and authentication (e.g., OAuth[2]). Typically, this interconnectivity is achieved through web services, and to facilitate such functionalities, the use of third-party Software Development Kits (SDKs) is a widely observed practice in software applications. Due to the versatile nature of the Representational State Transfer (REST) architectural style, the JavaScript Object Notation (JSON) has become the de-facto standard for communication in these web services [1].

Automated software testing techniques (e.g., search-based white-box fuzzing) have proven to be effective approaches to efficiently test such complex applications [2], [3]. However, the generation of well-formed inputs (e.g., based on JSON schemas) for search-based white-box fuzzing is one of several challenges in achieving fully automated testing.

Data Transfer Object (DTO) is a design pattern used in most statically typed programming languages (e.g., Java, Kotlin, C++, Go) to represent data, particularly when transferring data over networks or between processes. Typically, DTOs are simple classes that encapsulate data. The usage of such complex input structures is not only common for data transfers in web and mobile applications. For example, there are millions of programs that use *YAML Ain't Markup Language* (YAML) (also known as *Yet Another Markup Language*) configuration files, with Docker[3] being one such example.

User-specified grammars [4] deal with complex inputs for fuzzing. However, using user-specified grammars may not always be feasible. For example, if the System Under Test (SUT) uses an SDK from a vendor, user-specified grammars may not be practical unless the documentation is provided for the SDK. Moreover, to ensure fully automated testing, it is vital to reduce such manual user intervention to near zero.

In this paper, we present novel techniques to infer JSON-based schemas in a fully automated manner for search-based, white-box fuzzing. We developed our techniques as an academic proof-of-concept. Therefore, we will focus on Java and other programming languages that compile into JVM bytecode (e.g., Kotlin), as Java is one of the most widely used programming languages in industry and academia. However, our techniques can be adapted to other statically typed programming languages (e.g., C++, C#, and Go).

In white-box testing, the source code of the SUT needs to be analyzed. The capabilities of the Java Virtual Machine (JVM) allow us to perform this analysis effectively. In particular, Java Reflection and bytecode instrumentation are useful for implementing our techniques. The most popular libraries for JSON parsing are *Gson*[4] and *Jackson*[5] for JVM-based applications [5]. Additionally, our adapted real-world examples are developed using both libraries. As a result, we developed our techniques supporting both libraries.

Existing white-box techniques can handle parsing of JSON data when it is marshaled into DTOs [6], [7]. However, these techniques fail when dealing with parsing container data structures (e.g., maps and lists) when their content-type is not known at the time of parsing. In this paper, we provide an automated solution to this problem that is based on a multi-phase taint analysis.

We implemented our techniques as extensions to the open-source white-box fuzzer for Web APIs named EvoMaster,[6] rather than implementing them from scratch, since developing a search-based white-box fuzzer from scratch can be a huge engineering effort. Based on the existing literature, EvoMaster performs best compared to other fuzzers for Web APIs [8]–[10].

The rest of this article is organized as follows. First, Section II provides background information and discusses the

[1] https://stripe.com/
[2] https://oauth.net
[3] https://www.docker.com/
[4] https://github.com/google/gson
[5] https://github.com/FasterXML/jackson
[6] https://github.com/WebFuzzing/EvoMaster/tree/master

```java
1  public Map<String, String> convertToMap(String token) {
2      Gson gson = new Gson();
3      Map<String, String> tokenMap = new HashMap<String, String>();
4      try {
5          tokenMap = gson.fromJson(token,
6                                   Map.class);
7      }
8      catch (JsonParseException e) {
9          System.out.println("The format of token is invalid. For example {\"source1\":\"put-your-token1-here\",\"source2\":\"put-your-token2-here\"}");
10     }
11     return tokenMap;
12 }
13
```

Fig. 1. Code block from *genome-nexus* responsible for converting *String* into a *Map* of variants.

motivation behind this work. Second, Section III presents a literature review. Third, Section IV details our novel techniques. Fourth, Section V discusses our empirical study to demonstrate the effectiveness of our techniques. After that, Section VI addresses threats to the validity of our work. Finally, Section VII concludes the paper and discusses potential future work.

II. Background

In automated testing, there can be cases where inputs are complex and structured rather than simple primitive values (e.g., integer, string, double). To improve the efficiency of testing in terms of code coverage and fault-finding, generating valid inputs is crucial. A study conducted on automated mock generation of external web services observed the impact of not having valid inputs for complex data structures [11]. One way to tackle the challenge of considering such inputs is through grammar-based fuzzing [12]. However, this is not always practical because it may not be always possible to find documentation for third-party SDKs. In addition, it may not always be a scalable approach as the application grows in size (i.e., lines of code). Moreover, generating well-formed inputs in a fully automated manner for testing has not received enough attention in the literature.

To illustrate this issue, we use the code block in Figure 1 as a motivating example. Figure 1 is taken from one of our SUTs used in our empirical study, which will be presented in Section V. In Line 5, the given string input (i.e., the variable `token`) representing a JSON object should be unmarshalled without any failures (i.e., no exceptions are thrown) for the SUT to move further in the execution.

In this example, the SUT is expecting a map of strings as keys and values. Moreover, there can be cases in which the input may contain data structures that are more complex than strings, such as objects or arrays. From our preliminary investigation of the selected case studies, we found more interesting and complex cases of JSON unmarshalling. Additionally, we chose to focus solely on JSON, as it is one of the most common formats for data exchange on the web [13]. Furthermore, compared with grammar-based fuzzing, by focusing solely on JSON, we can perform bytecode analyses that are more efficient and precise than trying to infer input grammars in a black-box or gray-box manner.

III. Related Work

A. Search-based White-Box Fuzzing

Fuzzing is the process of providing random or invalid inputs to software in order to analyze its behavior [14]. Fuzzing is commonly used in the industry to uncover tens of thousands of bugs in large and interconnected enterprise software systems with millions of lines of code [2], [15]. In addition to its usage in industry, fuzzing has received attention in academic research [16], [17]. Fuzzing methods can be categorized as black-box or white-box. In the black-box fuzzing, the tester does not have access to the source code of the application being fuzzed. On the contrary, in white-box fuzzing, the tester has full access to the source code of the application being fuzzed. According to the existing literature, white-box fuzzing is an effective software testing technique [3], [14], [16], [18]–[20].

Recently, the use of several fuzzing methods, such as white-box and black-box, to test REST-based web services has gained attention [8], [11], [17]. One common usage of white-box fuzzing is search-based test generation [2], [7], [8], [11], [20]. In search-based test generation, several heuristics can be gathered through white-box fuzzing to improve search performance in terms of code coverage and fault detection rates [7]. Specifically, in an existing study, heuristics about external web services played a key role in improving the quality of test cases generated through automated mocking for search-based test generation [11]. However, the existing literature falls short in terms of addressing the challenges related to heuristics for JSON-based schemas.

B. Representational State Transfer (REST) and JavaScript Object Notation (JSON)

REST Application Programming Interfaces (APIs) have gradually become the primary means of communication between various services, including microservices and software-as-a-service solutions [1], [21]. REST is a simple and scalable architectural style that uses stateless connections, and this nature makes it suitable for a wide range of use cases. Based on the existing literature, besides its popularity in industry, REST APIs are one of the popular topics in academia as well [8], [17], [22], [23]. In general, REST APIs use JSON or Extensible Markup Language (XML) as the two common formats to exchange data [13], [20], [24]. JSON is a lightweight data interchange format that supports complex data structures (e.g., arrays and objects) in a key-value pair format and serves as an alternative to XML [25]. In addition to its usage in industry, JSON has received attention in academic research (e.g., [5], [7], [26], [27]).

Additionally, schema specification standards such as OpenAPI [28] are used to create a standardized interface between

REST APIs. Such standardization of schemas makes fully automated testing of RESTful APIs feasible [17], [20]. However, for automated search-based fuzzing, it is crucial to infer all schemas used in the SUT to achieve better code coverage. Inferring these complex JSON-based schemas in an automated manner seems to be an area that has received less attention in the existing literature.

C. Grammars In Fuzzing

Compared with traditional random input-based fuzzing, well-formed inputs in grammar-based fuzzing that use formal grammar specifications enhance the efficiency of the testing process [12], [26]. A combination of search-based testing with grammar-based fuzzing improves the branch coverage for JSON-related classes significantly [29]. In general, such grammars can be user-specified, inferred from documentation (e.g., OpenAPI), or generated using grammar-mining techniques [30]. However, for fully automated and more efficient testing, better grammar-inference techniques are needed.

D. Taint analysis

Taint analysis is a program analysis technique that tracks data flow within a system, enabling the detection of software faults, especially software vulnerabilities. Taint analysis has been used in various previous studies. Notably, the use of dynamic taint analysis can help to identify security faults with zero-code modifications to the SUT [31]. A study on the detection of sensitive data leakage further demonstrated the viability of using taint analysis across different programming languages [32]. A study on the use of taint analysis for advanced white-box heuristics for search-based fuzzing of REST APIs showed significant improvements in testing metrics such as code coverage and fault detection [7]. However, to the best of our knowledge, heuristics for complex, untyped data structure collections (i.e., *Map*, *List*) in search-based testing using dynamic taint analysis have not been presented yet in the existing literature.

IV. Multi-Phase Taint Analysis

```
 1 @PostMapping(consumes = MediaType.
       APPLICATION_JSON_VALUE)
 2 public ResponseEntity<String> post(@RequestBody
       String json) throws JsonProcessingException
       {
 3
 4     ObjectMapper mapper = new ObjectMapper();
 5     Map collection = mapper.readValue(
 6             json, Map.class
 7     );
 8     List<Integer> z = (List<Integer>) collection
 9             .get("z");
10
11     if (z.get(1) == 2025) {
12         return ResponseEntity.ok().body("OK");
13     } else {
14         return ResponseEntity.badRequest()
15                 .body("FAIL");
16     }
17 }
```

Fig. 2. Code block of a REST endpoint from one of our artificial case studies.

Consider the artificial example in Figure 2. Here, an input string is parsed as a JSON element with the Jackson library in Line 5. This JSON string is first marshaled as a `Map`. From this `Map`, the entry `z` is read as a list of integers. On this list, the second element is read and checked if it is equal to 2025. If so, the constant `"OK"` is returned; otherwise, `"FAIL"` is returned. Existing fuzzers (e.g., EvoMaster) would fall short in covering the branch returning `"OK"` since their fitness functions do not provide a gradient (i.e., guidance towards the optimal solution) to the search. By taking this into account, in this paper, we aim to tackle this challenge.

In order to tackle the challenge of covering such cases in testing, given the advanced white-box heuristics based on taint analysis presented in [7], [33], EvoMaster can send tainted values such as `"_EM_k_XYZ_"` (where k is a positive integer, e.g., $k = 42$). Several common APIs are automatically instrumented when the bytecode is first loaded into the JVM of the running SUT [33], such as `readValue()` in *Jackson* [7]. These instrumented calls are semantically equivalent to the original calls (i.e., the SUT must behave the same when it is instrumented) but can analyze if any input is a tainted string. If so, the instrumentation can inform back the search-engine that the input string with the value `"_EM_42_XYZ_"` was attempted to be marshaled into a map object. The next mutation operation on this test case can then replace `"_EM_42_XYZ_"` with a valid map representation, such as {}.

This approach works [7], especially well when the marshaled object is a POJO (Plain Old Java Object) or DTO, i.e., a Java class with named fields matching a JSON object representation. For example, consider a JSON object with two fields: `x` (numeric) and `y` (a string), e.g., {`"x":123,"y":"foo"`}. This could be marshaled into a Java class instance having two fields called `x` (of type, for example `double`) and `y` (of type `String`). However, the same JSON object could also be marshaled into a `Map<String,Object>`, where the values `"x"` and `"y"` are string keys in such a map. This is not an unusual case; rather, it is common when there is no POJO/DTO representation for the parsed JSON objects. This can occur, for example, when making a call to an external web service, and the user is interested in only specific fields of the response. Using a `Map` data structure and extracting the needed fields from it may be simpler than implementing a DTO for the entire response. Unfortunately, in this case, when calling `readValue()`, there would be no information on what fields (and their type) would be needed for the remainder of the computation. In other words, by using the techniques in [7], EvoMaster can send string inputs in the form of `"{}"`, but it has no gradient in the search to determine that a field called `z` is needed in the map.

To tackle this first issue, we extend the search-engine of EvoMaster to support the concept of `TaintedMapGene`. When a tainted value `"_EM_k_XYZ_"` is used as input to a marshaling operation for a `Map` in Jackson or Gson,

```
1   L2
2    LINENUMBER 24 L2
3    ALOAD 3
4    LDC "z"
5    INVOKEINTERFACE java/util/Map.get (Ljava/lang
     /Object;)Ljava/lang/Object; (itf)
6    CHECKCAST java/util/List
7    ASTORE 4
8   L3
9    LINENUMBER 26 L3
10   ALOAD 4
11   ICONST_1
12   INVOKEINTERFACE java/util/List.get (I)Ljava/
     lang/Object; (itf)
13   CHECKCAST java/lang/Integer
14   INVOKEVIRTUAL java/lang/Integer.intValue ()I
15   SIPUSH 2025
16   IF_ICMPNE L4
```

Fig. 3. This is a part of the generated bytecode for the example in Figure 2.

the search process is informed that the entity holding the tainted value (e.g., a body payload, a query parameter, or field in an object) should be treated as a `TaintedMapGene`. A tainted map will be a immutable value in the form of {`"EM_tainted_map":"_EM_j_XYZ_"`}, for some $j \neq k$, as all tainted values during the search should be unique. Such a map has a specific field, with a tainted string as value. This is a valid map object that would not cause a function such as `mapper.readValue(json, Map.class)` to throw any exceptions. Giving such a map representation as input, we keep track of all key access invocations on the instantiated map (e.g., the variable called `collection` in Figure 2). This was already implemented in EvoMaster as part of the advanced branch distance computations for Java collections [7], [33]. For example, every time an instrumented version of `Map.get(keyName)` is called, we check within that instrumented call to see if the map contains any key named `"EM_tainted_map"`.

If so, then we are dealing with a tainted map. Then, the search-engine can be informed that a specific key with value `keyName` (e.g., `"z"`) was accessed on a tainted map with id `"_EM_j_XYZ_"`. With such information, at the next mutation operation, the tainted map will get extended with a new key entry with the name `"z"`.

The second issue is that, although we can infer that the key `"z"` was accessed, we do not know at the time of calling `collection.get("z")` what is the expected type of `"z"`, as `Map.get(keyName)` returns an instance of the root class `java.lang.Object`. The information about `"z"`, which is expected to be a `List<Integer>` is only obtainable *after* `Map.get(keyName)` is executed. Figure 3 shows a snippet of bytecode for a couple of lines of the example in Figure 2. After the `Map.get` is called in Line 5 in Figure 3, the casting is executed with the bytecode command `CHECKCAST java/util/List`. To discover the information about `"z"`, our solution requires two steps. First, when a new key access (e.g., `"z"`) on a tainted map is discovered, the new entry added to the map by the mutation operation will be of type *String*, with a tainted value. For example, we would end up with something like this: {`"EM_tainted_map":"_EM_j_XYZ_", "z":"_EM_t_XYZ_"`}, for $t \neq j \neq k$ (i.e., all tainted values must be unique). Second, we instrument each call to `CHECKCAST` by adding a probe function before it, in the form of `Object executingCheckCast(Object value, String classType)`. This probe takes as an input the element that is going to be cast and the information on the target cast (`java/util/List` in our example). Such input will be analyzed, and the output will be the input element without modifications (as the probe must not alter the semantics of the function). In this probe, if the input element is a tainted *String* (e.g., `"_EM_t_XYZ_"`), then the search-engine can be informed that such tainted string was tried to be cast into a `List`. At the next mutation operation in the evolutionary process, the entry `"z"` in the tainted map would be replaced with a valid list, e.g., `[]`.

Unfortunately, this is not sufficient, as there is a subtle side effect in the evolutionary process. Before applying our extended taint analysis, an input such as {`"EM_tainted_map":"_EM_j_XYZ_"`} would cause the execution to proceed all the way to `z.get(1)`, throwing a null pointer exception there, as the variable `z` is `null`. A `null` value is a valid `List<Integer>`. On the other hand, an evolved input like {`"EM_tainted_map":"_EM_j_XYZ_", "z":"_EM_t_XYZ_"`} would result in a cast exception being thrown earlier, as a `String` is not a valid `List<Integer>`. This leads to less covered code in the SUT, which in turn results in a worse fitness value. As such, the new evolving test case would die out.

To tackle this third issue, we introduced a new testing target type in EvoMaster. In particular, in addition to maximizing line and branch coverage (as well as other advanced code-level metrics [7], [33]), we also aim to successfully cover each `CHECKCAST` bytecode instruction with a non-null input. Each `CHECKCAST` instruction will have a fitness score in $[0, 1]$, similar to any other testing target tracked in EvoMaster. The value 0 means that the bytecode instruction was never executed, whereas a 1 means there was at least one non-null input for which the `CHECKCAST` did not throw an exception. Intermediate values are intended to provide gradients for the search. Specifically, a `null` value will be given a non-zero score (e.g., 0.1), while a tainted *String* will get a higher value (e.g., 0.5). This can be computed in the probes added before the `CHECKCAST` instructions. This way, the search has a gradient to reward test cases that are learning the types of accessed fields in the map.

The fourth issue to tackle is that the list itself might be of any type. To handle this case, similar to maps, we introduce the concept of `TaintedArrayGene` in EvoMaster. When we determine that a tainted string is cast into a list or a list is marshaled from a JSON entry, we create a string array with some tainted values. This could be something like `["_EM_a_XYZ_","_EM_a_XYZ_","_EM_a_XYZ_"]`, for $a \neq t \neq j \neq k$. When an element is read from

```java
@Test @Timeout(60)
public void test_6() {

  given().accept("*/*")
    .header("x-EMextraHeader123", "")
    .contentType("application/json")
    .body(" { " +
            " \"EM_tainted_map\": \"_EM_1897_XYZ_\", " +
            " \"z\": [ " +
            " 268436093, " +
            " 2025, " +
            " -2176 " +
            " ] " +
            " } ")
    .post(baseUrlOfSut+"/api/json")
    .then()
    .statusCode(200)
    .assertThat()
    .contentType("text/plain")
    .body(containsString("OK"));
}
```

Fig. 4. A generated test for the example mentioned in Figure 2.

```java
1  private List<LocationIQResponse> CallLocationIQAPI(String search) {
2    ArrayList<LocationIQResponse> locationsResponse = new ArrayList<>();
3
4    try {
5      locationsResponse = new ObjectMapper()
6        .readValue(
7            search,
8            new TypeReference<ArrayList<LocationIQResponse>>() {}
9        );
10
11   } catch (IOException e) {
12     LOGGER.error("getLocationIQResponse - IOException - Error with message: {}", e.getMessage());
13   }
14
15   return locationsResponse;
16 }
17
```

Fig. 5. The code block is part of the application logic obtained from the *gestaohospital-rest*.

such a list, a cast may take place (see `CHECKCAST java/lang/Integer` in Figure 3). In exactly the same way as discussed before for maps, the search-engine can be informed via taint analysis that the elements extracted from this list are cast into integers. At the next mutation operation, the string array `["_EM_a_XYZ_","_EM_a_XYZ_","_EM_a_XYZ_"]` will be mutated into an integer array of random values, e.g., `[42,-5,134234]`.

The last step is to evolve the second element in the array (e.g., -5) into the target value of 2025. To achieve this, we do not need anything novel since we can simply rely on the existing instrumentation for branch-jump instructions (e.g., `IF_ICMPNE`) in EvoMaster, using the traditional branch distance on numerical constraints [34].

By using our novel techniques presented in this paper, it is quite easy for our extended version of EvoMaster to evolve a test case such as the one presented in Figure 4, where the `"OK"` branch is covered. Our approach can handle all the different cases in which basic maps and lists/sets/arrays are marshaled from strings representing JSON data. Our approach is multi-phase, as each new fitness evaluation uncovers additional information. Then, the discovery of this new information is rewarded in the fitness function, allowing the evolutionary process to continue.

One benefit of this approach is that it is lightweight, as it does not require complex static code analysis. Since it relies on actual input values tracked dynamically through test execution, the control flow of the code has no direct impact on our techniques.

V. Empirical Study

To evaluate the novel techniques presented in this paper, we conducted an empirical study to answer the following research question:

RQ: Are our novel techniques capable of automatically inferring JSON-based schemas for case studies adapted from real-world complex examples?

We developed case studies with various business logic adapted from five real-world open-source applications from the EMB corpus [35]. The EMB repository consists of open-source Web APIs, which have been used by different research groups in various studies [8], [36]–[38]. Originally, the adapted examples from the case studies dealt with external web services, database interactions, and messaging brokers. For our experiment purposes, we removed the other factors (i.e., external web service connection and database interactions) and modified the code while preserving the actual logic.

Due to space limitations, we are unable to discuss the case studies we utilized in detail. Therefore, we will provide key details of a few of our case studies in the following steps. At the higher level, all the other case studies have similar elements except the logic behind the JSON unmarshalling.

Figure 5 contains the code block adapted from *gestaohospital-rest*. The original code for a service that is responsible for fetching location information from an external web service and converting it into an *ArrayList* of `LocationIQResponse` is shown in Line 6.

To replicate the same behavior as the original application, we created a REST endpoint. Once the conditions are met in Line 9 in Figure 6, the endpoint will respond with HTTP 418 and the body `"Tea"` (Line 11); otherwise, it will respond with HTTP 204 and the body `"No tea for you!"`.

Figure 7 contains a test from the generated test suite. In Line 7, we can see a successfully generated array of `LocationIQResponse` that allows the application to reach the line of code returning HTTP 418.

Similar to the previous example, Figure 8 contains the code

```java
@PostMapping(path = "/json", consumes = MediaType
    .APPLICATION_JSON, produces = MediaType.
    APPLICATION_JSON)
public ResponseEntity<String> parseJson(
    @RequestBody String json) {
  LocationIQService service = new
    LocationIQService();

  List<LocationIQResponse> responses = service
        .getLocationIQResponse(json);

  if (responses.get(2).getPlaceId()
        .equals("teashop")) {
      return ResponseEntity.status(418)
        .body("Tea");
  }

  return ResponseEntity.status(204)
    .body("No tea for you!");
}

```

Fig. 6. The REST endpoint developed as part of *gestaohospital-rest* case study adaptation.

```kotlin
@Test @Timeout(60)
fun test_5()  {

    given().accept("application/json")
      .header("x-EMextraHeader123", "")
      .contentType("application/json")
      .body(" [ " +
          " { " +
          " \"placeId\": \"CsUUUU7e8Lpl\" " +
          " }, " +
          " { " +
          " \"placeId\": \"_EM_32_XYZ_\" " +
          " }, " +
          " { " +
          " \"placeId\": \"teashop\" " +
          " } " +
          " ] ")
      .post("${baseUrlOfSut}/api/json?
    EMextraParam123=_EM_21_XYZ_")
      .then()
      .statusCode(418)
      .assertThat()
      .contentType("application/json")
      .body(containsString("Tea"))
}

```

Fig. 7. Automatically generated test for *gestaohospital-rest* case study.

obtained from one of the case studies, *proxyprint*.[7] This code is responsible for calculating the budget for already registered print requests. Due to space limitations, we modified and redacted several lines from the original code while preserving the actual logic. The application takes several identifiers of print shops and then calculates the budget based on the data obtained from the database about the shops.

Figure 9 contains the code block responsible for parsing JSON inside the `ResultExtender` in the SUT, *language-*

```java
public Map<Long, String>
    calcBudgetForPrintRequest(String requestJSON
    ) throws IOException {
  PrintRequest printRequest = new PrintRequest();

  List<Long> pshopIDs = null;
  Map prequest = new Gson().fromJson(
    requestJSON, Map.class
  );

  // PrintShops
  List<Double> tmpPshopIDs = (List<Double>)
    prequest.get("printshops");
  pshopIDs = new ArrayList<>();
  for (double doubleID : tmpPshopIDs) {
    pshopIDs.add((long) Double.valueOf(
      (double) doubleID).intValue()
    );
  }

  // Finally, calculate the budgets :D
  List<PrintShop> pshops = getListOfPrintShops(
    pshopIDs
  );
  Map<Long, String> budgets = printRequest.
    calcBudgetsForPrintShops(pshops);

  return budgets;
}
```

Fig. 8. The application logic obtained from the *proxyprint* case study.

```java
 1 private List<RemoteRuleMatch> parseJson(String
      inputStream) throws IOException {
 2   Map map = mapper.readValue(inputStream, Map.
      class);
 3   List matches = (ArrayList) map.get("matches");
 4   List<RemoteRuleMatch> result = new ArrayList
      <>();
 5   for (Object match : matches) {
 6     RemoteRuleMatch remoteMatch = getMatch(
 7         (Map<String, Object>) match
 8     );
 9     result.add(remoteMatch);
10   }
11   return result;
12 }
```

Fig. 9. The code block demonstrates the application logic obtained from the *languagetool*.

tool.[8] This component is responsible for extending the results by adding rule matches from another remote server that contains language rules. Due to this, the original code has an external web service connection. For our purpose, we modified the method argument type from `InputStream` to `String`. In Line 2, the code unmarshals the string to `Map`. In further steps, obtained data will be extracted through `getMatch()` as in Line 8 and will be added to a new `ArrayList`. Finally, the new `ArrayList` with additional rules will be returned at the end.

In our adaptation, as shown in Figure 10, we get the second element from the list and validate that the message string is

[7]https://github.com/ProxyPrint/proxyprint-kitchen

[8]https://github.com/languagetool-org/languagetool

```
@PostMapping(path = "/json", consumes = MediaType
    .APPLICATION_JSON, produces = MediaType.
    APPLICATION_JSON)
public ResponseEntity<String> parseJson(
    @RequestBody String json) {
  try {
    ResultExtender resultExtender = new
    ResultExtender();

    List<RemoteRuleMatch> rules = resultExtender
        .getExtensionMatches(json);

    if (rules.get(2).getMessage()
                .equals("vowels")) {
        return ResponseEntity.ok("vowels");
    }

    return ResponseEntity.status(204)
                .body("Nothing found");
  } catch (IOException e) {
    return ResponseEntity.status(500).build();
  }
}
```

Fig. 10. The endpoint developed to adapt the example from the *languagetool* case study.

```
 1 @Test @Timeout(60)
 2 fun test_5()  {
 3
 4   given().accept("application/json")
 5   .header("x-EMextraHeader123", "")
 6   .contentType("application/json")
 7   .body(" { " +
 8     " \"EM_tainted_map\": \"_EM_38_XYZ_\", " +
 9     " \"printshops\": [ " +
10     " 3.833483191701392 " +
11     " ] " +
12     " } ")
13   .post("${baseUrlOfSut}/api/json")
14   .then()
15   .statusCode(200)
16   .assertThat()
17   .contentType("application/json")
18   .body(containsString("Printing"))
19 }
```

Fig. 11. A sample of an automatically generated test case for the *proxyprint* case study.

"vowels". If that's the case, we respond with HTTP 200 and the body "vowels". Otherwise, we respond with HTTP 204 and the body "Nothing found".

Our novel technique automatically produces a usable test suite for all these five examples derived from the EMB repository at the end of the search (i.e., for the endpoints presented in Figure 1, 2, 6, 8 and 10). Figure 11 contains one test from the generated suite for the *proxyprint* case study. Line 7 contains the automatically generated JSON-based schema required for the endpoint to respond with HTTP status code 200. Therefore, we are able to automatically generate an array of "printshops" identifiers of the Long data type. By taking all of those into account, we can answer the research question as follows.

***RQ**: Our approach, which uses white-box heuristics and taint analysis, demonstrates the capability of inferring JSON-based schemas for examples taken from real-world complex APIs.*

VI. Threats to Validity

Internal validity. Our technique is implemented only for JVM-based applications. There could be other necessary steps required in other programming languages to make our techniques generic. Additionally, our implementation is an extension of EvoMaster. These factors may pose threats to internal validity due to potential faults in our implemented software and the presence of uncovered cases. Furthermore, we cannot ensure that our implementation is free of faults or that our novel techniques can be directly applied to other programming languages. Our extension is open-source, with the link removed due to double-blind reviews.

External validity. Our experiments were conducted on five artificial REST APIs. The five APIs are based and adapted from all the APIs in EMB [35] that have JSON-based schemas handling. Although we managed to cover several generic scenarios, there could be more edge cases that are not covered under our constructed APIs. Therefore, there is a need to investigate more case studies in future work. Additionally, there is a need to conduct more experiments to study the impact of our approach on testing metrics such as code coverage and fault detection rate on whole, real-world APIs.

VII. Conclusion

In this paper, we have provided techniques to infer JSON-based schemas for automated search-based white-box fuzzing. Our techniques have been implemented as an extension to the fuzzer EvoMaster [10]. Experiments on five examples based open-source APIs demonstrate the viability of our novel techniques.

To the best of our knowledge, this is the first work in the literature to offer a working solution for tackling this important problem. Therefore, there are several avenues for further enhancement of our techniques in future work. For example, libraries like Jackson[5] use a tree-based data structures (e.g., JsonNodes) to represent JSON documents. Those tree-based data structures appear in few APIs in EMB [35]. Our novel techniques do not support such custom representations at the moment.

Acknowledgment

This work is supported 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] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.
[2] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-box fuzzing rpc-based apis with evomaster: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.

[3] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[4] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.

[5] K. Maeda, "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)*. IEEE, 2012, pp. 177–182.

[6] A. Arcuri and J. P. Galeotti, "Enhancing Search-based Testing with Testability Transformations for Existing APIs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–34, 2021.

[7] A. Arcuri, M. Zhang, and J. P. Galeotti, "Advanced white-box heuristics for search-based fuzzing of rest apis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2024. [Online]. Available: https://doi.org/10.1145/3652157

[8] M. Kim, Q. Xin, S. Sinha, and A. Orso, "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*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: https://doi.org/10.1145/3533767.3534401

[9] M. Zhang and A. Arcuri, "Open problems in fuzzing restful apis: A comparison of tools," 2023. [Online]. Available: https://doi.org/10.1145/3597205

[10] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, "Evomaster: A search-based system test generation tool," *Journal of Open Source Software*, vol. 6, no. 57, p. 2153, 2021.

[11] S. Seran, M. Zhang, and A. Arcuri, "Search-based mock generation of external web service interactions," in *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2023.

[12] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary grammar-based fuzzing," in *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 2020, pp. 105–120.

[13] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public rest web service apis," *IEEE Transactions on Services Computing*, 2018.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[15] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "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*, 2021, pp. 1393–1403.

[16] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, sep 2022. [Online]. Available: https://doi.org/10.1145/3512345

[17] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing restful apis: A survey," *ACM Transactions on Software Engineering and Methodology*, aug 2023. [Online]. Available: https://doi.org/10.1145/3617175

[18] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[19] A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortés, "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, 2021, pp. 231–241.

[20] A. Arcuri, "Automated black-and white-box testing of restful apis with evomaster," *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.

[21] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[22] C.-H. Tsai, S.-C. Tsai, and S.-K. Huang, "Rest api fuzzing by coverage level guided blackbox testing," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 291–300.

[23] O. Banias, D. Florea, R. Gyalai, and D.-I. Curiac, "Automated specification-based testing of rest apis," *Sensors*, vol. 21, no. 16, p. 5375, 2021.

[24] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "Rest apis: a large-scale analysis of compliance with principles and best practices," in *International Conference on Web Engineering*. Springer, 2016, pp. 21–39.

[25] "Json schema specification wright draft 00," https://datatracker.ietf.org/doc/html/draft-wright-json-schema-00.

[26] L. Veldkamp, M. Olsthoorn, and A. Panichella, "Grammar-based evolutionary fuzzing for json-rpc apis," in *The 16th International Workshop on Search-Based and Fuzz Testing*. IEEE/ACM, 2023.

[27] I. Ballesteros, L. E. B. de Barrio, L.-A. Fredlund, and J. Marino, "Tool demonstration: Testing json web services using jsongen," *biblioteca.sistedes.es*, 2018.

[28] "Open api specification," https://swagger.io/specification/.

[29] M. Olsthoorn, A. van Deursen, and A. Panichella, "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*, 2020, pp. 1224–1228.

[30] R. Gopinath, B. Mathis, and A. Zeller, "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*, 2020, pp. 172–183.

[31] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[32] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 932–944.

[33] A. Arcuri and J. P. Galeotti, "Enhancing search-based testing with testability transformations for existing apis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–34, 2021.

[34] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[35] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, and S. Seran, "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, 2023, pp. 433–442.

[36] O. Sahin and B. Akay, "A discrete dynamic artificial bee colony with hyper-scout for restful web service api test suite generation," *Applied Soft Computing*, vol. 104, p. 107246, 2021.

[37] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Improving test case generation for rest apis through hierarchical clustering," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 117–128.

[38] M. Zhang, B. Marculescu, and A. Arcuri, "Resource and dependency based test case generation for restful web services," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–61, 2021.