IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received January 12, 2021, accepted January 29, 2021, date of publication February 2, 2021, date of current version February 12, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3056505

A Black Box Tool for Robustness Testing of REST Services

NUNO LARANJEIRO[1], (Member, IEEE), JOÃO AGNELO[1], AND JORGE BERNARDINO[1,2], (Member, IEEE)

[1]University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, 3030-290 Coimbra, Portugal
[2]Polytechnic of Coimbra, Coimbra Institute of Engineering (ISEC), 3030-199 Coimbra, Portugal

Corresponding author: Nuno Laranjeiro (cnl@dei.uc.pt)

This work has been supported by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020; has been supported by the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 823788 (project ADVANCE); by the project MobiWise: From mobile sensing to mobility advising (P2020SAICTPAC/0011/2015), co-financed by COMPETE 2020, Portugal 2020 - Operational Program for Competitiveness and Internationalization (POCI), European Union's ERDF (European Regional Development Fund), and the Portuguese Foundation for Science and Technology (FCT); by the European Regional Development Fund (FEDER), through the Regional Operational Programme of Lisbon (POR LISBOA 2020) and the Competitiveness and Internationalization Operational Programme (COMPETE 2020) of the Portugal 2020 framework [Project 5G with Nr. 024539 (POCI-01-0247-FEDER-024539)]; and by the TalkConnect project ''Voice Architecture over Distributed Network'' (reference: POCI-01-0247-FEDER-039676), co-financed by the European Regional Development Fund, through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

ABSTRACT REST services are nowadays being used to support many businesses, with most major companies exposing their services via REST interfaces (e.g., Google, Amazon, Instagram, and Slack). In this type of scenarios, heterogeneity is prevalent and software is sometimes exposed to unexpected conditions that may activate residual bugs, leading service operations to fail. Such failures may lead to financial or reputation losses (e.g., information disclosure). Although techniques and tools for assessing robustness have been thoroughly studied and applied to a large diversity of domains, REST services still lack practical approaches that specialize in robustness evaluation. In this paper, we present a tool (named bBOXRT) for performing robustness tests over REST services, solely based on minimal information expressed in their interface descriptions. We used bBOXRT to evaluate an heterogeneous set of 52 REST services that comprise 1,351 operations and fit in distinct categories (e.g., public, private, in-house). We were able to disclose several different types of robustness problems, including issues in services with strong reliability requirements and also a few security vulnerabilities. The results show that REST services are being deployed preserving software defects that harm service integration, and also carrying security vulnerabilities that can be exploited by malicious users.

INDEX TERMS REST, RESTful, web API, web services, robustness testing.

I. INTRODUCTION

REST is an architectural style for the design of services that is based on the principles that support the World Wide Web [1]. Recently, it has become the *de facto* standard way for offering a service on the Web [2]. A REST service (also known as RESTful, Web API, or REST API) relies on Uniform Resource Identifiers (URI) for identification of its resources (e.g., a user profile, a purchase order) and on HTTP for exchanging messages (which are usually JSON documents). The use of HTTP includes the presence of a verb (e.g., GET, POST) that specifies the type of operation that should be executed over the identified resource [1].

Major companies like Google, Amazon, Instagram, Spotify, and Slack are now providing access to their services via REST services. In fact, the use of other types of interfaces to expose services is now residual, at least considering popular sites on the Web [2]. REST is a relatively loose architectural style and some rigid aspects that are present in other similar styles or technologies (e.g., SOAP services), like the mandatory presence of an interface description document (e.g., a WSDL document), lost their meaning in REST [1]. At the time of writing, there is no standard way of describing the interface of a REST service although the OpenAPI specification [3] is gaining popularity [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

The less rigid access to REST services opens space for unexpected inputs to be sent to services, potentially triggering residual faults that were not caught by Verification and Validation (V&V) activities performed by developers (e.g., static analysis, code inspections, and unit testing). Although it may be acceptable for a client to make mistakes and invoke a certain operation with wrong parameters (e.g., out of bounds or in the wrong format), it is not acceptable that the server crashes or returns some kind of incorrect response. This is especially true when the service is supporting a business (or mission) critical activity.

The additional software layer and tools required to provide REST services also add complexity to the development. Currently, the developer has to focus on new tasks like matching the right HTTP verbs to certain operations, specifying arguments in different ways (e.g., in REST many times a particular argument can be found as a path parameter, i.e., it is part of the URI that identifies the resource [2]), or correctly documenting the API. For instance, a mismatch between the API documentation and the actual service implementation may lead clients to perform wrong invocations by introducing mistakes in the request payload. Regardless of what is sent by a client, the service must be prepared to respond in a robust manner.

Robustness is the degree to which a certain system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [4] and has been the target of several studies in the last decades. Koopman *et al.* [5], [6] have most notably conducted work on the operating systems domain, but, especially due to the valuable outcomes produced by robustness testing, numerous authors have designed approaches and tools for other domains. These include communication software [7]–[9], embedded systems [10]–[12], middleware [13]–[15], self-adaptive systems [16], web applications [17], and SOAP services [18]–[20]. Despite this variety of explored domains, the robustness of REST services has been largely disregarded by researchers and practitioners.

In this paper, we aim at filling this gap by presenting an approach and a tool, named bBOXRT - black BOX tool for Robustness Testing of REST services. The approach, implemented by bBOXRT, uses a service description document as input to generate a set of invalid inputs (e.g., empty and boundary values, strings in special formats, malicious values) that are set to the service in combination with valid parameters. The tool can also operate as a fault injection proxy between client and server (i.e., without requiring information regarding the service interface). Service responses are, at the time of writing, preliminarily analysed for suspicious cases of failure (e.g., the presence of exceptions in the response, response codes referring to internal server errors) and are stored by the tool for a later detailed analysis by the tool user.

We demonstrate the usefulness of our approach and our tool's capabilities by performing tests over a set of 52 services (comprising 1,351 operations) that fit in different types: public services, middleware management services (i.e., Docker), services built in-house (i.e., two TPC performance benchmarks), and private services. We performed a total of 399,901 tests, in which we disclosed a total of 24,373 robustness problems. In addition to being important information for the service developers and providers, results mostly show that bBOXRT is able to disclose different kinds of problems in the tested services (usage of incorrect data types, missing input validation), which map to different corrective actions (e.g., correcting a specification or fixing the implementation). It was also able, despite not being its main focus, to disclose security issues (e.g., services carrying SQL Injection security vulnerabilities) and also private information (e.g., stack traces, database queries, database instance names). The source code of our tool along with the detailed results from a robustness testing campaign of REST services are available online at [21].

The main contributions of this paper are the following:

- The definition of an approach that specializes in evaluating the robustness of REST services and requires minimal information regarding the interface of the service being tested;
- A robustness testing tool, named bBOXRT, that implements our approach and is readily available to be used by researchers and practitioners;
- The practical application of the tool to an heterogeneous set of 52 REST services, including business-critical services, in which it was able to show the presence of several different software defects (including security vulnerabilities) and the presence of bad programming practices, illustrating the overall usefulness of the approach.

The rest of this paper is organized as follows. Section II presents related work and Section III presents the approach, mapping it to the main components of our tool and how they interact. Section IV presents the experimental evaluation and results and Section V presents the main threats to the validity of this work. Finally, Section VI concludes the paper.

II. RELATED WORK

Research on assessing dependability and specialized dependability properties, like robustness of software systems started several decades ago. Robustness testing approaches [16], [22] typically make use of *fault injection*, in which faults are deliberately introduced into the system (e.g., to evaluate fault tolerance mechanisms [23]). A typical case found across different works [11], [22], [24]–[26] is the corruption of parameters in order to activate implementation faults (i.e., trying to trigger failures), which is also known in the literature as *error injection* [23].

Robustness testing became popular mostly due to its application to operating systems (OS) by Koopman *et al.* [5], Kropp *et al.* [22], Shelton *et al.* [27] among others [28]–[30], but its application to many other different types of systems (which hold different specificities) has been the object of research in numerous domains. Robustness testing techniques

can be found, in general, applied to communication systems [7]–[9], embedded systems [10]–[12], [31], middleware [13]–[15], [32], web applications [17], [33], COTS [34]–[36], autonomous and adaptive systems [16], [25], [26], [37], and SOAP web services [18]–[20], [24], [38].

The work in the Ballista project [22] is recognized as a landmark regarding the use of interface-level fault injection to evaluate the robustness of operating systems. The approach is based on exposing the operating systems to invalid input conditions introduced at its interface. Ballista works by calling each interface function of the system under test using a combination of valid, boundary or invalid values, and logging responses. The non-valid values are selected from a database of faults that are essentially limit conditions that apply to certain data types (e.g., the maximum value for an integer, the minimum value for a float, values around these limits) or values holding special characteristics (0, 1, −1). Triggered failures are categorized according to their severity using a five step failure mode scale, the CRASH scale [5].

While the work within Ballista mostly used the application-operating system interface, other author created approaches using different perspectives, namely the OS-driver interface, such as the work by [39] which also uses more diverse faults, e.g., bit-flips, random values (e.g., as in fuzzing approaches), in addition to invalid and boundary values. In [39] the authors use 4 categories (including a 'no failure' category) to classify behavior according to the severity of the observed failures.

A recent trend regarding operating systems robustness evaluation approaches is set on mobile operating systems [40], [41] and wearable applications. Cotroneo *et al.* [40] propose a tool, named AndroFIT, for the evaluation of dependability properties of the Android operating system. The tool relies on the injection of faults on Inter Process Communication (IPC) messages, library, and system calls. The faults used include random and boundary faults, bit-level faults, but also faults related with timing aspects (delays or unresponsiveness), and invalid return values (i.e., interactions that return an error code or terminate abruptly with an exception). The observed behavior is categorized using a four level scale (including 'no failure'), corresponding to the occurrence of a crash, a fatal error, or a non-responsive application.

Liu *et al.* discuss a tool for fuzzing Android system services in [41]. The tool identifies target interfaces and further deeply nested interfaces (and other information like variable types, names, and dependencies) to generate random sequences of transactions to be executed by the target system. The tool showed to be able to trigger several failures, which served to identify several software bugs. Wearables have been the target of Yi *et al.* [42], in which the authors target IPC messages and user interface events using random values. The experiments were able to trigger several different types of failures, including reboots, crashes, or non-responsive applications. In [43], the authors present a more extensive work that takes the state of the system into consideration.

The robustness evaluation works that are the closest to the domain of our work aim at evaluating the robustness of SOAP web services. Salva and Rabhi [19] test stateful SOAP web services by actively modelling their state (in the form of Symbolic Transition Systems) through repeated system calls. The work by Laranjeiro *et al.* [24] uses the mandatory information present in a WSDL document, which describes the service interface to generate a random workload (i.e., set of valid calls) that is used to observe the normal service behavior. Then a set of invalid inputs are injected into the workload requests and used to build test cases. Responses from the service are observed and compared to the baseline behavior and analyzed to find robustness problems, which are categorized using an adaptation of the CRASH scale [5].

Salas *et al.* [38] use XML code injection, cross-site scripting and XPath injection in SOAP messages to exploit robustness as well as security issues in services. Other works focus on orchestrated compositions of SOAP web services, such as Kuk and Kim [18] where the specifications of the component services are read to generate virtual versions that will then trigger erroneous behavior, or Ilieva *et al.* [20] that propose a framework for testing BPEL service compositions, which is able to inject invalid, boundary, unexpected or random data in requests being processed by the composition. Further detail on robustness evaluation techniques can be found at [44].

There are several recent approaches regarding the functional evaluation of REST services, which are based on diverse techniques (e.g., genetic programming, fuzzing, model-based testing). Regarding black-box approaches, most notably, Ed-douibi *et al.* [45] use a model of the OpenAPI specification to generate test cases that are based on invalid and boundary conditions. The tool is offered as a plugin for Eclipse and is based on a relatively small set of faults, that do not account for security issues. The tool is demonstrated on a set of services selected from APIs.guru, where it was able to detect several cases of failing services.

Viglianisi *et al.* [46] propose a black-box approach for testing REST services, which is based on the application of just three types of mutation operators: missing inputs, wrong types, limit value overflow. The approach was demonstrated on a set of services listed at APIs.guru, and showed to be able to trigger crashes of the services being tested. In [47] the authors propose an approach that is based on the creation of faulty variants of test cases that essentially use omission and out-of-range faults, and has the particular goal of exploring inter-parameter dependencies. Thus, it requires that an OpenAPI specification is written. Also worthwhile mentioning is the work by Karlsson *et al.* [48], which is also based on test generation (despite not explicitly targetting robustness), although it requires the specification of service properties, which is time or resource consuming, requires expertise, thus not always applicable.

Regarding REST testing approaches (in general), there are several other cases which are intrinsically different from our own, such as being white-box approaches [49], or state-based [50], [51], model-based [52], [53] or implementing other

kinds of testing (e.g., regression testing) [54]. In the industry there is also a wide variety of tools for testing REST services [55]–[57], but these largely focus on functional or performance testing and do not specialize in evaluating robustness. Their adaptation to a typical robustness testing approach is either not possible (e.g., licensing issues) or not practical due to involving complex adaptation code.

In summary, current testing approaches for REST are either complex ones (e.g., involving genetic algorithms, requiring expertise for necessary artifacts) or are simply intrinsically different from the one proposed in this paper (e.g., white-box aproaches, model-based).

In this paper, we describe bBOXRT, a simple rule-based black-box tool which is particularly specialized in robustness evaluation and demonstrate its effectiveness in disclosing robustness issues, even in highly tested business critical services, where robustness is a major concern.

III. APPROACH FOR EVALUATING THE ROBUSTNESS OF REST SERVICES

This section describes the approach defined for testing the robustness of REST services. We first explain the main concepts that support the approach and then introduce bBOXRT tool. We overview the tool's internal architecture and explain the different roles and responsibilities of its main components and how they work together to support each of the steps of the approach.

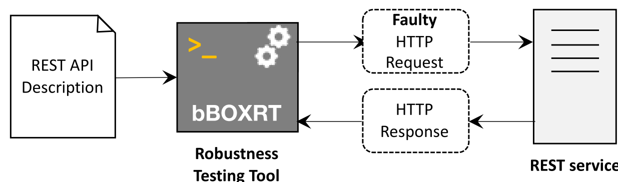

FIGURE 1. Conceptual view of the approach.

A. APPROACH OVERVIEW

The conceptual view of our approach is shown in Figure 1. In practice, using information regarding the interface of the service under test (i.e., API description document), our approach generates a combination of valid and invalid requests that attempt to activate faults present in the service. The approach is decomposed in the following steps: -

- **Step 1: Interface description analysis**. Basic information about the service under test is collected by reading and analysing the service's interface description document. Information regarding operations (e.g., resource URIs and HTTP verbs to use), input and output data types, error codes, or example requests is gathered to be used in the next steps.
- **Step 2: Workload generation and execution**. Valid requests (i.e., correct according to the specification) are generated and sent to the service. This allows us to understand the behavior of the service in presence of a non-faulty workload.
- **Step 3: Faultload generation and execution**. Faulty requests are created by injecting a single fault in each request (e.g., a field is removed from a JSON document). The faulty requests are sent to the service in an attempt to trigger erroneous behaviors.
- **Step 4: Result storage and analysis**. Service responses and test metadata (e.g., type of fault injected, resource targeted) is stored for supporting the subsequent behavior analysis.

In the next section (III-B), we explain these steps in further detail and how they map to our tool's different components.

B. TOOL ARCHITECTURE AND OPERATION

The tool was implemented in Java, and was developed with modularity and extensibility in mind. Users may add new functionalities to some of the tool's components to optimize its operation with respect to their use cases. The tool's interface is command-line and its source code and documentation are available at [21]. The architecture of the tool is shown in Figure 2, which depicts bBOXRT as a container whose frontiers are delimited by a dashed rectangle. It is comprised of multiple components that interact with each other and/or with external entities, as detailed in the following paragraphs.

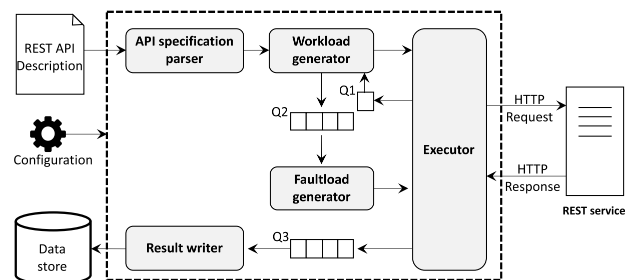

FIGURE 2. bBOXRT architecture.

The *API Specification Parser* is the component that supports **Step 1 - Interface description analysis** (identified in the previous section). It reads an OpenAPI document (formerly known as Swagger [58]), specified in either JSON or YAML format, that describes the interface of a given REST service. The OpenAPI specification is becoming a popular option for describing interfaces for this type of services [2] and this is the reason we chose to support it, by default. However, users may extend the tool to support for other API specification languages (e.g., RAML [59]).

The main idea behind this first step is to identify and extract relevant information for testing the service. An API specification defines one or more target server URLs and the set of unique API endpoint URIs (i.e., unique resources at the server). Each endpoint is associated to one or more HTTP verbs [1], typically POST, GET, PUT and DELETE (PATCH and HEAD may also be found in some APIs). Each set `{endpoint, verb}` is named an operation. Finally, each operation may have from none to several input parameters (e.g., headers, payload), as well as a set of different

expected responses, each identified by a unique HTTP status code (e.g., 201 Created or 401 Unauthorized).

In **Step 2 - Workload generation and execution**, we have the *Workload Generator* and the *Executor* components involved. The main idea behind the *Workload Generator* is to be able to generate a set of valid HTTP requests. If there is access to an existent workload, the tool is also able to work as a proxy for a certain client application or simply read a set of stored requests (detailed instructions on how to start the tool are available at [21]). Otherwise, this component generates a random workload that complies with the specification (i.e., in terms of data types, domain of the values used, and overall parameter structure). The possible locations for the generated values are the request payload (i.e., typically a JSON object), the HTTP headers, the endpoint URI itself (i.e., which is named a path parameter) or the HTTP query string. The tool user can set the number of times each possible parameter of an operation is to be randomly generated by specifying a configuration value `WL_REP`. This may have the effect of increasing the overall diversity of the set of requests.

In practice, the *Workload Generator* creates a request, dispatches it to the *Executor* (which sends the request to the service) and waits for the response on a queue (*Q1* in Figure 2). The response is useful for allowing the *Workload Generator* to gather feedback (e.g., success or failure) and having the possibility of adjusting the request generation process (the tool currently does not perform any specific adjustment of the workload generation process, but we intend to implement specific adjustment strategies in future work). For cases where the response represents an unsuccessful case (e.g., status codes 4xx or 5xx), the user may set the boolean configuration value `WL_RETRY` to `true`, which will make the *Workload Generator* keep the failed request in memory to be retried after executing all other workload requests (e.g., operations dependent on the execution of others). Requests holding a success status code (e.g., 200 OK) are placed in a queue (*Q2* in Figure 2) and will be used in the next step of the approach. The current step ends if all generated requests are executed or if the user sets a maximum time limit (i.e., by specifying the `WL_MAX_TIME` parameter).

The *Faultload Generator* component is responsible for injecting faults in the workload requests whose execution was carried out with success (i.e., those available in queue *Q2*). The faulty requests will then be delivered to the *Executor*, thus supporting **Step 3 - Faultload generation and execution**. Internally, the *Faultload Generator* is helped by the Fault Mapper (not visible in Figure 2) that keeps track of the possible injection locations in a request. This component keeps track of which parameters composing each request have been covered with faults (and which faults) and, thus, it guides the fault injection process by preventing the injection of faults at an already explored location.

Table 1 shows the fault model currently used by the *Fault Generator*. The faults are organized by data type and formatted as defined in the OpenAPI specification [58]. For that reason, a data type may have multiple formats (e.g., a string may contain a sequence of bytes or a date, or it may have an unrestricted format - the default format). Each of the 57 faults is described as a mutation rule to apply on a given request parameter. Faults that share a common property are grouped into a single fault type (e.g., numeric faults that focus on near or over boundary values are grouped in the numeric boundary fault type). Note that for the string fault types, printable or non-printable refers to the respective portions of the ASCII table, and malicious essentially refers to SQL injection strings. The set of SQL injection strings comprises several hundreds of malicious strings (over 800 strings at the time of writing), extracted from [60]. Also, regarding the date and date-time formats, we note that the faults of the former are also applicable to the latter.

For each parameter of each request, the set of applicable faults (those that match the parameter's data type) is finished in order. Each generated fault is placed at the location (e.g., path, payload) of the parameter being targeted. This process is repeated `FL_REP` times, which allows us to understand if the service shows consistent behavior in presence of a specific fault. For certain types of faults, it also allows to achieve a greater diversity of invalid requests for fault types of stochastic nature (e.g., remove a random element from an array). When the set of faults is finished for a given parameter, the Fault Mapper is set to point to the next unexplored parameter in that request.

After a fault is injected, the corresponding faulty request is dispatched to the *Executor*, which then sends it to the service and waits for the response. Once returned, the response is placed in a queue (*Q3* in Figure 2) along with the original request, the reference of the injected fault and the parameter targeted. As in Step 2, the user may specify a configuration to limit the duration of this process (i.e., `FL_MAX_TIME`), otherwise all applicable faults are injected. Finally, the *Result Writer* component retrieves all pending information from queue *Q3* and saves it to persistent storage for later analysis, thus supporting **Step 4 - Result storage and analysis**.

The bBOXRT tool does not currently fully automate the analysis of the service behavior, due to the typical complexity involved in the identification of robustness problems (with exception of cases where a perfect service specification exists). The main difficulty is related with the large diversity of responses (i.e., the underlying systems are also heterogeneous) that can be generated by a web service, which usually requires the presence of an expert to manually distinguish robust behavior from non-robust. We are currently exploring the applicability of machine learning algorithms to this task, which we plan to integrate in the tool in future work. Despite this, we propose an analysis that covers two main aspects (described in the next paragraphs): i) the severity of the observed failure; and ii) the behavior of the service.

The behavior of the service is analysed and classified with a failure mode scale, such as the CRASH scale [5], in which we perform minor adjustments to fit the particular case of REST services. CRASH distinguishes the following cases of failure:

TABLE 1. Fault model.

OpenAPI Data types		Fault ID	Parameter mutation rule
All	Applicable to all types	Any_Empty	Replace with empty value
		Any_Null	Replace with null
Array	–	A_Duplicate	Duplicate random elements in the array
		A_RemoveOne	Remove random element from array
		A_RemoveAll	Remove all elements in the array
		A_RemoveAllButFirst	Remove all elements in the array except the first one
Boolean	–	B_Negate	Negate boolean value
		B_Overflow	Overflow string representation of boolean value
Number	32-bit integer, 64-bit integer, Single-precision (float), Double-precision (double)	N_Add1	Add 1 unit
		N_Sub1	Subtract 1 unit
		N_ReplacePositive	Replace with a random positive value
		N_ReplaceNegative	Replace with a random negative value
		N_Replace0	Replace with 0
		N_Replace1	Replace with 1
		N_Replace-1	Replace with -1
		N_ReplaceTypeMax	Replace with data type maximum
		N_ReplaceTypeMin	Replace with data type minimum
		N_ReplaceTypeMax+1	Replace with data type maximum + 1
		N_ReplaceTypeMin-1	Replace with data type minimum - 1
		N_ReplaceDomainMax	Replace with parameter domain maximum
		N_ReplaceDomainMin	Replace with parameter domain minimum
		N_ReplaceDomainMax+1	Replace with parameter domain maximum + 1
		N_ReplaceDomainMin-1	Replace with parameter domain minimum - 1
String	No format specified, Password	S_AppendPrintable	Append random printable characters to overflow maximum length
		S_ReplacePrintable	Replace with random printable character string of equal length
		S_ReplaceAlphanumeric	Replace with random alphanumeric string of equal length
		S_AppendNonPrintable	Append random non-printable characters
		S_InsertNonPrintable	Insert random non-printable characters at random positions
		S_ReplaceNonPrintable	Replace with random non-printable string of equal length
		S_Malicious	Append SQL injection attack to original input
	Byte, Binary	Byte_Duplicate	Duplicate random elements to overflow maximum length
		By_Swap	Swap a random number of element pairs in the string
	Date	D_Add100Years	Add 100 years to the date
		D_Sub100Years	Subtract 100 years from the date
		D_LastDayPreviousMil	Replace with last day of the previous millennium
		D_FirstDayCurrentMil	Replace with first day of the current millennium
		D_Replace1985-2-29	Replace with invalid date 1985-2-29
		D_Replace1998-4-31	Replace with invalid date 1998-4-31
		D_Replace1997-13-1	Replace with invalid date 1997-13-1
		D_Replace1994-12-0	Replace with invalid date 1994-12-0
		D_Replace1993-8-32	Replace with invalid date 1993-8-32
	Date-time	–	(All mutations from Date format apply)
		T_Add24Hours	Add 24 hours to the time
		T_Sub24Hours	Subtract 24 hours from the time
		T_Replace13:00:61	Replace with invalid time 13:00:61
		T_Replace10:73:02	Replace with invalid time 10:73:02
		T_Replace25:58:04	Replace with invalid time 25:58:04
		T_Replace04:03:60	Replace with invalid time 04:03:60
		T_Replace07:60:15	Replace with invalid time 07:60:15
		T_Replace24:05:01	Replace with invalid time 24:05:01

- **Catastrophic:** The service supplier (i.e., the underlying middleware) becomes corrupted, or the server or operating system crashes. A restart of the system does not allow recovering from the failure.
- **Restart:** The service provider becomes unresponsive and must be terminated by force. A restart of the system will allow recovering from the failure.
- **Abort:** Abnormal termination when executing a certain service operation. For instance, unexpected behavior occurs when an unexpected exception is thrown by the service implementation.
- **Silent:** A service operation cannot be concluded, or is concluded in an abnormal way, and the service implementation does not indicate any error.
- **Hindering:** The returned error message or code is incorrect and does not correspond to the actual error.

Some of the above failure modes are not distinguishable, unless we have access to the service provider (e.g., a client will not be able to distinguish between a catastrophic and a restart failure), still when full access exists it is important to distinguish the different cases of severity.

The service behavior should also be characterized in a more detailed manner. In previous work [24], and based on the analysis of results of robustness testing applied to SOAP web services, we proposed the application of a set of behavior tags. The main idea is to characterize behavior using a finer level of granularity (i.e., the CRASH scale may be too coarse-grained) and, at the same time, abstract implementation details that result in different response messages that, in practice, represent the same behavior (e.g., a null pointer expressed with different messages at the server side).

TABLE 2. Behavior tags.

Tag ID	Behavior tag	Description
AE	Allocation error	An exception is thrown as a consequence of the allocation of memory addresses (e.g., an out-of-memory exception)
AO	Arithmetic operations	An indication of an arithmetic error is returned by the service operation
AOB	Array out of bounds	Occurrence of an array access with an index that exceeds its limits (upper or lower)
AOF	Argument out of format	The operation requires a restriction on the format of a parameter, which is not specified in the interface specification document
CI	Conversion issues	A conversion problem exists in the service (e.g., data type conversion)
CSD	Command or schema disclosure	An internal command is completely or partially disclosed (e.g., an SQL statement), or the data schema is revealed (e.g., the table names in a relational database)
DAO	Data access operations	A problem exists related with data access operations
DZ	Division by zero	The service operation indicates that a division by zero has been attempted
IFND	Internal function name disclosure	The name of an internal or system procedure is disclosed (e.g., a database procedure)
NR	Null references	A null pointer or reference exception is thrown by the server application
O	Other	Any other service response that does not fit into any of the previous categories
Ovf	Overflow	The operation is unable to properly handle a value that is larger than the capacity of its container, signalling an overflow error
ParseE	Parsing error	An exception is thrown as a consequence of failure to parse input or improper input handling
PersE	Persistence error	An exception is thrown signalling a persistence problem (e.g., SQL exception thrown due to improper parameter handling)
SIND	System instance name disclosure	The name of a system instance is revealed to the client (e.g., a database instance name)
SRD	Server resource disclosure	Information about code structure, filesystem or physical resources of the server is disclosed
SSFM	Specific server failure message	An exception is thrown and application-specific information is revealed. This information is too vague or too context-specific to allow an association with another tag
SUD	System user disclosure	A system username or password is exposed to the client (e.g., a database or operating system username)
SVD	System vendor disclosure	System vendor information is disclosed (e.g., database or operating system vendor)
WEI	Wrapped error information	An error response is wrapped in an expected object. The response indicates the occurrence of an internal error
WTD	Wrong type definition	The service operation expects a value whose type is not consistent with what is defined in the service interface specification file

Table 2 presents our proposal to better characterize service behavior is a revision of the set of tags used in [24], which we now adapted to fit the context of REST services. We performed minor alterations to the tags as some SOAP elements do not exist in REST (e.g., WSDL documents). Also, we extended the set with two extra tags, allocation error (AE) and parsing error, after analysing the experimental results described in the next section.

IV. EVALUATION AND RESULTS

This section describes the experimental campaign carried out to show the practical usefulness of our approach. We first overview the experiments preparation and setup (section IV.A), then we detail the main results (section IV.B) and conclude with the main findings observed during the experiments (section IV.C).

A. EXPERIMENTAL SETUP

To show the usefulness of the approach, we used bBOXRT to test various types of services, which we selected according to the following different factors: i) service dimension (e.g., number of endpoints); ii) workload needs (e.g., just a few parameters or several); iii) service observability (e.g., from black-box to full access to code); and iv) context of usage (e.g., some services serve to manage other services, some services have strict reliability requirements). This resulted in the definition of the following five sets of interest, which represent a mix of the different factors presented and that are described in the next paragraphs:

- Set 1: Popular services;
- Set 2: Public services;
- Set 3: Middleware management services;
- Set 4: Private services;
- Set 5: In-house services.

We began by identifying a set of 6 services, named **Set 1 (Popular services)**, comprising a total of 395 testable operations (out of 594) provided by popular Web sites. We used the *alexa.com Top 500 popular web sites* rank to handpick the services, and we were particularly interested in services having a seemingly detailed specification. The selected services are Google Drive, Google Calendar, Spotify, Trello, Slack, and Giphy. These services are known to be used in business-critical environments (in particular, the former four) and are expected to be developed with strong reliability requirements (in *lato sensu*), as failures have direct (and also indirect) impact on the business of the respective companies.

A set of 40 REST services was selected from the APIs.guru online database to compose **Set 2 (Public services)**. APIs.guru is a well known large repository of hundreds of public services and has been used as information source for REST services in other works [61]–[63]. We performed tests against 823 operations (out of 1,012) of the services, which showed to be quite diverse (i.e., considering the abovementioned criteria), being useful for showing the effectiveness of our approach when applied to such different cases.

For **Set 3 (Middleware management services)**, we selected a single service from Docker, a platform used for managing virtualized software containers. We performed tests against 70 of the 106 Docker Engine REST service operations [64]. In the case of the Docker service, observability is set at a higher level, as we also had the server logs to complement the analysis carried out over the service responses.

Regarding **Set 4 (Private services)**, we performed tests against all the 45 available operations of Kazoo Crossbar, a business-critical, cloud-based, Voice over IP and telecommunications service. This service supports part of the business of a private company valued at over 1B$, thus, strong reliability requirements are involved as any failure may have impact on the business. For this service, we had no access to source code, but a rather complete interface description and also direct contact and feedback from the company's developers are available.

Finally, for **Set 5 (In-house services)** we tested 4 services holding 18 operations. The set of services is composed of two implementations of TPC-App (version Vx0 and version VxA), which is a performance benchmark for web services that emulates the business of an online store [65], and two implementations of TPC-C (Vx0 and VxA) which is a performance benchmark for transaction processing systems that emulates the business of a wholesale supplier [66]. All implementations were created in-house, where Vx0 and VxA mostly differ in how the developers built the SQL queries used by the services. While Vx0 uses `PreparedStatements` to clean inputs and avoid SQL code injection, VxA directly concatenates static strings with user input without performing any validation, thus being prone to being unsecure. Both TPC-App versions have an average cyclomatic complexity [67] of 4.67 (measured by the Statistic plugin [68] of IntelliJ IDEA 2019.2.3 [69]), and Vx0 has 558 lines of code (LoC) while VxA has 566. The TPC-C versions have an average cyclomatic complexity of 17.2, Vx0 has 1128 LoC and VxA has 1179 LoC. We also ran a static analysis tool, namely SpotBugs 4.0.2 [70] (with the Find Security Bugs plugin 1.10.1 [71]), which was able to identify 6 SQL injection vulnerabilities in TPC-App VxA and 30 in TPC-C VxA. We manually inspected the code to confirm the existence of these issues, which were due to query concatenation with user input. SpotBugs also identified further issues, but they were mostly bad coding practices or false-positives.

The above five sets of services comprise 1,775 operations, of which we tested 76% (1,351), because, at the time of writing, the tool does not yet support generation of certain, less usual, payloads (i.e., media types like files or MessagePack messages [72]). The experiments carried out against the 1,351 operations used the same values for the configuration parameters of bBOXRT, namely: `WL_REP=10` indicates that each request parameter is generated 10 times; `WL_RETRY=true` denotes that each unsuccessful request is retried once; and `FL_REP=3` means that each applicable fault is used 3 times per parameter. Both `WL_MAX_TIME` and `FL_MAX_TIME` were set to `0`, meaning that neither the workload or faultload execution phases were time-limited. In the case of Set 5 (In-house services), we used the benchmarks' client emulator applications and our tool worked only as a fault injection proxy.

As a result of the above setup, the tests produced 399,901 responses that needed to be manually analyzed for the identification of robustness issues. For the analysis, we grouped the responses by operation and then by status code. In a first round, and considering the large number of responses to analyse, we performed a fast search to signal responses showing obvious signs of robustness problems (e.g., a 500 status code response holding an SQL exception) or strongly suspicious cases (e.g., a stack trace in the response payload). All other cases were optimistically marked as being correct behavior. This resulted in 68,372 responses that we analysed in detail. For each signalled response, we analysed its whole contents and the respective service specification. Responses that did not comply with the service specification were marked as a problem, with the same happening with unspecified cases that showed obvious signs of unexpected behavior. The whole process resulted in a total of 68,245 individual responses referring to robustness problems (repeated test cases are included in this number). Some doubts remained in 127 cases (e.g., the service states some problem occurred, but there is no clear indication that a failure has occurred), thus we marked them as *Dubious* and omitted them from the discussion.

B. EXPERIMENTAL RESULTS

We begin with an overview of the results and then drill down into the detail of the results for each set of services. In about half of the services tested (i.e., 26 out of 52) we detected at least one case of robustness problem. At the operation level, bBOXRT detected at least one robustness problem in 167 of the 1,351 service operations tested (i.e., in about 12% of the operations). Of the 7,167 parameters available in these operations, 525 were involved in at least one robustness problem. The tool actually disclosed 24,372 problems, each being the consequence of injecting one type of fault at a particular parameter of a certain service operation (i.e., not counting any repetitions of a certain fault). The vast majority (i.e., ∼97%) of the problems found represent failures of type *Abort* (i.e., 23,764 failures that refer to unexpected exceptions or error messages), with 608 fitting the *Hindering* failure mode.

Out of the total 49 faults implemented (shown previously in Table 1), 44 were able to trigger some failure. The five faults not involved in failures are: the faults `N_ReplaceDomainMax` and `N_ReplaceDomainMax+1` for numeric parameters; the fault `S_AppendPrintable` for string parameters; and the faults `Bi_Duplicate` and `Bi_Swap` for byte and binary parameters, respectively. The former two faults were used but not involved in failures, while the remaining three were not used at all, due to the respective data types not being found in the tested sets of services.

Figure 3 presents the distribution of the 44 types of faults that resulted in the 24,372 robustness problems (problems arising from repetitions of a certain injection are not counted, as they are essentially a confirmation that the problem exists and its observation is repeatable). A longer bar means that a particular type of fault was involved in disclosing a larger number of robustness problems, thus the x-axis represents the frequency of fault types (i.e., the number of times a given fault type triggered a robustness issue divided by the total number of robustness problems). The numbers between parenthesis show the total number of times a given fault type triggered a robustness issue (i.e., considering all parameters where it was injected and excluding repetitions), and add up to the total of 24,372. It is worth mentioning that we found 146 cases where the random workload actually triggered a failure. In 81 of these we determined that the workload had emulated one of the faults and we included these cases in the analysis. The remaining 65 were excluded from the analysis,

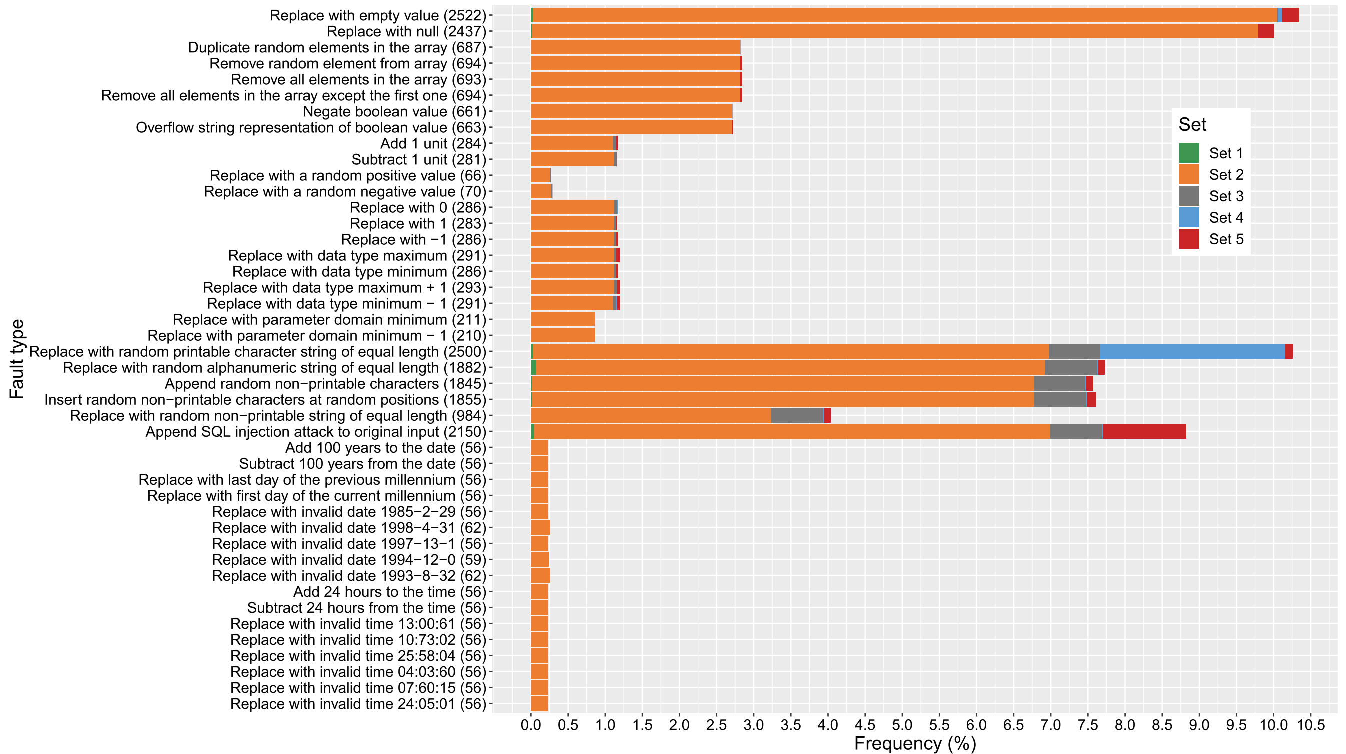

FIGURE 3. Distribution of the successful faults over the service sets.

as we were unable to determine what was the reason for the failure.

In Figure 4 we can see the prevalence of behavior tags with respect to the services in which robustness problems were disclosed. Each tag is represented by its abbreviated name (see Table 2 for the complete designation). The numbers between parenthesis in the vertical axis refer to the overall number of services displaying a given behavior (i.e., marked by a certain tag).

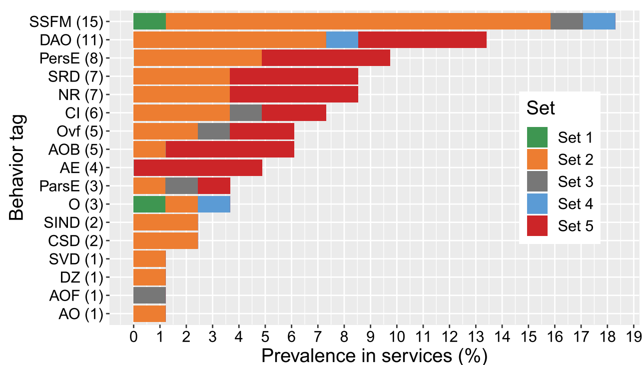

FIGURE 4. Prevalence of the behavior tags in the tested services.

Figure 4 shows that the top issues across the services include cases that are too vague to be classified (e.g., general server error responses), meaning that some services are not really informative to clients, which is an obstacle for reliable service integration. There are also frequent issues related with storage operations, which show that robustness issues can be triggered deeper in the service code, namely at the points of contact with subsystems. Server resource disclosure is also a frequent behavior, with some services disclosing internal information that should be kept from the outside. Null references and conversion issues are also still in the top cases of problems found across services, which are known to be typical sources of robustness problems [24].

Table 3 presents the cases where robustness problems were detected in Sets 1 and 2 (the results regarding Sets 3 through 5 are discussed later in this section). Table 3 includes the name of the service, the number of operations tested (and the total number of operations available in that service), the status code of the robustness failure, the operations with error responses, the number of arguments for those particular operations and the number of vulnerable arguments, the target datatype of the injected faults that resulted in some failure, a description of the failure using the behavior tags, and the CRASH scale classification [5]. Each type of injected fault is accompanied by a number between parenthesis, which refers to the number of arguments it affected in a particular operation (and resulted in a robustness failure). We use the word *All* along with the number of affected parameters, when all faults of a given data type triggered a certain problem. In some cases the list or name of operations is too large to display and we replaced it with the number of operations affected. We also removed all cases of dubious behavior, as they are not relevant

TABLE 3. Results of the experimental evaluation for Sets 1 and 2.

Set	API	Operations tested	Status code	Operations with error responses	Parameters: Total	Parameters: Affected	Injected fault types: Any	Injected fault types: Array | Bool | DateTime	Injected fault types: Number	Injected fault types: String	Behaviour Tags	CRASH
Set 1	**Google Drive**	**30 (65%)**	500	drive.permissions.list, drive.revisions.list	9	2				S_ReplaceAlphanumeric (2)	**O**	**H**
Set 1	**Spotify**	**36 (97%)**	500	GET_/users/{user_id}/playlists/{playlist_id}/followers/contains, PUT_/users/{user_id}/playlists/{playlist_id}/tracks, DELETE_/users/{user_id}/playlists/{playlist_id}/tracks	9	3	Any_Empty (2) Any_Null (1)			S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_Malicious (1)	**SSFM**	**A**
Set 1	**Spotify**		502	PUT_/users/{user_id}/playlists/{playlist_id}/followers, DELETE_/users/{user_id}/playlists/{playlist_id}/followers, GET_/users/{user_id}/playlists/{playlist_id}/followers/contains	8	4				S_ReplacePrintable (2) S_ReplaceAlphanumeric (4) S_Malicious (2)	**SSFM**	**A**
Set 2	**Ably**	**20 (91%)**	500	getPresenceOfChannel	6	2	Any_Empty (2)				**SSFM**	**A**
Set 2	**Ably**		500	getPresenceOfChannel		2	**All** (2)				**DAO, SIND**	**A**
Set 2	**Ably**		500	getStats	7	1	**All** (1)				**SSFM**	**A**
Set 2	**Api2Pdf**	**9 (100%)**	400	mergePost	3	1				S_ReplaceAlphanumeric (1)	**DAO, SRD**	**A**
Set 2	**AppVeyor**	**45 (88%)**	500	reRunBuild, addRole, encryptValue, updateCollaborator, updateUser, startBuild, updateProjectEnvironmentVariables, updateProjectBuildNumber, addEnvironment, updateRole, addProject, startDeployment, updateEnvironment, updateProject	206	170	**All** (170)	**All Array** (49) All Boolean (47) All Date&Time (4)	N_Add1 (17) N_Sub1 (17) N_Replace1 (17) N_Replace0 (17) N_Replace-1 (17) N_ReplaceTypeMax (17) N_ReplaceTypeMin (17) N_ReplaceTypeMax+1 (17) N_ReplaceTypeMin-1 (17) N_ReplaceDomainMin (15) N_ReplaceDomainMin-1 (15)	S_ReplaceAlphanumeric (52) S_ReplacePrintable (52) S_AppendNonPrintable (52) S_InsertNonPrintable (52) S_ReplaceNonPrintable (52) S_Malicious (52)	**SSFM**	**A**
Set 2	**Auckland Museum**	**5 (83%)**	400	get search	3	1				S_ReplaceAlphanumeric (1)	**SSFM**	**A**
Set 2	**Auckland Museum**		404	get search		2	Any_Empty (2)					**A**
Set 2	**BikeWise**	**4 (100%)**	500	GET--version-locations-markers---format-, GET--version-locations---format-	16	2	Any_Empty (2)				**CI**	**A**
Set 2	**BikeWise**		500						N_ReplaceNegative (2)		**CSD, PersE, DAO, SIND, CI, SVD**	**A**
Set 2	**BikeWise**		500	GET--version-incidents--id---format-	1	2			N_ReplaceTypeMax+1 (2) N_ReplaceTypeMin-1 (2)		**CSD, Ovf, CI, PersE, DAO, SIND, SVD**	**A**
Set 2	**BulkSMS**	**9 (82%)**	500	POST_/messages	5	1	**All** (1)	**All** (1)			**SSFM**	**A**
Set 2	**Cross Browser Testing**	**3 (60%)**	500	**3 operation**	19	3			N_ReplacePositive (3) N_ReplaceNegative (3)		**SSFM**	**A**
Set 2	**D7SMS**	**2 (67%)**	500	SendPost, SendbatchPost	8	2				S_ReplaceAlphanumeric (2)	**O**	**H**
Set 2	**Europeana**	**13 (100%)**	500	getProvider, listProviderDatasets, getDataset, getSingleRecordRDF, getSingleRecordJsonLD, getSingleRecordJson	23	10				S_ReplacePrintable (9) S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_Malicious (9)	**SSFM**	**A**
Set 2	**Europeana**		500	searchRecords, translateQueryUsingGET	21	2				S_AppendNonPrintable (2) S_ReplaceNonPrintable (2) S_InsertNonPrintable (2)	**PersE, DAO**	**A**
Set 2	**Figshare**	**117 (94%)**	500	POST_/account/institution/review/{curation_id}/comments	2	1			N_Add1 (1) N_Sub1 (1) N_Replace0 (1) N_Replace1 (1) N_Replace-1 (1) N_ReplaceTypeMax (1) N_ReplaceTypeMin (1) N_ReplaceTypeMin-1 (1) N_ReplaceDomainMin (1)		**SSFM**	**A**
Set 2	**Greenwire**	**6 (100%)**	500	GET_/volunteers, GET_/events, GET_/groups	7	3				S_ReplaceAlphanumeric (3)	**SSFM**	**A**
Set 2	**HHS Media Services**	**29 (100%)**	400	getLanguages, getSources, getCampaigns, getMediaByCampaignId	13	4				S_ReplaceAlphanumeric (4)	**PersE, DAO, SRD**	**A**
Set 2	**HHS Media Services**		400	getMediaByTagId	4	1			N_Replace0 (1)		**AO, DZ**	**A**
Set 2	**HHS Media Services**		400	getMediaByTagId	4	1			N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)		**CI, Ovf, ParsE, NR**	**A**
Set 2	**HHS Media Services**		400	getSources, getLanguages, getCampaigns, getMediaByCampaignId, getFeaturedMedia	16	5			N_ReplacePositive (5) N_ReplaceNegative (5) N_Add1 (5) N_Sub1 (5) N_Replace0 (5) N_Replace1 (5) N_Replace-1 (5) N_ReplaceTypeMax (5) N_ReplaceTypeMin (5) N_ReplaceTypeMax+1 (5) N_ReplaceTypeMin-1 (5)		**CI, Ovf, ParsE**	**A**
Set 2	**HHS Media Services**		400	getMedia	53	1			N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)		**CI, Ovf, ParsE**	**A**
Set 2	**HHS Media Services**		400	searchMedia	3	2	Any_Empty (2)				**CI**	**A**
Set 2	**Highways England**	**10 (100%)**	500	GET_/v{version}/sites/{site_Ids}	2	1				S_ReplaceAlphanumeric (1) S_ReplacePrintable (1)	**SSFM**	**A**
Set 2	**NeoWS**	**6 (86%)**	500	retrieveSentryRiskDataById, retrieveNearEarthObjectById, browseNearEarthObjects, retrieveSentryRiskData	7	6			N_ReplacePositive (4) N_ReplaceNegative (4)	S_ReplaceNonPrintable (2)	**SSFM**	**A**
Set 2	**NSIDC**	**3 (75%)**	500	facets, open search, id	20	6		D_Replace1998-4-31 (2) D_Replace1994-12-0 (1) D_Replace1993-8-32 (2)	N_ReplacePositive (2) N_ReplaceNegative (2)	S_ReplacePrintable (1) S_ReplaceAlphanumeric (1) S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_ReplaceNonPrintable (2) S_Malicious (1)	**SSFM**	**A**
Set 2	**Open Skills**	**12 (92%)**	500	GET_/jobs, GET_/skills	4	2			N_ReplaceTypeMax+1 (2)		**SSFM**	**A**
Set 2	**Rat Genome Database**	**89 (97%)**	500	getGeneBySymbolUsingGET, getGeneByRgdIdUsingGET, getQTLByRgdIdUsingGET, getStrainByRgdIdUsingGET, getExpressionForTaxonUsingGET, getAllelesForTaxonUsingGET, getPhenotypesForTaxonUsingGET, getVariantsForTaxonUsingGET, getGenesForLatestAssemblyUsingGET, getAffectedGenomicModelsUsingGET	9	10	Any_Empty (7)		N_ReplacePositive (3) N_ReplaceNegative (3) N_Add1 (3) N_Sub1 (3) N_Replace0 (3) N_Replace1 (3) N_Replace-1 (3) N_ReplaceTypeMax (3) N_ReplaceTypeMin (3)	S_ReplacePrintable (6) S_ReplaceAlphanumeric (6) S_AppendNonPrintable (6) S_InsertNonPrintable (6) S_ReplaceNonPrintable (7) S_Malicious (6)	**SRD**	**A**
Set 2	**Rat Genome Database**		500	getChartInfoUsingGET, getChartInfoUsingGET_1	5	2	Any_Empty (2)			S_ReplacePrintable (2) S_ReplaceAlphanumeric (2) S_AppendNonPrintable (2) S_InsertNonPrintable (2) S_ReplaceNonPrintable (2) S_Malicious (2)	**SRD, AOB**	**A**
Set 2	**Rat Genome Database**		500	getTermStatsUsingGET	2	1	Any_Empty (1)			S_ReplacePrintable (1) S_InsertNonPrintable (1) S_ReplaceNonPrintable (1)	**SRD, NR**	**A**
Set 2	**Rat Genome Database**		500	**24 operations**	62	3				S_ReplacePrintable (36) S_ReplaceAlphanumeric (36) S_AppendNonPrintable (36) S_InsertNonPrintable (36) S_Malicious (36)	**SRD, CI**	**A**
Set 2	**Rat Genome Database**		500	getAnnotationCountByAccIdAndObjectTypeUsingGET	4	1			N_ReplacePositive (1) N_ReplaceNegative (1) N_Sub1 (1) N_Replace0 (1) N_Replace1 (1) N_Replace-1 (1) N_ReplaceTypeMax (1) N_ReplaceTypeMin (1)		**PersE, DAO, SRD, CSD**	**A**
Set 2	**Rat Genome Database**		500	getGenesByKeywordUsingGET	2	1				S_Malicious (1)		**A**
Set 2	**Traccar**	**56 (92%)**	400	GET_/devices, GET_/commands/types, GET_/events/{id}	8	3	Any_Empty (2)	**All** (1)	N_ReplacePositive (1) N_ReplaceNegative (1) N_Add1 (1) N_Sub1 (1) N_Replace0 (1) N_Replace1 (1) N_Replace-1 (1) N_ReplaceTypeMax (1) N_ReplaceTypeMin (1) N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)		**NR**	**A**

as those that represent clear robustness problems. Please refer to [21] for a more detailed view of the results, including short textual descriptions of each observation as well as the full raw output of each test suite. In the following subsections, we detail the results for each one of the sets.

1) RESULTS OF SET 1

Starting with **Set 1 (Popular services)**, we found no error responses for the Google Calendar, Giphy, and Trello services. Regarding Google Drive, we detected one case, where the service replied a `500 internal server error`

status code and a JSON payload with `{error: {code: 500, message: null}}`. This was observed in the operation `drive.permissions.list` and in the operation `drive.revisions.list`, as a consequence of injecting the `S_ReplaceAlphanumeric` fault in the `pageToken` query parameter (present in both operations). The goal of the former operation is to list all the permissions set associated with a certain file, and the goal of the latter is to list all the revisions (i.e., the change history) that have been performed on a file. This case fits in *Hindering* failure mode, as the returned error response is not correct. Note that other problems in that particular operation were properly handled with 4xx status codes and responses specifying the problem.

Three of the Spotify service operations (see Table 3) produced a response with a 500 status code accompanied by the payload `{error: {status: 500, message: Server error}}`. This response was consistently triggered by the empty parameter fault, null replacement fault, the malicious string fault, and by appending or randomly inserting non-printable characters. We also observed `502 bad gateway` status code responses in three operations (one of which shared with the previous error response) accompanied by a message signaling that the server was unable to perform the intended action (i.e., either it *could not follow playlist*, *could not retrieve followers* or *could not unfollow playlist* [21]). The documentation of Spotify does not specify the expected behavior for these cases, but still the status code is anomalous, considering the remaining cases of messages returned by the service (including cases where the server detects a problem, in which 4xx codes are used).

2) RESULTS OF SET 2

We observed several different cases of robustness issues in **Set 2 (Public services)**, with most services displaying more than one type of problem, ranging from pure robustness issues to potential security problems. The next paragraphs discuss just a few different examples, with the full information being available at [21].

In a single operation of the Api2Pdf service, a response with a `400 Bad Request` status code was accompanied by a full stack trace, revealing the partial structure of the service source code, which is essentially a problem of information disclosure. Therefore, an experienced user could take advantage of this information for identifying entry points in the service, such as the use of libraries with known vulnerabilities. We tagged this behavior with *Data access operations* because the payload message refers to the inability to find a particular file, as well as *Server resource disclosure* due to application-specific information being revealed by the stack trace.

Another case of information disclosure was observed in two operations of the BikeWise service, with a full SQL query being included in the response payload and also specific information about the database management system being used, namely PostgreSQL (identified by the payload exception `PG::InvalidRowCountInLimitClause`). This occurred because the `limit` query parameter contained a negative number, and the service directly used this value in the `LIMIT` clause of a PostgreSQL query without prior validation. In addition, this operation actually treats this particular input as a string (instead of a number), which we could verify by replacing the input with a typical SQL Injection string that would work in the `LIMIT` clause (e.g., `10 UNION SELECT 1, 2, 3, 4, 5 --`), which was accepted by the operation, confirming it is vulnerable to SQL Injection. This behavior is marked with the tags *Command or schema disclosure*, *Persistence error* and *Data access operations* (i.e., the problem relates to a database exception), *System Vendor Disclosure* and *System instance name disclosure* (i.e., the service revealed the name of the database vendor and instance), and *conversion issues* (i.e., we detected that the service actually treats this parameter as the wrong data type).

Another operation of this service also disclosed a query due to mishandling an out of bounds 32-bit integer (e.g., minimum minus one or maximum plus one) in the `id` path parameter of the operation URI (`/v2/incidents/{id}`), and failed to convert its string representation (i.e., in the HTTP request) to a 32-bit integer space (i.e., *Conversion issues* and *Overflow*). Similarly to the previous case, the PostgreSQL-specific exception `PG::NumericValueOutOfRange` was also included in the error response. We were also able to confirm the operation is vulnerable to SQL Injection, by using a particular malicious string (i.e., `100 UNION SELECT 1, 2, 3 --`) that was accepted by the operation.

In several APIs, we observed problems that are obvious cases of missing or incomplete input validation. For instance, in the Rat Genome Database service, we observed one case of *array out of bounds* accompanied with the disclosure of a full stack trace (*server resource disclosure*), triggered by the injection of `Any_empty` fault and a few string faults in the `termString` parameter of the vulnerable operations (see Table 3). In the Traccar service, we observed a *null reference* in three operations after injecting multiple faults (e.g., `Any_Empty`, `N_Replace0`, `B_Overflow`) over the `deviceId`, `id` and `all` parameters. The returned response included a reference to a null pointer exception.

One operation of the HHS Media Services API returned an error after attempting to divide a number by zero (*Arithmetic operations* and *Division by zero*), as a consequence of the `N_Replace0` fault, which replaces a numeric input with the value zero. The same operation also produced an error response related with the attempt to convert a null pointer to a 32-bit integer. Further analysis revealed that the *null* pointer originated from the failure to parse a string representation of an out of bounds integer (e.g., integer minimum minus one).

3) RESULTS OF SET 3

In **Set 3 (Middleware management services)**, all robustness issues observed (during testing of the Docker Engine API) are associated with the `500 internal server`

TABLE 4. Results of the experimental evaluation for Sets 3-5.

Set	API	Operations tested	Status code	Operations with error responses	Parameters: Total	Parameters: Affected	Injected fault types: Any	Injected fault types: Array | Boolean	Injected fault types: Number	Injected fault types: String	Behaviour categorization	CRASH
Set 3	Docker Engine API	70 (66%)	500	BuildPrune, VolumePrune, NetworkList, ImagePrune, NodeList, PluginList, ConfigList, SecretList, TaskList, NetworkPrune, ImageSearch, ImageList, ContainerList	22	13				S_ReplacePrintable (13) S_ReplaceAlphanumeric (13) S_ReplaceNonPrintable (13) S_InsertNonPrintable (13) S_AppendNonPrintable (13) S_Malicious (13)	ParsE	A
				PluginUpgrade, PluginPull	8	2	Any_Empty (2)					A
				ContainerUpdate	35	1			N_ReplacePositive (1) N_ReplaceNegative (1)		ParsE, CI, Ovf, AOF	A
				ContainerList, NodeList	5	2				S_ReplaceAlphanumeric (2)	ParsE, CI	A
				NodeUpdate	6	1			N_ReplacePositive (1) N_ReplaceNegative (1)		SSFM	A
				ImageSearch, ContainerRestart, PluginEnable	7	3			N_Add1 (3) N_Sub1 (3) N_Replace1 (3) N_Replace0 (3) N_Replace-1 (3) N_ReplaceTypeMax (3) N_ReplaceTypeMin (3)		CI, ParsE, AOF	A
									N_ReplaceTypeMax+1 (3) N_ReplaceTypeMin-1 (3)		CI, Ovf, AOF, ParsE	A
Set 4	Kazoo Crossbar	45 (100%)	500	searchAvailableNumbers, listUsCityNumbers, fetchAccountByNumber	7	4	Any_Empty (3)		N_Replace0 (1) N_ReplaceTypeMin-1 (1)	S_ReplacePrintable (3) S_ReplaceAlphanumeric (1) S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_ReplaceNonPrintable (1) S_Malicious (1)	SSFM	A
				getLocalityOfNumbers	2	1		A_RemoveOne (1) A_RemoveAllButFirst (1)				A
				rebootDevice	2	1	Any_Empty (1)				DAO	A
				23 operations	39	26				S_ReplacePrintable (26)	O	H
			502	changeAccount	185	2	Any_Empty (1)			S_ReplaceNonPrintable (1)	SSFM	H
			503	loadUserDevices	2	1	Any_Empty (1)				DAO	A
Set 5	TPC-App (Vx0)	4 (100%)	400	newCustomer_Vx0, changePaymentMethod_Vx0	20	2	**All** (2)				NR	A
				newCustomer_Vx0	16	16	Any_Empty (13) Any_Null (16)		N_Add1 (1) N_Replace0 (1) N_Replace1 (1) N_ReplaceTypeMax (1) N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)	S_ReplaceAlphanumeric (6) S_ReplacePrintable (5) S_ReplaceNonPrintable (8) S_InsertNonPrintable (11) S_AppendNonPrintable (14) S_Malicious (16)	PersE, DAO	A
				newCustomer_Vx0, changePaymentMethod_Vx0, newProducts_Vx0	23	3	Any_Empty (1)		N_ReplaceTypeMax+1 (1)	S_ReplaceAlphanumeric (1)		A
				changePaymentMethod_Vx0	4	1				S_ReplaceAlphanumeric (1) S_ReplacePrintable (1) S_AppendNonPrintable (1) S_ReplaceNonPrintable (1) S_Malicious (1)		A
				newProducts_Vx0	3	1			N_Replace-1 (1) N_ReplaceTypeMin (1) N_ReplaceTypeMax+1 (1)			A
							Any_Empty (1)		N_ReplaceTypeMin-1 (1)		AOB	A
			500						N_ReplaceTypeMax (1) N_ReplaceTypeMin-1 (1)		AE, SRD	A
	TPC-App (VxA)	4 (100%)	400	newProducts_VxA, productDetail_VxA	4	2	Any_Empty (2)		N_ReplaceTypeMax (2)		AOB	A
							Any_Empty (2)				PersE, DAO	A
				newCustomer_VxA, newProducts_VxA	19	2				S_Malicious (2)		A
				newCustomer_VxA	16	16	Any_Empty (11) Any_Null (16)		N_Add1 (1) N_Replace1 (1) N_Replace-1 (1) N_ReplaceTypeMax (1) N_ReplaceTypeMin (1)	S_ReplaceAlphanumeric (9) S_ReplacePrintable (8) S_ReplaceNonPrintable (11) S_InsertNonPrintable (16) S_AppendNonPrintable (9) S_Malicious (16)		A
				changePaymentMethod_VxA, newCustomer_VxA	20	13				S_ReplacePrintable (3) S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_ReplaceNonPrintable (1) S_Malicious (11)		A
				changePaymentMethod_VxA	4	1	Any_Null (1)				NR	A
							Any_Empty (1)			S_ReplacePrintable (1) S_ReplaceAlphanumeric (1) S_Malicious (1)	PersE, DAO	A
				newProducts_VxA	3	1			N_Replace-1 (1) N_ReplaceTypeMin (1) N_ReplaceTypeMax+1 (1)			A
			500						N_ReplaceTypeMax (1) N_ReplaceTypeMin-1 (1)		AE, SRD	A
	TPC-C (Vx0)	5 (100%)	400	paymentTransaction_Vx0	8	1		B_Overflow (1)			CI, ParsE	A
				deliveryTransaction_Vx0	2	1				S_Malicious (1)	PersE, DAO	A
				newOrderTransaction_Vx0	8	2			N_ReplaceTypeMax+1 (2) N_ReplaceTypeMin-1 (2)		Ovf, CI, ParsE	A
						5		A_RemoveOne (3) A_RemoveAll (3) A_RemoveAllButFirst (3)	N_Add1 (1) N_Replace-1 (1) N_ReplaceTypeMin (1)		AOB	A
						3	**All** (3)				NR	A
			500	newOrderTransaction_Vx0	8	1			N_ReplaceTypeMax (1)		AE, SRD	A
	TPC-C (VxA)	5 (100%)	400	paymentTransaction_VxA	8	1		B_Overflow (1)			CI	A
				deliveryTransaction_VxA, newOrderTransaction_VxA, orderStatusTransaction_VxA, paymentTransaction_VxA, stockLevelTransaction_VxA	26	26	Any_Empty (2) Any_Null (1)			S_Malicious (26)	PersE, DAO	A
				deliveryTransaction_VxA	2	1				S_Malicious (1)		A
				orderStatusTransaction_VxA, paymentTransaction_VxA	13	5				S_Malicious (5)		A
				newOrderTransaction_VxA	8	2			N_ReplaceTypeMax+1 (2) N_ReplaceTypeMin-1 (2)		Ovf, CI	A
						5		A_RemoveOne (3) A_RemoveAll (3) A_RemoveAllButFirst (3)	N_Add1 (1) N_Replace-1 (1) N_ReplaceTypeMin (1)		AOB	A
						3	**All** (3)				NR	A
			500			1			N_ReplaceTypeMax (1)		AE, SRD	A

`error` status code (refer to Table 4). A common issue is the failure to process or parse certain types of inputs introduced by several different faults like `Any_Empty`, `N_Replace-1` or `S_AppendNonPrintable`. These cases, marked with the *Parsing error* tag, resulted in a message holding `could not parse filters:`

invalid character '<char>'. Notice that we found many other cases of invalid inputs resulting in a proper 4xx code being returned. The fact that a 5xx code is being returned strongly suggests that the usual validation before parsing is either not being carried out, or is being carried out incorrectly, resulting in a failed attempt to parse inputs. This inconsistent behavior should be made uniform. A similar case occurs with the PluginUpgrade and PluginPull operations, which fail with a 500 code whenever a JSON array in the request payload is replaced with an empty value (but fail with 4xx codes, when the array is null or carries other types of faults).

In the ContainerUpdate operation, we detected a mismatch between the service specification and the service itself. A long integer in field BlkioWeight makes the service fail with a 500 status code mentioning that it was not able to convert the value into a 16-bit unsigned integer. Again, this suggests that no validation is performed before attempting parsing. The problem with this kind of issue is that a client may not be expecting this kind of server failure (as the specification does not state the exact datatype). Similar conversion issues were found in a few other operations (e.g., NodeUpdate, ImageSearch, ContainerRestart).

4) RESULTS OF SET 4

Regarding **Set 4 (Private services)**, there are some interesting cases to mention. For instance, calls to the operation rebootDevice using the empty fault on the DEVICE_ID path parameter resulted in a 500 internal server error status code, with a payload message containining datastore fault (marked with tag *Data access operations*). Similarly, this same fault in the USER_ID path parameter of the loadUserDevices operation resulted in the status code 503 service unavailable, accompanied by the error message datastore fatal error, indicating a severe problem has occurred at the datastore level.

In the changeAccount operation the empty value and the random non-printable string triggered a 502 bad gateway status code, accompanied by an HTML payload. Note that this API is purely JSON-based (i.e., no HTML responses should ever be returned to clients) and this type of response typically originates from middleware that supports these services. In some other cases, responses were vague (e.g., *init failed*) or carried no payload at all (triggered by the S_ReplacePrintable fault).

We contacted the developers, reported the issues and received feedback. The developers confirmed that all reported cases of responses holding 5xx codes are indeed problems in the service that must be corrected, which emphasizes the ability of the tool to detect issues that, in this case, have been neglected by verification activities put in place from the experienced developers in charge of the service.

5) RESULTS OF SET 5

The tests involving **Set 5 (In-house services)** resulted in the disclosure of several robustness problems, from which we present some highlights. In ***TPC-App***, calls to the New products operation in both Vx0 and VxA implementations that use negative values or the maximum plus one on the itemLimit parameter resulted in an error message stating a call to setFetchSize (a parameter that specifies the number of rows to be fetched from the database) using invalid arguments. What is interesting is that this is a function of the query parsing middleware and the Oracle JDBC driver used (v10.2.0.2.0), does not validate the input of this function against negative or out of bounds values, a known bug, which has been fixed (identified as bug report CORE-2130 in liquibase.jira.com [73]). This case gives emphasis to the fact that robustness tests can be used to detect bugs not only in the implementation logic of a certain service, but also at the middleware level.

In the same two previous operations, using the data type maximum and minimum minus one, faults in the itemLimit parameter resulted in a 500 status code with the message OutOfMemoryError: Requested array size exceeds VM limit followed by a full stack trace (*Server resource disclosure*). The value is used (without validation) to create an array, which exceeds the amount of available memory and fails unexpectedly (according to the specification).

We also observed differences between what the specification states and the actual implementation, namely the use of strings to represent numbers. For instance, in the Vx0 Change payment method operation, the wrong representation of the poId argument as a String, led some faulty requests to trigger a failure where the Oracle driver tried to parse a number from a database query string. This means that the injected invalid string reached the query (and could have been gracefully stopped at the entry of the operation).

The TPC-App VxA version is known to hold six vulnerabilities, which our tool was also able to detect, despite not being its main target. The large set of malicious faults used were able to modify the structure of all vulnerable queries (check detailed results at [21]). This was evident at the client side, when it has received responses, such as ORA-00933 SQL command is not properly ended.

Regarding ***TPC-C*** implementation, we observed similar issues to the ones presented earlier. Namely misrepresentation of datatypes, for instance in the Payment operation of both Vx0 and VxA a boolean was misrepresented as string, leading the service to abort the execution of the operation with an unexpected response (failure to parse a boolean). Typical robustness issues, such as unhandled exceptions due to array index out of bounds or null pointers were also observed. All these issues could be easily avoided with proper validation of inputs.

Like in the previous benchmark, we again observed an *out of memory error*. However, in this case it was actually a data structure (i.e., an array) placed directly on the service code (and not an internal JDBC driver data structure). The code tried to instantiate an array by directly using the user

input, allowing for a malicious initialization with a very large number that led to the unhandled exception.

In the VxA version, which builds queries through concatenation, several Oracle errors were triggered by the query parsing middleware, including `ORA-00907` which states a parenthesis is missing, and `ORA-01756` where a quoted string is not properly terminated.

In the VxA version of TPC-C, we successfully detected 28 of the 30 known SQL injection vulnerabilities. Our tool could not reach the two undetected cases (present in the `New Order` operation) because, in the `New Order` operation code, they are preceded by an `INSERT` statement. As the selection of the `S_Malicious` faults is random, the tool simply selected cases that always broke the syntax of the `INSERT` statement, leading the operation to abort with an unexpected exception and not allowing to reach the two remaining vulnerable queries.

C. MAIN FINDINGS

This section highlights the main findings of our experimental evaluation by going through key aspects, which we identified during this work. The main findings are the following:

- REST services are being made available on-line, carrying residual bugs that affect the overall robustness of the services.
- The robustness tests were able to disclose bugs that reside at the service implementation level, but also at the middleware that supports the service.
- Robustness tests were able to detect security issues, where malicious inputs trigger issues related with missing validation or wrong input usage, as it is the case of SQL Injection vulnerabilities.
- A common security issue found was related with information disclosure, namely code structure, SQL commands and database structures or database vendor. A malicious user may take advantage of the information to explore entry points or exploit know vulnerabilities in the system.
- The faults that were most frequently involved in the detection of robustness problems were the injection of null and empty values and also string-related faults. In this latter case, faults with random characters and also malicious (SQL Injection) revealed to be quite effective.
- Considering the whole set of services tested, we frequently observed services being affected by problems related to storage operations, null references, and conversion issues.
- Contrary to what was found in previous work regarding SOAP web services [24], the large number of null/empty value faults that triggered robustness issues, did not actually directly led to the disclosure of *null references* problems. Either they triggered other kinds of problems (e.g., *Data Access Operations*), or they triggered issues that were camouflaged by the services and resulted in vague responses being delivered to the client.
- The experiments revealed only Abort and Hindering failures. Considering the large amount of executed tests, this means that severe failure modes like Catastrophic [22] seem to be difficult to trigger.
- Mismatches between the interface description and the actual service implementation were detected during the analysis of the tests results, emphasizing the ability of the tests to flag such cases.
- It became evident that current OpenAPI specifications associated with the services being tested are being written without attention to basic operation details (e.g., missing data type details) and several detected cases turned out to be associated with robustness problems.
- Most of the OpenAPI specifications analysed lack complete information regarding the expected behavior of the service (e.g., when in presence of invalid inputs), which opens space for doubts when analyzing tests results and create issues for applications that wish integrate these services.
- In almost half of the services tested, we found non descriptive error messages which, accompanied with a poor specification, do not allow clients to get much insight regarding the real issues.
- Access to server logs was not sufficient information to allows us to understand the exact root cause of the problems detected with the Docker Engine service.
- Robustness testing revealed useful even in services with high reliability requirements (i.e., Set 1 and Set 4), being able to detect issues that had escaped the verification activities in place.
- Missing validation is the main cause for the detected problems in the four TPC implementations (although some related with poor practices, like query concatenation with user input). While some cases should be obvious to avoid by senior programmers (e.g., using prepared statements), others would be difficult to detect (e.g., the use of a driver holding a bug).
- Robustness testing results seem to be highly repeatable, at least considering the set of services tested. Each fault was repeated three times per operation parameter and, on average, in about 2.8 times we observed the same outcome.

V. THREATS TO VALIDITY

In this section, we present the main threats to the validity of this work and discuss mitigation strategies. We start by mentioning the fact that *we manually analyzed 399,001 results*, which may have added some error to the process (e.g., cases of undetected failures). We focused our attention on the obvious or highly suspicious cases of robustness problems. *The identification of robustness issues can be subjective*, which is especially true when the service specification lacks sufficient information. To reduce the likelihood of error, we had these cases double checked by an Experienced Researcher.

The *tests over the public services were carried out without isolation for other concurrent* requests (i.e., from other users).

Even so, we repeated the injection of faults three times to observe that on average, in 2.8 times the outcomes were the same.

The robustness testing tool may hold residual software defects, which might have affected some observations (e.g., by not emulating a certain fault as expected). We tried to carry out typical V&V tasks throughout the development of the tool, and we placed it in contact with diverse service APIs (i.e., with different number of operations, parameters and payload requirements) so that any issue would become evident. In the end, when analysing the results of the tests we checked the requests that were involved in each failure to verify if the tool had injected the expected fault.

Finally, the *tool configuration used during the tests*, namely the number of faults to generate for a certain parameter (some faults involve random value generation), the random workload involved (which may not provide sufficient coverage, or may provide a biased coverage) may have led us to disclose fewer issues in some services, showing an image of the overall robustness of services that is not really representative of reality. We did try to run a relatively large number of tests (i.e., especially considering the manual time-consuming steps of the experiments) and we acknowledge that there may be many other issues that we were not able to disclose. Nevertheless, the diversity and number of disclosed issues provide us and developers with useful information for robustness assessment and that can be used by providers to improve their services.

VI. CONCLUSION

In this paper, we presented an approach and a tool for carrying out robustness tests on REST APIs. We introduced the concepts behind our approach, and described bBOXRT architecture, mapping its components to the different phases of a typical robustness testing campaign. We showcased the capabilities of bBOXRT tool in the form of a practical experiment, by performing tests over 52 REST services grouped in 5 different sets. Results showed the ability of the tool to test different kinds of services and disclose robustness and also security issues across nearly half of the tested services.

We are actively improving the tool by fixing any underlying issues. As future work, we intend to improve upon its extensibility in order to ease the process of adding new functionalities (useful for researchers and practitioners alike). We also intend to add support for additional payload types (e.g., files, media types other than JSON) as well as implement faults that specialize in the middleware that supports REST services. Finally, we will also consider studying the applicability of Machine Learning algorithms to automate the process of classifying test results.

REFERENCES

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Inst. Softw. Res., Univ. California, Irvine, CA, USA, 2000.

[2] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST Web service APIs," *IEEE Trans. Services Comput.*, early access, Jun. 14, 2018, doi: 10.1109/TSC.2018.2847344 .

[3] T. L. Foundation. (2016). *OpenAPI Initiative*. Accessed: May 2, 2020. [Online]. Available: https://www.openapis.org/

[4] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12-1990, Dec. 1990. [Online]. Available: http://standards.ieee.org

[5] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. SRDS, 16th IEEE Symp. Reliable Distrib. Syst.*, Oct. 1997, pp. 72–79.

[6] P. Koopman and J. DeVale, "Comparing the robustness of posix operating systems," in *Proc. 29th Annu. Int. Symp. Fault-Tolerant Comput.* Washington, DC, USA: IEEE Computer Society, 1999, p. 30. [Online]. Available: http://dl.acm.org/citation.cfm?id=795672.796953

[7] R. Kaksonen, M. Laakso, and A. Takanen, *Software Security Assessment Through Specification Mutations and Fault Injection*. Boston, MA, USA: Springer, 2001, pp. 173–183.

[8] F. Saad-Khorchef, A. Rollet, and R. Castanet, "A framework and a tool for robustness testing of communicating software," in *Proc. ACM Symp. Appl. Comput. (SAC)*. New York, NY, USA: ACM, 2007, pp. 1461–1466.

[9] Y. Fu and O. Kone, "Security and robustness by protocol testing," *IEEE Syst. J.*, vol. 8, no. 3, pp. 699–707, Sep. 2014.

[10] C. P. Dingman, J. Marshall, and D. P. Siewiorek, "Measuring robustness of a fault tolerant aerospace system," in *25th Int. Symp. Fault-Tolerant Computing. Dig. Papers*, Jun. 1995, pp. 522–527.

[11] H.-N. Chu, J. Arlat, M.-O. Killijian, B. Lussier, and D. Powell, "Robustness testing of robot controller software," in *Proc. 12th Eur. Workshop Dependable Comput. (EWDC)*, 2009, p. 2.

[12] F. Mattiello-Francisco, E. Martins, A. R. Cavalli, and E. T. Yano, "InRob: An approach for testing interoperability and robustness of real-time embedded software," *J. Syst. Softw.*, vol. 85, no. 1, pp. 3–15, Jan. 2012.

[13] Z. Micskei, I. Majzik, and F. Tam, "Robustness testing techniques for high availability middleware solutions," in *Proc. Int. Workshop Eng. Fault Tolerant Syst. (EFTS)*, 2006, p. 14.

[14] D. R. Azevedo, A. M. Ambrosio, and M. Vieira, "HLA middleware robustness and scalability evaluation in the context of satellite simulators," in *Proc. IEEE 19th Pacific Rim Int. Symp. Dependable Comput.*, Dec. 2013, pp. 312–317.

[15] W. Cardoso, E. Martins, N. Laranjeiro, and N. Antunes, "Combining state and interface–based robustness testing for OpenStack components," in *Proc. 9th Latin-Amer. Symp. Dependable Comput. (LADC)*, Nov. 2019, pp. 1–10.

[16] J. Camara, R. D. Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness evaluation of controllers in self-adaptive software systems," in *Proc. 6th Latin-Amer. Symp. Dependable Comput.*, Apr. 2013, pp. 1–10.

[17] N. Mendes, J. Duraes, and H. Madeira, "Evaluating and comparing the impact of software faults on Web servers," in *Proc. Eur. Dependable Comput. Conf.*, Apr. 2010, pp. 33–42.

[18] S. H. Kuk and H. S. Kim, "Robustness testing framework for Web services composition," in *Proc. IEEE Asia–Pacific Services Comput. Conf. (APSCC)*, Dec. 2009, pp. 319–324.

[19] S. Salva and I. Rabhi, "Stateful Web service robustness," in *Proc. 5th Int. Conf. Internet Web Appl. Services*, May 2010, pp. 167–173.

[20] S. Ilieva, D. Manova, I. Manova, C. Bartolini, A. Bertolino, and F. Lonetti, "An automated approach to robustness testing of BPEL orchestrations," in *Proc. IEEE 6th Int. Symp. Service Oriented Syst. (SOSE)*, Dec. 2011, pp. 193–203.

[21] N. Laranjeiro, J. Agnelo, and J. Bernardino. (2020). *bBOXRT Tool and Results*. [Online]. Available: https://eden.dei.uc.pt/~cnl/papers/2020-access.zip

[22] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Dig. Papers. 28th Annu. Int. Symp. Fault-Tolerant Comput.*, Washington, DC, USA: IEEE Computer Society, Jun. 1998, p. 230.

[23] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–55, Feb. 2016, doi: 10.1145/2841425.

[24] N. Laranjeiro, M. Vieira, and H. Madeira, "A robustness testing approach for SOAP Web services," *J. Internet Services Appl.*, vol. 3, no. 2, pp. 215–232, Sep. 2012.

[25] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. L. Goues, and P. Koopman, "Robustness testing of autonomy software," in *Proc. 40th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, May 2018, pp. 276–285.

[26] D. Katz, M. Zizyte, C. Hutchison, D. Guttendorf, P. E. Lanigan, E. Sample, P. Koopman, M. Wagner, and C. Le Goues, "Robustness inside out testing," in *Proc. 50th Annu. IEEE-IFIP Int. Conf. Dependable Syst. Netw.-Supplemental*, Jul. 2020, pp. 1–4.

[27] C. P. Shelton, P. Koopman, and K. Devale, "Robustness testing of the microsoft Win32 API," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*. Washington, DC, USA: IEEE Computer Society, Dec. 2000, p. 261.

[28] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proc. 1st Int. Workshop Random Test. (RT)*. New York, NY, USA: Association Computing Machinery, 2006, pp. 46–54, doi: 10.1145/1145735.1145743.

[29] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proc. Joint Int. Workshop Dyn. Anal. (WODA) Softw. Syst. Perform. Test., Debugging, Analytics (PERTEA)*. New York, NY, USA: Association Computing Machinery, 2014, pp. 1–5, doi: 10.1145/2632168.2632169.

[30] H. Feng and K. G. Shin, "Bindercracker: Assessing the robustness of Android system services," *CoRR*, vol. abs/1604.06964, pp. 1–14, Apr. 2016. [Online]. Available: http://arxiv.org/abs/1604.06964

[31] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a COTS system for space applications," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2002, pp. 325–330.

[32] N. Laranjeiro, M. Vieira, and H. Madeira, "Experimental robustness evaluation of JMS middleware," in *Proc. IEEE Int. Conf. Services Comput.*, vol. 1, Jul. 2008, pp. 119–126.

[33] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott, "Robustness testing of Java server applications," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 292–311, Apr. 2005.

[34] J. Duraes and H. Madeira, "Emulation of software faults by educated mutations at machine-code level," in *Proc. 13th Int. Symp. Softw. Rel. Eng.*, Nov. 2002, pp. 329–340.

[35] F. Belli, A. Hollmann, and W. E. Wong, "Towards scalable robustness testing," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, Jun. 2010, pp. 208–216.

[36] H. A.-H. Ahmad, Y. Sedaghat, and M. Moradiyan, "LDSFI: A lightweight dynamic software-based fault injection," in *Proc. 9th Int. Conf. Comput. Knowl. Eng. (ICCKE)*, Oct. 2019, pp. 207–213.

[37] M. N. Bennani and D. A. Menasce, "Assessing the robustness of self-managing computer systems under highly variable workloads," in *Proc. Int. Conf. Autonomic Comput.*, May 2004, pp. 62–69.

[38] M. I. P. Salas, P. L. D. Geus, and E. Martins, "Security testing methodology for evaluation of Web services robustness–case: XML injection," in *Proc. IEEE World Congr. Services*, Jun. 2015, pp. 303–310.

[39] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for OS robustness evaluation," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 502–511.

[40] D. Cotroneo, A. K. Iannillo, R. Natella, and S. Rosiello, "Dependability assessment of the Android OS through fault injection," *IEEE Trans. Rel.*, early access, Dec. 3, 2019, doi: 10.1109/TR.2019.2954384.

[41] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing Android native system services via automated interface analysis," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2020, pp. 307–323. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/liu

[42] E. Barsallo Yi, A. Maji, and S. Bagchi, "How reliable is my wearable: A fuzz testing-based study," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 410–417.

[43] E. B. Yi, H. Zhang, A. K. Maji, K. Xu, and S. Bagchi, "Vulcan: Lessons on reliability of wearables through state-aware fuzzing," in *Proc. 18th Int. Conf. Mobile Syst., Appl., Services* New York, NY, USA: Association Computing Machinery, Jun. 2020, pp. 391–403, doi: 10.1145/3386901.3388916.

[44] J. Agnelo, "A robustness testing approach for RESTful Web services," M.S. thesis, Dept. Inform. Eng., Fac. Sci. Technol., Univ. Coimbra, Coimbra, Portugal, Sep. 2020. [Online]. Available: https://estudogeral.sib.uc.pt/handle/10316/92292

[45] H. Ed-douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic generation of test cases for REST APIs: A specification-based approach," in *Proc. IEEE 22nd Int. Enterprise Distrib. Object Comput. Conf. (EDOC)*, Oct. 2018, pp. 181–190.

[46] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated black-box testing of RESTful APIs," in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 142–152.

[47] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-box constraint-based testing of RESTful Web APIs," in *Service-Oriented Computing* (Lecture Notes in Computer Science), E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, Eds. Cham, Switzerland: Springer, 2020, pp. 459–475.

[48] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 131–141.

[49] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Trans. Softw. Eng. Methodology*, vol. 28, no. 1, pp. 1–37, Feb. 2019, doi: 10.1145/3293455.

[50] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of RESTful Web APIs," in *Proc. 40th Int. Conf. Softw. Eng.* New York, NY, USA: ACM, May 2018, p. 882, doi: 10.1145/3180155.3182528.

[51] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API fuzzing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 748–758.

[52] P. V. P. Pinheiro, A. T. Endo, and A. Simao, "Model-based testing of restful Web services using UML protocol state machines," in *Proc. Brazilian Workshop Systematic Automated Softw. Test.*, 2013, pp. 1–10.

[53] T. Fertig and P. Braun, "Model-driven testing of RESTful APIs," in *Proc. 24th Int. Conf. World Wide Web*. New York, NY, USA: Association Computing Machinery, May 2015, pp. 1497–1502, doi: 10.1145/2740908.2743045.

[54] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for REST APIs," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 312–323, doi: 10.1145/3395363.3397374.

[55] H. Johan. (2010). *REST Assured*. Accessed: May 2, 2020. [Online]. Available: http://rest-assured.io/

[56] Apache Software Foundation. (1998). *Apache JMeter*. Accessed: May 2, 2020. [Online]. Available: https://jmeter.apache.org/

[57] Progress Software Corporation. (2003). *Fiddler*. Accessed: May 2, 2020. [Online]. Available: https://www.telerik.com/fiddler

[58] SmartBear Software. (2020). *OpenAPI Specification Version 3.0.3*. [Online]. Available: https://swagger.io/specification/

[59] RAML Workgroup. (2016). *RAML Specification Version 1.0*. [Online]. Available: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md

[60] Ismail Tasdelen. (Oct. 2019). *SQL Injection Payload List*. [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list

[61] E. Wittern, A. T. T. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young, and A. A. Slominski, "Opportunities in software engineering research for Web API consumption," in *Proc. IEEE/ACM 1st Int. Workshop API Usage Evol. (WAPI)*, May 2017, pp. 7–10.

[62] B. A. Sanchez, K. Barmpis, P. Neubauer, R. F. Paige, and D. S. Kolovos, "Restmule: Enabling resilient clients for remote APIs," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 537–541.

[63] Y. W. Kim, M. P. Consens, and O. Hartig, "An empirical analysis of GraphQL API schemas in open code repositories and package registries," in *Proc. AMW*, 2019, pp. 1–5. [Online]. Available: http://ceur-ws.org/Vol-2369/short04.pdf

[64] Docker Inc. (2019). *Docker Engine API Reference Version 1.40*. [Online]. Available: https://docs.docker.com/engine/api/v1.40/

[65] Transaction Processing Performance Council (TPC). (Feb. 2008). *TPC Benchmark App Specification Version 1.3*. [Online]. Available: http://www.tpc.org

[66] Transaction Processing Performance Council (TPC). (Feb. 2010). *TPC Benchmark C Specification v5.11*. [Online]. Available: http://www.tpc.org

[67] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[68] T. Topinka. (Apr. 2020). *Statistic Plugin*. [Online]. Available: https://plugins.jetbrains.com/plugin/4509-statistic

[69] JetBrains. (Sep. 2019). *IntelliJ IDEA*. [Online]. Available: https://www.jetbrains.com/idea/
[70] K. Toda. (Apr. 2020). *SpotBugs*. [Online]. Available: https://spotbugs.github.io/
[71] P. Arteau. (Oct. 2019). *Find Security Bugs Plugin*. [Online]. Available: https://find-sec-bugs.github.io/
[72] S. Furuhashi. (2008). *MessagePack*. [Online]. Available: https://msgpack.org/
[73] T. Becker. (Nov. 2014). *Oracle JDBC Driver bug report CORE-2130*. [Online]. Available: https://liquibase.jira.com/browse/CORE-2130

NUNO LARANJEIRO (Member, IEEE) received the Ph.D. degree from the University of Coimbra, Portugal, where he is currently an Assistant Professor. His research interests include robustness of software services as well as experimental dependability evaluation, interoperability, security, and enterprise integration. He contributed, as an author and a reviewer, to leading conferences and journals in the dependability and services computing areas. He has participated in several national and international projects.

JOÃO AGNELO received B.Sc. degree in informatics engineering from the ISEC—Polytechnic Institute of Coimbra and the M.Sc. degree in informatics engineering from the University of Coimbra. His research interests include dependability, artificial intelligence, and computer graphics.

JORGE BERNARDINO (Member, IEEE) received the Ph.D. degree from the University of Coimbra, in 2002. From 2005 to 2010, he was the President of ISEC. From 2017 to 2019, he was also the President of ISEC Scientific Council. In 2014, he was a Visiting Professor at CMU. He is currently a Coordinator Professor with the Polytechnic of Coimbra—ISEC, Portugal. He has authored more than 200 publications in refereed conferences and journals. His main research interests include big data, NoSQL, data warehousing, dependability, and software engineering. He participated in several national and international projects. He is a member of ACM. He is currently the Director of the Applied Research Institute (i2A), Polytechnic of Coimbra.