João André Nunes Agnelo

A ROBUSTNESS TESTING APPROACH FOR RESTful WEB SERVICES

1 2 9 0 UNIVERSIDADE D COIMBRA

UNIVERSIDADE D
COIMBRA

João André Nunes Agnelo

A ROBUSTNESS TESTING APPROACH FOR RESTful WEB SERVICES

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering, advised by Prof. Nuno Laranjeiro and Prof. Jorge Bernardino and presented to
Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2020

Faculty of Sciences and Technology

Department of Informatics Engineering

A robustness testing approach for RESTful Web Services

João André Nunes Agnelo

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Nuno Laranjeiro and Prof. Jorge Bernardino and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2020

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Abstract

Robustness is the degree to which a certain system or component can operate correctly in the presence of invalid inputs or stressful environmental conditions. With the increasing complexity and wide-spread use of software systems, obtaining assurances regarding their robustness has become of vital importance. This is especially true in mission- or business-critical systems where a failure may have severe consequences on the business or in human lives. Web services, in particular, are often exposed to abnormal and malicious input that could lead to serious issues, such as loss of data and information disclosure.

Most modern web companies, including Google, Instagram, Spotify and Slack, are supported by REST services (also known as REST APIs), a type of software system that follows the relatively loose REST architectural style. 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, which may result in financial or reputation losses (e.g., information disclosure). Robustness is, therefore, a key property in REST services.

In this dissertation, we present a systematic literature review on software robustness testing, and an approach and tool (named bBOXRT) for performing robustness tests over REST services. We begin by performing a comprehensive analysis, in the form of a systematic literature review, of the state of the art on software robustness testing, which we complement with an overview of the related work on REST API testing. This allows us to show that multiple techniques and tools for robustness assessment have been thoroughly studied and applied to a large diversity of domains, but REST services still lack practical approaches that specialize in robustness evaluation. We fill-in this gap by proposing bBOXRT, a tool for testing the robustness of REST services solely based on minimal information present in their interface descriptions.

We used bBOXRT to evaluate an heterogeneous set of 52 REST services that comprise 1351 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 online holding software defects that harm service integration, and also carrying security vulnerabilities that can be exploited by malicious users.

Keywords

Software testing, software robustness, robustness evaluation, web services, RESTful web services, REST API

This page is intentionally left blank.

Resumo

Robustez é o grau com que um certo sistema ou componente pode operar corretamente na presença de entradas inválidas ou condições de stress. Com o aumento da complexidade e o uso alargado de sistemas de software, obter garantias quanto à sua robustez tornou-se uma tarefa essencial. Isto é especialmente verdade nos sistemas críticos (quanto à sua missão ou ao seu objetivo de negócio) onde uma falha pode ter consequências graves no negócio ou em vidas humanas. Os serviços web, em particular, são frequentemente expostos a entradas inesperadas ou maliciosas que podem levar a problemas graves, tais como perda de dados e divulgação de informação.

A maioria das empresas web modernas, incluindo a Google, o Instagram, o Spotify e o Slack, são suportadas por serviços REST (também conhecidos como APIs REST), um tipo de sistema de software que segue o estilo arquitetural REST. Neste tipo de cenários, a heterogeneidade é dominante e o software é, por vezes, exposto a condições inesperadas que poderão ativar bugs residuais, levando à falha do serviço, o que poderá resultar em perdas financeiras ou de reputação (e.g., divulgação de informação). A robustez é, como tal, uma propriedade essencial em serviços REST.

Nesta tese, apresentamos uma revisão sistemática da literatura sobre testes de robustez em software, e uma abordagem e uma ferramenta (bBOXRT) para a execução de testes de robustez em serviços REST. Começamos por realizar uma análise exaustiva, sob a forma de uma revisão sistemática da literatura, do estado da arte sobre testes de robustez em software, que complementamos com uma visão geral do trabalho relacionado sobre testes em APIs REST. Isto permite-nos mostrar que, embora várias técnicas e ferramentas para avaliação de robustez já tenham sido estudadas em detalhe e aplicadas a uma grande variedade de domínios, os serviços REST ainda necessitam de abordagens práticas especializadas na avaliação de robustez. Propomos preencher esta lacuna com a bBOXRT, uma ferramenta para testar a robustez de serviços REST que necessita apenas de uma pequena fração da informação presente nas suas descrições de interface.

A bBOXRT foi usada para avaliar um conjunto heterogéneo de 52 serviços REST que englobam 1351 operações e se inserem em diferentes categorias (e.g., públicos, privados, *in-house*). Com esta avaliação, conseguimos revelar vários tipos de problemas de robustez, incluindo problemas em serviços com elevada necessidade de confiabilidade e também algumas vulnerabilidades de segurança. Os resultados mostram que os serviços REST estão a ser postos em operação online contendo ainda defeitos de software que prejudicam a sua integração, e também apresentam vulnerabilidades de segurança que podem ser exploradas por utilizadores maliciosos.

Palavras-Chave

Testes de software, robustez de software, avaliação de robustez, serviços web, serviços web RESTful, API REST

This page is intentionally left blank.

Acknowledgments

This work has been partially 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 METRICS, funded by the Portuguese Foundation for Science and Technology (FCT) – agreement No. POCI-01-0145-FEDER-032504; by the project MobiWise: From mobile sensing to mobility advising (reference: P2020 SAICT-PAC/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); 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).

Regarding personal acknowledgments, I would like to thank my MSc thesis advisors Prof. Nuno Laranjeiro and Prof. Jorge Bernardino for their endless patience and wisdom in this long-lasting endeavor. This work has given me much knowledge regarding how to formally approach a research proposal, and it has made me grow as an engineer. One particular takeaway from this work, is that in the future I must apply greater effort to the planning stage of a large project such as this, as time is the most valuable resource.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

API Application Programming Interface. 1, 2, 6–8, 22–25, 27, 34, 37, 46, 59–61, 64, 65, 67–72, 74–83, 85, 86, 107, 130, 131

bBOXRT black-BOX tool for Robustness Testing of rest services. 2, 3, 85, 86, 89, 92–94, 107, 131

BPEL Business Process Execution Language. 52, 54, 74, 80

CORBA Common Object Request Broker Architecture. 40, 42

COTS Commercial Off-The-Shelf. 19, 20, 22, 24, 33, 42, 43, 46, 48, 59

DBMS Database Management System. 43

DDS Data Distribution Service. 41, 42

DoS Denial-of-Service. 26, 41

DPI Driver Programming Interface. 24, 25

DSL Domain-Specific Language. 75, 78

FSM Finite State Machines. 29, 31, 37

FTP File Transfer Protocol. 74

GUI Graphical User Interface. 20, 47, 77

HA High Availability. 40, 42

HLA High Level Architecture. 39–42

HTML Hypertext Markup Language. 5, 49, 53, 100

HTTP Hypertext Transfer Protocol. 1, 2, 5–8, 19, 28, 49, 54, 67–80, 82, 85–87, 98

IaaS Infrastructure as a Service. 41

IOLTS Input-Output Labelled Transition System. 29–31, 35

IPC Inter-Process Communication. 25, 26, 33

JMS Java Message Service. 38, 40–42

JSON JavaScript Object Notation. 1, 5–8, 75–79, 82, 85–87, 96, 98, 100

LEAP Localized Encryption and Authentication Protocol. 30, 31

LoC Lines of Code. 93

LTS Labelled Transition System. 35

MACD Move, Add, Change, Delete. 27, 31, 40, 62, 63

NAS Non-Access Stratum. 30, 31

OData Open Data Protocol. 6

ORB Object Request Broker. 40

POSIX Portable Operating System Interface. 22, 23, 25, 27

QoS Quality of Service. 8, 41, 55

RADIUS Remote Authentication Dial-In User Service. 31

RAML RESTful API Modeling Language. 6, 74, 76, 80, 82, 86

REST Representational State Transfer. 1–3, 5–8, 11, 55, 59, 65, 67–72, 74–83, 85, 86, 89, 93, 101, 107, 130–132

RPC Remote Procedure Calls. 26, 40

RSDL RESTful Service Description Language. 6

RTI Run-Time Infrastructure. 39, 40

RTOS Real-Time Operating System. 33, 35–37

SIP Session Initiation Protocol. 19, 30, 31

SMTP Simple Mail Transfer Protocol. 28, 74

SOAP Simple Object Access Protocol. 1, 2, 19, 49–55, 59, 61, 74, 75, 77, 78, 80, 89, 102

SQL Structured Query Language. 2, 87, 93, 94, 98, 101, 102

STS Symbolic Transition System. 53, 54

TA Timed Automata. 31, 35, 37

TIOA Timed Input-Output Automata. 34, 36

TPC Transaction Processing Performance Council. 2, 102

TTCN-3 Testing and Test Control Notation-3. 29–31

UML Unified Modeling Language. 42, 46, 47, 69

URI Uniform Resource Identifier. 1, 2, 5–7, 69, 76, 78, 79, 82, 85–87

URL Uniform Resource Locator. 65, 73, 75–78, 86

V&V Verification and Validation. 1, 105

WADL Web Application Description Language. 69, 70, 74, 80

WAP Wireless Application Protocol. 27, 30, 31

WSDL Web Services Description Language. 1, 49–53, 55, 74, 80, 89

WSN Wireless Sensor Network. 30, 31

WWW World Wide Web. 1, 5

XML eXtensible Markup Language. 5–7, 49, 51, 54, 68, 75, 76, 78

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Software systems are nowadays pervasively being used to support businesses, allowing people to use services in multiple domains, such as banking, aerospace, online entertainment, healthcare, product manufacturing and autonomous vehicles, just to name a few [13]. Software is expected to operate in a robust manner when facing unexpected conditions, as a single failure may strongly affect the users or the software provider. This is especially true in mission-critical software, where a failure may have disastrous consequences on human lives [14]. Web services are particularly susceptible, as they are often exposed to issues arising from abnormal and malicious input which, when left unchecked, may lead to serious issues such as loss of data and disclosure of private information.

Most modern web systems are supported by Representational State Transfer (REST) services [15], a type of software system that follows the REST architectural style, which is essentially based on the principles that support the World Wide Web (WWW) [16]. In short, a REST service (also known as a RESTful service, a Web Application Programming Interface (API), or a REST API) relies on a Uniform Resource Identifier (URI) for the identification of each of its resources (e.g., a user profile, a purchase order), and on the Hypertext Transfer Protocol (HTTP) for exchanging messages (which are usually JavaScript Object Notation (JSON) documents). The use of HTTP includes the presence of an HTTP verb (e.g., GET, PUT, POST, DELETE) that specifies the type of operation that should be executed over the identified resource [16].

Major companies like Google, Instagram, Spotify, or Slack are now providing access to their services via REST APIs. In fact, the use of other types of interfaces to expose services is now residual, at least considering popular sites on the web [15]. REST is a relatively loose architectural style and some rigid aspects that are present in other similar styles or technologies (e.g., Simple Object Access Protocol (SOAP) services), like the mandatory presence of an interface description document (e.g., a Web Services Description Language (WSDL) document), lost their meaning in REST [16]. At the time of writing, there is no standard way of describing the interface of a REST service although the OpenAPI specification [17] is gaining popularity to the point of a quasi-standard [15].

The less rigid access to REST services opens space for unexpected inputs to be sent by client applications, potentially triggering residual faults that were not caught by Verification and Validation (V&V) activities performed by developers (e.g., static analysis, code inspections, 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, with the developer now having 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 certain argument can be found as a path parameter, i.e., it is part of the URI that identifies a resource [15]), or correctly documenting the API of a REST service. For instance, a mismatch between the service documentation and the actual 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 [18], and has been the target of several studies in the last decades. Koopman et al. [3, 19] 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 [4, 20, 21], embedded systems [22, 23, 24], middleware [25, 26, 27], self-adaptive systems [10], web applications [28], and SOAP services [29, 30, 31]. Despite this variety of explored domains, the robustness of REST services has been largely disregarded by researchers and practitioners.

In this dissertation, we present a systematic literature review on software robustness testing which highlights a gap in approaches for evaluating the robustness of REST services, and we aim at filling this gap by proposing an approach and a black-BOX tool for Robustness Testing of rest services (bBOXRT). We begin by thoroughly analyzing the state of the art on software robustness testing, in the form of a systematic literature review, and we also explore the related work on REST API testing. We use the gathered knowledge to emphasize the absence of academic and industry support for solutions to evaluate the robustness of REST APIs, a problem which we aim to solve by proposing a novel approach and tool.

Our approach for testing the robustness of REST APIs starts with a service description document of a REST service, and generates a set of invalid inputs (e.g., empty and boundary values, strings in special formats, malicious values) that are sent to the service in combination with valid parameters. We implemented the approach in the form of a tool, named bBOXRT, which can operate as a client application as well as a fault injection proxy between another client and the server (i.e., without requiring information regarding the service interface). Service responses are preliminarily analyzed 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 1351 operations) that fit in different types: public services, middleware management services (i.e., Docker), services built in-house (i.e., two Transaction Processing Performance Council (TPC) benchmarks), and private services. We performed a total of 399901 tests, in which we disclosed a total of 24373 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 (e.g., 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., in services carrying Structured Query Language (SQL) injection security vulnerabilities) as well as private information (e.g., stack traces disclosing the service code structure, database queries, database instance names).

The main contributions of this dissertation are the following:

- A comprehensive analysis, in the form of a systematic review, of the different approaches proposed in the literature in the past decades for performing robustness evaluation on software systems;
- 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 in [32] to be used by researchers and practitioners;
- The practical application of bBOXRT 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 outcomes of this dissertation comprise the following two paper submissions, both of which are, at the time of writing, under evaluation:

- A submission to the ACM journal on Computing Surveys (ACM CSUR) reporting on the systematic literature review on software robustness evaluation;
- A submission to the Journal of Systems and Software (JSS) reporting on the design of our approach for REST service evaluation, its implementation in the form of bBOXRT, and the practical evaluation of bBOXRT.

The rest of this work is organized as follows. Chapter 2 provides some background on concepts regarding the REST architecture and software testing. Chapter 3 presents a comprehensive analysis, in the form of a systematic literature review, of the state of the art on software robustness testing, and Chapter 4 provides an overview of the related work, from both the academic and industry perspectives, on approaches for testing REST services. Chapter 5 introduces our approach for testing the robustness of REST services, and describes the architecture of our tool, bBOXRT, and its configuration properties. Chapter 6 describes an experimental campaign carried out to evaluate our approach and tool, presents the results of the evaluation, and highlights our main findings. Chapter 7 discusses the threats to the validity of this work. Finally, Chapter 8 concludes this work. We additionally include Appendix A, where we present an overview of the work plan for this dissertation and briefly discuss the positive and negative aspects of its execution.

This page is intentionally left blank.

Chapter 2

Background

In this chapter, we provide background on fundamental concepts regarding the main topics of this dissertation, namely, Representational State Transfer (REST) services and software testing. In Section 2.1 we introduce the REST architectural style, describe its supporting ideas, and provide a small (but representative) example architecture of a REST service and its composing elements. In Section 2.2, we detail the main characteristics that comprise a software testing activity, and we briefly explain where robustness testing, a particular focus of this work, fits in the context of software testing.

2.1 REST architecture

Introduced by Fielding in the year 2000 [16], REpresentational State Transfer (REST) is an architectural style for distributed hypermedia systems (i.e., hypermedia is a generalization of hypertext for content other than Hypertext Markup Language (HTML), such as JavaScript Object Notation (JSON) or eXtensible Markup Language (XML) documents, for example). It employs the client-server paradigm and is essentially based on the principles that support the World Wide Web (WWW) [16], by using the Hypertext Transfer Protocol (HTTP) for exchanging messages. The most common use of the REST architecture is on web services (i.e., REST or RESTful services), and it is based on the following set of principles [16, 33]:

- *Client-Server*: The server holds resources with which the client interacts by sending requests to access and manipulate data. In a REST system, storage and interface concerns must be properly separated.
- *Stateless*: No client state should be stored in the server between requests, therefore each client request must contain all necessary information for the server to properly interpret it.
- *Cacheable responses*: Responses must be defined as cacheable or non-cacheable (for scalability purposes).
- *Uniform interface*
  - *Resource addressability*: Every resource in a REST service should be referenced by a unique Uniform Resource Identifier (URI).
  - *Manipulation of resources through representations*: If a client holds a representation of a resource and any related metadata, it should hold enough information to manipulate the resource.

  - *Self-descriptive messages*: Requests and responses must be self-descriptive, i.e., they should hold enough information to describe how to process them.
  - *Hypermedia As The Engine Of Application State*: Also known as HATEOAS, this principle states that server responses must provide links to related and available resources, and requesting a new resource places the client in a new state.

- *Layered system*: Server layers should be opaque to clients, and they cannot know whether they are communicating with the end server or an intermediate one.

- *Code on demand*: If required, the server may send executable code (e.g., in the form of JavaScript) to run on client-side.

REST systems expose their services through an interface, commonly referred to as a REST or Web Application Programming Interface (API). A REST API relies on the use of URIs to uniquely identify resources (i.e., metadata, files, or generally any other type of data), and each resource must provide one or more actions to perform over it (e.g., modify, remove). To achieve this, the REST architecture supports the use of the HTTP methods GET, POST, PUT, and DELETE, where each holds its own semantic value [33, 34]. These conveniently map to the CRUD operations (Create, Read, Update and Delete) which are well known in the database context. For instance, the GET method accesses an existing resource, and the POST method creates a new resource. The PUT method updates an existing resource and DELETE removes it. Additional HTTP methods such as HEAD, PATCH, CONNECT or OPTIONS are also considered in some REST services, but usually more attention is given to GET, POST, PUT and DELETE [15].

In a REST service, each pair composed of a unique resource URI and an HTTP method is referred to as an operation. An operation may optionally require parameters and a payload to be provided (e.g., POST operations commonly expect a payload, while GET may only contain parameters). Operation parameters can be defined in the HTTP headers or in URI itself. Header parameters include HTTP cookies, and values defined in custom-named headers. URI parameters include path parameters (i.e., variable parts of the URI, commonly documented as `<variable>`, such as in the URI `api/resource/<variable>` which could be instantiated as `api/resource/child1`) and query string parameters (i.e., defined after the `?` special character at the end of the URI and separated with a `&`, such as `api/resource?param1=value1&param2=value2`). The operation payload, on the other hand, is sent in the body section of HTTP requests, and even though most REST services use JSON objects for representing payloads [15], other formats such as XML, plain text, or even file formats (e.g., video files) are also widely accepted (mainly depending on the business logic of the service in question).

The interface (i.e., API) of a REST service must be documented for developers to be able to build fitting client applications. This documentation is usually written in API specifications, and there are many known specification formats nowadays, including Open Data Protocol (OData) [35], RESTful Service Description Language (RSDL) [36], and even RESTful API Modeling Language (RAML) [37], just to name a few, but the most widely accepted format is the OpenAPI specification (formerly known as Swagger) [17]. In an API specification, apart from describing the available resource URIs, their HTTP methods, and the parameters and payload used (if any), each operation can also detail the expected responses by HTTP status code. For instance, for the 200 OK (i.e., success) response, one may specify the HTTP response payload returned by the service, and for an error response (e.g., 400 Bad Request) a different payload may be returned (e.g., containing a message describing the error that occurred). This allows developers to verify if both the service

implementation and its specification are in agreement regarding the different responses returned.

Figure 2.1 depicts the general architecture of a REST service and a simplified representation of how it communicates with a client.

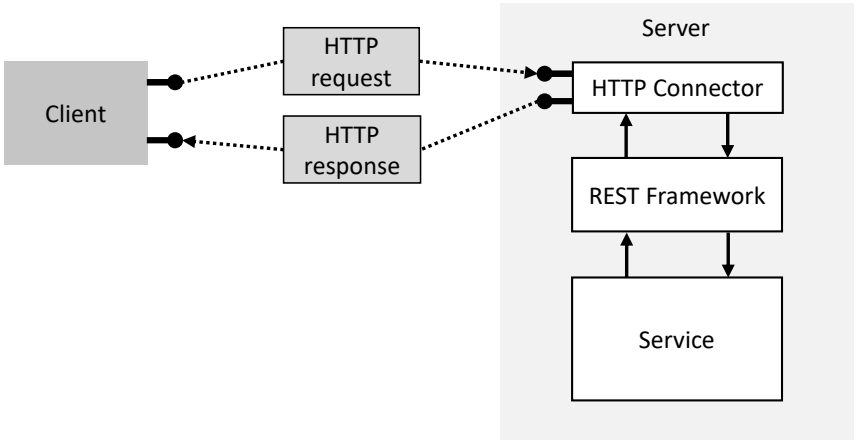


Figure 2.1: General architecture of a REST web service.

The client application is the one to initiate a communication, which is done by performing an HTTP request, as we can see in Figure 2.1. As stated earlier, the HTTP request must at least contain a URI for a valid resource in the server, as well an HTTP method which the server implements for that resource (e.g., GET, for access). Parameters and payload are optional and depend on what the targeted API operation requires. The HTTP request is sent through the Internet as a sequence of bytes, and eventually arrives at the server. The outermost component in the server, the *HTTP Connector*, is responsible for receiving this sequence of bytes and converting it into a more suitable format (e.g., a programming-level object). Malformed HTTP requests (i.e., those that do not conform to the HTTP specification [38]) are immediately rejected, and the common course of action by most REST frameworks is to send back to the client an HTTP response with a generic error message.

Otherwise, if the received HTTP request is well-formed, it is then passed on to the *REST Framework* component. Here, the contents of the request are interpreted in such a way that the server may identify the API operation the client targets (i.e., through the resource URI and the HTTP method used). If the specified URI does not match that of any resource on the server or if, for instance, there are security requirements which the client does not verify, this component halts the processing of the request and replies back to the client with a fitting error message. Likewise, if the specified URI is valid, but the HTTP verb does not match any of those the server implements for that resource (e.g., attempting a POST on a resource for which the server only implements GET), the server will send a response to the client with a suitable error message. In case the targeted API operation requires parameters or a payload, this component will also verify their presence (e.g., in case they are mandatory, as some inputs may be optional), extract them, and in the case of media type-formatted payloads (e.g., a JSON object, or an XML document), the contents of the HTTP request body (where the payload is located) will be tentatively converted into the correct format. A malformed payload will again trigger the server to send an error response back to the client.

Upon identifying the required API operation, the respective function of the *Service* com-

ponent (i.e., that which implements the API operation's logic) is called, and the inputs extracted from the HTTP request (if any) are passed to it. Here the service may perform calls to external systems which are outside of our scope, such as databases. The result from the execution of the called function is returned back to the calling component, the *REST Framework*, which may wrap it in a JSON object, for example. The necessary response elements are sent back to the *HTTP Connector*, which encapsulates everything in a fitting HTTP response object, serializes it into a sequence of bytes, and sends it back to the calling client application through the Internet, thus concluding one successful request-response communication cycle.

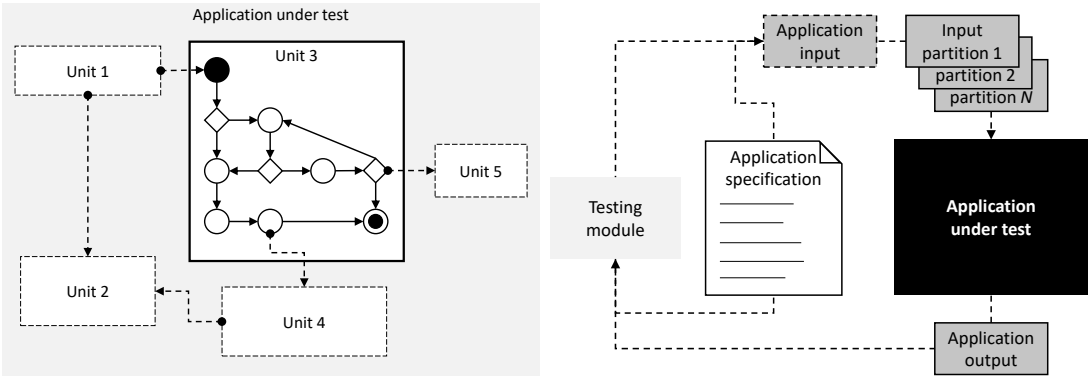
Multi-layered systems such as this, where various components with different responsibilities interact with each other in a coordinated manner, are prone to their own set of issues (i.e., software defects), which may ultimately impact the overall Quality of Service (QoS). It is thus necessary to apply fitting testing activities to REST services, as a way of disclosing and correcting potential defects before they ever cause any real problems. In the following section, we provide background on software testing by briefly explaining some of its essential concepts.

2.2 Software testing concepts

Software testing is the process of verifying that computer code behaves as designed and produces predictable results when executed [2, 39]. Contrary to passive verification methods such as design review, code analysis or formal methods such as inspections, software testing is an active process because it requires computer code to be executed. The goal of testing software is to find faults within its code (i.e., the root cause of a problem, also known as a bug or a defect) whichthrough the , when executed, produce errors the (i.e., anusederror is the difference between a computed value and the theoretically correct value [18]) which would ultimately result in a failure (i.e., the inability of a system or component to perform a required function according to its specifications [18]). It is important to note that software testing does not confirm the absence of code defects but, on the contrary, it is intended to confirm their presence [2]. Software testing methods can be categorized according to visibility (e.g., white-box) and granularity (e.g., system testing), which we will discuss in more detail throughout the next paragraphs.

In software testing, **visibility** refers to the degree to which a testing activity accounts for the logic and internal structure of the system or component under test. Full visibility of the entity under test is known as *white-box* testing, and refers to situations where testers (i.e., the users who define test cases) hold knowledge regarding its internal structure and reflect that on the test cases they create [1, 2, 39] (e.g., by defining test cases which exercise specific code paths of the functions under test). In Figure 2.2a, we show an example of the control flow white-box testing technique, which essentially constructs the control flow graph of the unit under test (i.e., a unit of code, such as a function) and evaluates all possible paths in order to detect an infeasible one (i.e., a path which, based on the data and decision structures involved, should not be reachable).

On the opposite side, lack of visibility regarding the system or component under test is known as *black-box* testing, wherein the internal structure is unknown to the testers. In this case, test cases are essentially based on available specification and interface definitions, as well as any additional knowledge of the system on behalf of the tester [1, 39]. We show in Figure 2.2b an example of the equivalence class partitioning black-box testing technique. This technique works by dividing (i.e., partitioning) the value ranges of inputs to the system



(a) Control flow testing. (b) Equivalence class partitioning.

Figure 2.2: Example of white-box (a) and black-box (b) testing, inspired by [1].

under test into logical classes (e.g., distinguishing negative from positive numbers results in two classes). Rather than selecting purely random inputs (i.e., which is not ideal for large value ranges), testers may select at least one value (e.g., also randomly) from each of the partitioned input classes. The correctness of the output returned by the system under test is then verified against the specification. Note that an additional intermediate level, called grey-box testing [40], also exists but is far less commonly used by software testers.

Regarding the **granularity** factor, it refers to the scale of a given testing activity, and it is intrinsically related to the development life-cycle of a software system. Figure 2.3 shows the *V-Model*, adapted from [1, 2], a model which describes the main phases of a software development process and how each phase relates with a specific testing activity. In essence, the *V-Model* shows that while software development begins with coarse-grained activities (e.g., definition of the requirements and functional goals) and eventually reaches fine processes with a more focused scope, where smaller-scale elements are produced (e.g., code implementation), testing activities start by focusing precisely on those elements (i.e., units of code, using unit testing) and gradually increase in scope by integrating those elements with each other (e.g., integration testing, system testing). Note that, in Figure 2.3, we included *acceptance testing* for consistency only, and it is not relevant to this work as it is a user-centered testing activity, rather than focusing on implementation correctness or specification compliance [1, 2].

During the software development process, the first testing activities begin in parallel with the implementation phase (Figure 2.3). At this point, *unit testing* is introduced into the development process, a fine-grained software testing activity for exercising logic and data structures in individual units of code (e.g., functions, classes) [1, 2]. Unit testing is generally carried out using white-box testing techniques (e.g., control flow testing, data flow testing), as testers are interested in verifying that individual blocks of code are implemented properly. In this testing activity, units of code are tested in isolation (i.e., there is no communication with other units), which reduces the overall effort required to diagnose problems and identify root causes, because fewer dependencies are involved.

Module testing is a larger-scale form of unit testing where individual modules are evaluated in isolation (i.e., a module is a collection of related units assembled in a file, package, or class). The goal is to assess whether the composing units of code interact with each other in an expected manner [1]. The next level of testing during software development is called *integration testing*, and it is here that the previously tested modules are integrated with each other (i.e., their interfaces are connected, and the system under development begins to

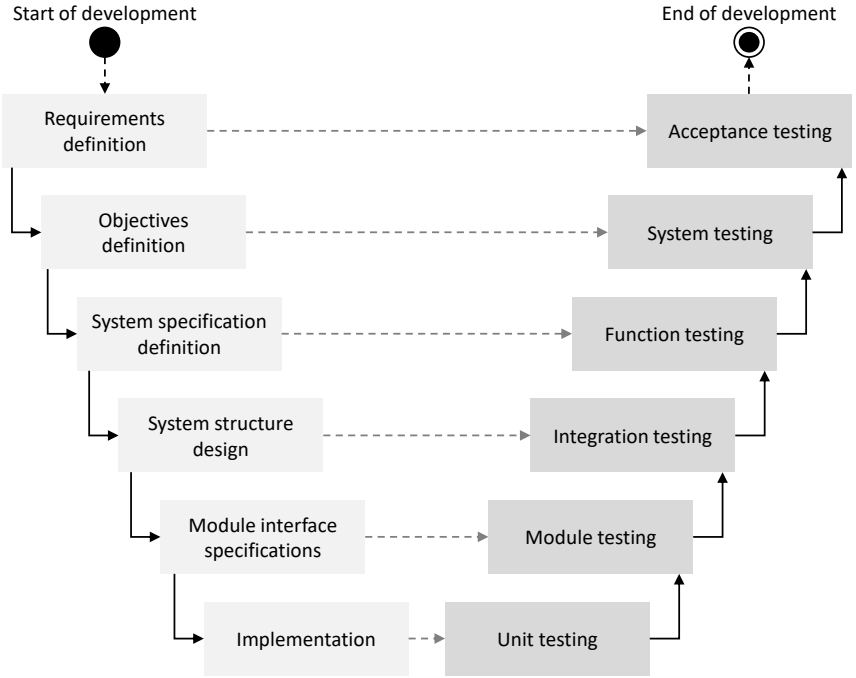


Figure 2.3: The *V-Model* for software development and testing, adapted from [1, 2].

take shape). The main assumption of integration testing is that the modules that compose the system work correctly [1], and the focus is primarily on assessing the correctness of inter-module communication protocols [2].

The first black-box testing activity is *function testing*, where the (now fully integrated) system is exercised using only knowledge regarding the interface and specification [1, 2]. Here, only the perspective of the user is considered (i.e., rather than the privileged perspective of the developer), and thus testing techniques such as equivalence class partitioning or boundary value analysis are used to guarantee a good coverage of the existing input value ranges defined in the system interface. The output of the system under test is then checked against its specification to identify discrepancies (i.e., inconsistencies between the implementation and the specification [39]).

We then reach *system testing*, whose main goal is to verify whether a software system complies with its original objectives (e.g., quality attributes, such as availability or reliability). For this, there must be a quantifiable and measurable way of assessing the level with which a system complies with its intended objectives [1, 2]. There are many types of system testing techniques, and the choice depends mainly on the quality attribute (i.e., system objective) that must be assessed. These include usability testing (i.e., user-centered testing technique which assesses how intuitive and user-friendly the interface of a system is), performance testing (i.e., used to verify that a system is able to complete certain tasks within well-defined time or volume constraints), stress testing (i.e., assesses whether a system is able to handle stress conditions caused by heavy workloads), security testing (i.e., evaluates the capability of a system to stop unauthorized users from accessing protected system resources), and reliability testing (i.e., assesses the stability of a system by measuring the mean time between consecutive failures within a given time-period), just to name a few [2].

Closely related to reliability testing, and also a form of system testing, *robustness testing* is the process of triggering design or programming faults within a given system in an

attempt to induce incorrect operation (i.e., a robustness failure) [41]. Robustness testing relies mostly on feeding limit condition (e.g., out-of-bounds values) or erroneous inputs to the interface of a system. The measure of robustness of a system is given as the ratio between the amount of test cases that expose robustness faults and the total number of executed test cases [41]. A system is thus believed to be robust if it maintains nominal operating behavior when exposed to external faults [42]. In this work, we are interested in evaluating the robustness quality attribute (with particular focus on REST web services), and in the following chapter we explore the existing state of the art on software robustness testing.

This page is intentionally left blank.

Chapter 3

Related work on software robustness evaluation

In this chapter, we present a systematic literature review on software robustness evaluation. We start by describing, In Section 3.1, the methodology we propose for conducting this review, where we analyze related surveys, present the research questions considered for our review, explain how we identified and selected relevant studies, and how we analyzed and synthesized the collected papers. In Section 3.2, we group the selected papers by type of software system, and analyze all the research in each of the groups, placing particular emphasis in the elements that allow us to answer the research questions. We then present a discussion, in Section 3.3, where we answer each of the proposed research questions. Finally, Section 3.4 concludes this chapter by highlighting clear gaps in the current state of the art and identifying challenges for future research.

3.1 Systematic review plan

In this section, we present the methodology used to perform a systematic literature review on software robustness evaluation, which is based on well-established guidelines for conducting systematic reviews [43, 44], and is comprised of the following steps:

- **Analysis of related surveys**: We begin by identifying and analysing secondary studies (i.e, surveys, other systematic reviews or mapping studies) that, at least partially, cover the topic of software robustness evaluation. The related surveys are analyzed so that gaps and limitations are identified and covered in our own discussion (in Section 3.3).
- **Definition of research questions**: Based on the gaps identified in the previous step, we define a set of research questions that form the objectives of our systematic review.
- **Identification of studies**: We identify the data sources (e.g., search engines, online libraries) that are used to identify relevant primary studies, i.e., the individual studies that contribute to the review. We then identify the query used to perform the search and the snowballing process [45] used to complement the identification of studies.
- **Study selection**: This step refers to the application of inclusion and exclusion criteria on the primary studies found, as well as the quality assessment criteria used.

The goal is to select the ones that agree with the goal and scope of our review.

- **Data extraction and synthesis**: In this step we extract the relevant data (i.e., according to the research questions) from the selected primary studies. The information is then synthesized to allow answering the research questions.

Each of these steps is described in further detail in the following subsections.

3.1.1 Analysis of related surveys

In the following paragraphs, we present related secondary studies (e.g., surveys, other reviews) identified using the previously mentioned data sources. We identified the following secondary studies:

- Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. Robustness testing techniques and tools. In Resilience Assessment and Evaluation of Computing Systems, pages 323–339. Springer, 2012.
- Ali Shahrokni and Robert Feldt. A systematic review of software robustness. Information and Software Technology, 55(1):1 – 17, 2013.
- Syed Muhammad Ali Shah, Daniel Sundmark, Birgitta Lindström, and Sten F. Andler. Robustness testing of embedded software systems: An industrial interview study.IEEE Access, 4:1859–1871, 2016.

Micskei et al. [41] survey the state of the art on robustness testing techniques and tools, and show the evolution of basic testing techniques. The work identified a series of testing techniques, and mapped them to a timeline ranging from the early 1990's to the early 2000's. The identified techniques, in chronological order, are physical fault injection, random inputs, invalid inputs, type-specific tests, mutation techniques, and object-oriented tests. Additionally, a set of commonly targeted system types were also identified, which comprise state-based systems with a graphical user interface, high availability middleware, real-time microkernels, online transaction processing systems, database management systems, and web services.

The authors note that, throughout the years, researchers have shifted from general techniques which do not account for the specificities of the target system (i.e., physical fault injection), to more modern software-implemented testing techniques (e.g., object-oriented tests) which target a unique type of system (e.g., web services) with specific characteristics. Also, the state of the art analyzed by the authors emphasizes the need for additional studies on the identification of the most useful robustness models [41]. The analysis focuses on testing and does not consider techniques that are not based on testing. Also, it is limited to the identification of testing techniques and systems, without further insights being synthesized.

Shahrokni and Feldt [46] present a systematic review on the topic of software robustness, in which the authors include primary studies whose main goal may not be robustness evaluation (e.g., robustness improvement, dependability assessment). The authors identified a set of 144 works on software robustness, and categorized each according to the phase of software development targeted by a study (e.g., requirements, evaluation), the type of systems targeted, the type of research performed (e.g., philosophical paper, evaluation paper), contribution of the study to the research community (e.g., tool, model), and the

type of evaluation performed by the authors of a study (e.g., large academic evaluation, no evaluation).

The authors observed that there is a lack of research contributions on software robustness that focus on the requirements engineering phase of software development, and that almost all studies focus on robustness issues caused by invalid inputs and ignore other aspects such as stressful conditions. The areas with greater research focus are verification and validation, design, and architecture solutions for robustness improvement (e.g., building a wrapper around the software to filter out faulty inputs). The main gap that the authors highlight, is the fact that only a few studies cover elicitation and specification of robustness requirements. Also, the authors concluded that there is a need for expanding robustness research to real-world industry systems, and that many academic solutions are not evaluated enough to be useful in industry contexts. Although the survey is quite inclusive [46], there is little insight regarding the approaches used, as the authors tend to focus on processual aspects (e.g., type of research, type of contribution).

Shah et al. [47] present a review of the state of the art and practice in robustness testing of embedded systems, across seven different domains. The authors show that the state of the practice regarding robustness testing is quite different from the state of the art described in the scientific literature. For instance, testing methods the authors found often in the state of the art (e.g., fuzzing), are not used in the industry organizations the authors studied. Instead, ad-hoc approaches are used, which take specific scenarios into account (e.g., power failure or overload). Additionally, the authors also found that it is uncommon for practitioners to categorize behavior of the systems under test.

The analysis of the identified secondary studies lead us to identify significant gaps, which we believe are sufficient to motivate conducting a systematic literature review on software robustness evaluation. In summary, the main gaps identified during the analysis of the related surveys are as follows:

- The review carried out by Micskei et al. [41] is, in what concerns robustness evaluation, mostly limited to the identification of testing techniques and targeted systems. There is a need to discuss further synthesized knowledge regarding the technical aspects of robustness testing approaches analyzed and also discussing robustness evaluation approaches (e.g, model-based approaches that may not rely on testing).

- The systematic review by Shahrokni and Feldt [46] is the one that is closest to ours (in terms of its goals) and has a strong focus on processual aspects (e.g., type of research, type of contribution, phase of development targeted, evaluation size), rather than on the technical aspects (e.g., types of faults used) involving each study.

- Shah et al. [47] focus in robustness testing of embedded systems, across different domains. Besides the limited scope, other assessment techniques are not accounted for, which emphasizes the need of a broader review.

The following subsection describes the research questions that serve to guide this systematic review.

3.1.2 Systematic review research questions

Based on the analysis of existing secondary studies, we defined the following set of five research questions that form the objectives of this literature review:

- **RQ-1**: Which types of software systems are the subject of robustness evaluation?
- **RQ-2**: Which techniques are used to evaluate software robustness?
- **RQ-3**: Which are the targets used by software robustness evaluation approaches?
- **RQ-4**: Which types of faults are being used in software robustness evaluation?
- **RQ-5**: Which are the methods used to characterize robustness?

With the aforementioned research questions of this review, we mainly intend to analyze the different types of approaches considered by researchers for evaluating the robustness of software systems, by breaking down each approach into five parts. Specifically, in **RQ-1** we aim at identifying and systematizing the general types of systems that have been studied by researchers (e.g., operating systems, web services); in **RQ-2** we aim at identifying the main different testing techniques considered (e.g., fault injection, static code analysis); **RQ-3** aims at a particular, but central, aspect of the techniques used, which is the entity being targeted (e.g., a message, an API call, a model); **RQ-4** aims at characterizing the types of faults used is robustness evaluation (e.g., boundary inputs, timing faults); and with **RQ-5** we aim at understanding which are the methods used to classify behavior (e.g., a failure mode severity scale, binary classification).

3.1.3 Identification of studies

We used six well-known online libraries to search for primary studies. These are common presence in related studies (e.g., [48, 49, 50]) and were selected to allow reducing any possible search bias [43]. The data sources are the following:

- *Google Scholar* [51];
- *DBLP* [52];
- *IEEE Xplore* [53];
- *ACM Digital Library* [54];
- *Scopus* [55];
- *Springer Link* [56].

Google Scholar was our first selection of data source as it is known to index a quite large number of works. We then complemented this first search by using the remaining identified sources in order. To perform the search, we used the following query string, built based on preliminary observations using the respective search engines of the online libraries:

("software robustness" OR "system robustness") AND ((test OR testing) OR (benchmark OR benchmarking) OR (assessment OR assessing) OR (evaluation OR evaluating))

We found a need to group robustness with software or with system (i.e., "software robustness", "system robustness") which instructs the search engines to perform a literal match. This was due to the fact that the word robustness is very frequently used in other domains (e.g., medical, statistics) and by not grouping these pairs the search would mostly include

results unrelated with our context of interest. Also, our preliminary observations and previous knowledge indicates that words like *assessment*, *evaluation*, *testing*, or *benchmarking* are strongly used in software robustness evaluation research. So, we included variations of these words in the remainder of the query string, making it is less likely to miss relevant papers.

It is also worthwhile mentioning that, as a complement to the online search and with the goal of enriching the set of selected studies, we also used snowballing [45]. Thus, we analyzed the references of each identified work (to identify further related studies, not captured by the query string used in the different sources). We also analyzed the research that cites the identified studies, using the citation information available in Google Scholar.

3.1.4 Study selection and quality assessment

We filtered the identified studies mostly by applying a set of inclusion and exclusion criteria, which determine whether a particular primary study should be kept or discarded. Being aware of the error prone and complex process of quality assessment that may result in erroneously excluding papers from the review [57], we applied a conservative and objective criteria of including only peer-reviewed publications in our survey. We added this to the inclusion/exclusion criteria, which, in this sense, already take the quality assessment aspect into consideration. The inclusion and exclusion criteria we considered for this systematic review are the following:

- Inclusion Criteria:
  - A study whose main focus is set on the evaluation (in *lato sensu*) of robustness of software systems.
  - The study should be sufficiently complete to allow answering the identified set of research questions.
- Exclusion Criteria:
  - A study whose main focus is set on extraneous robustness aspects (e.g., the definition of techniques for improving robustness), although may have a robustness evaluation part, should be excluded.
  - Non-peer and uncited reviewed research should be excluded from the review.

To apply these criteria, we went through three pruning phases. In a first phase analyzed the *title of a publication*, a quite fast method for discarding unsuitable results. Because at this level we have little information for selecting studies (i.e., only the title), we found it to be over-inclusive. The second pruning phase was based on reading the *abstract* of each identified study and deciding if it should be included. We were conservative and included the work, even in the cases where the abstract was vague (i.e., because exclusions should be done with as much information as possible). In a third pruning phase, we *read the full text*, with focus on the description of the approach to decide about its inclusion or exclusion.

3.1.5 Data extraction and synthesis

This final step consists of reading each identified primary study and gathering information regarding each approach. Reading was performed with the five research questions in mind, thus we gathered information regarding the type of system being evaluated, the

technique(s) used, the target of each technique, the types of faults used, and the methods to characterize robustness. As the authors use very diverse terms, this was carried out iteratively, starting first with the collection of the specific terms used by the authors, which were then individually discussed and, whenever applicable, iteratively generalized to more widely accepted consensual terms. This process involved discussion and agreement by an Early Stage Researcher and an Experienced Researcher.

3.1.6 Outcome of the study identification and selection

We performed the online search on the 20th of July 2020, from which we obtained over 16300 results from our first data source, Google Scholar. Searching through the remaining sources allowed us to obtain 125 results from DBLP, 642 from IEEE Xplore, 2136 from Springer Link, 1612 from Scopus, and 793 from ACM Digital Library. As we later observed that the results reported by Google Scholar included all works retrieved from the other data sources, to simplify the presentation, we now report only on the list of works identified by Google Scholar. Table 3.1 presents the outcome of the study identification and selection.

Table 3.1: Identification and selection process using Google Scholar

Source	Total results	Title pruning	Abstract pruning	Content pruning	Snowballing	Final
Google Scholar	16300	173	144	119	13	132

As we can see in Table 3.1, although we started with a quite large set of works, title pruning allowed to quickly reduce the set. We must also note that snowballing contributed to about 10% of the final set of papers selected. Table 3.2 sets the outcome of this process in perspective with what is analyzed in related surveys. At the left-hand side, the table identifies the related surveys, the respective periods analyzed in each survey, the total number of papers analyzed and how many of those relate to robustness evaluation. At the right-hand side the table shows the number of papers that were not considered for analysis by the authors of each survey (and are included in our own survey) within each period of analysis, outside each period of analysis, and a total count.

Table 3.2: Number of studies analyzed in the related surveys

	Period	Total papers	Robustness evaluation	# of papers not considered		
				Inside period	Outside period	Total
Micskei et al. [41]	1988-2010	32	20	65	47	112
Shahrokni and Feldt [46]	1990-2011	144	42	48	42	90
Shah et al. [47]	1990-2014	22	20	94	18	112
This work	1990-2020	132	132	–	–	–

As we can see in Table 3.1, the outcome of the identification and selection of studies has resulted in the fact that our systematic review analyzes a comparatively larger number of papers, even considering the closest survey to our context by Shahrokni and Feldt [46]. The differences in the number of works analyzed are still clear even if we limit the comparison to the periods analyzed by other surveys. This is a natural consequence of the different process followed in our systematic review, which lead to a more extensive identification of related work on robustness evaluation.

3.2 Analysis of studies on software robustness evaluation

This section presents the studies selected during our review of the state of the art on robustness evaluation. The different works are organized throughout the next subsections according to the distinct types of systems targeted, which we found to concentrate around the following seven groups:

- **Operating systems**: General purpose operating systems were the first systems targeted by robustness evaluation research [3], and are widely regarded as a prominent case. As such, this type was selected to initiate the analysis.
- **Communication systems**: A broad category encompassing different types of network systems (e.g., sensor networks) and including communication protocols implementations (e.g., Session Initiation Protocol (SIP)) [58].
- **Embedded systems**: Systems generally designed to handle a certain single specific task (i.e., in contrast with general-purpose systems) and are frequently part of a larger system. They are often used in mission-critical environments, where quality attributes like timeliness or safety are usually important [59, 60].
- **Middleware**: Software that supports the operation of other software and lies between different layers of a more complex system [61]. In the case of this work, we consider a broad definition that includes classic middleware (e.g., Message-Oriented Middleware implementations, middleware management services like OpenStack or Docker).
- **Software components**: In this group, we mostly consider Commercial Off-The-Shelf (COTS) software components and applications, or other types of reusable software components (e.g., libraries) [62, 63].
- **Web services**: Client-server software that mostly relies on the Hypertext Transfer Protocol (HTTP) protocol for message transport, this includes services that mostly interact with human users (e.g., via a browser) or services for heterogeneous application integration, such as Simple Object Access Protocol (SOAP) web services [64, 65].
- **Autonomous and adaptive systems**: Systems that are able to adjust to changes in the the surrounding dynamic environment, usually using a feedback loop mechanism for performing the necessary adaptations that allow the system to reach the desired goals [66].

The next subsections go through each of the aforementioned types of systems. For each type of system, we begin by summarizing its main characteristics, and then we go through main aspects like the technique used by each work (e.g., fault injection), the target of the technique (e.g., messages, system calls) and the types of faults considered (e.g., round-off errors, timing faults). We also graphically represent one illustrative case per type of system, where we try to show the main concepts involving the selected robustness testing approach.

3.2.1 Operating systems

Operating systems are the software layers upon which applications run in most computer devices, and these applications use the operating system to interface with the hardware [67].

A failure in an operating system (e.g., caused by an application erroneously calling system functions) may put at risk all other applications running on top of it, as all of them rely on its correct operation [19]. Thus, robustness becomes an essential property for guaranteeing the correct functioning of operating systems [3]. Robustness evaluation techniques have first been applied in the operating systems domain in the nineties [3, 68, 69], but this research topic has remained relatively active until recently [70, 71].

In this subsection, we review existing approaches for robustness evaluation of general-purpose operating systems, including mobile operating systems. More specific cases such as real-time operating systems are discussed in Section 3.2.3. Table 3.3 holds an introductory short summary of the main types of systems, techniques, main technique targets (i.e., what a given technique focuses on), types of faults, and behavior classification schemes used to evaluate the robustness of operating systems. We also include the number of works found for each of these items. Note that some works use more than one technique or type of fault (for example), which means that the sum of the numbers in a given column may not add up to the total number of works described in this section.

The next paragraphs present the related work targeting this type of systems, by the end of this subsection we briefly analyze the evolution of the robustness assessment techniques for operating systems, and a more detailed discussion is left for Section 3.3.

Miller et al. presented, in 1990, the fuzz robustness testing technique, and applied it to a large set of command line utilities across 7 different UNIX systems [68]. The fuzz technique works by randomly generating strings composed of printable, control and NULL characters, and feeding them as input arguments to the command utility under test. Over the years, the authors explored different environments and applications to test with the fuzz technique, having also covered graphical applications. These are tested by generating random mouse and keyboard events (e.g., key up/down, multiple mouse clicks, mouse drags) in order to simulate the actions of human users. In the following years, the authors applied this testing method to X-Window Graphical User Interface (GUI) applications [81], and to Win32 GUI applications [89]. A later study published by the same authors [82] focused on several command line utilities and GUI applications of the UNIX-based Mac OS X operating system. Having covered a time span of more than a decade testing different operating systems and several of their native applications, the authors concluded that operating system developers have continued to make similar robustness mistakes throughout the years, and that the same kinds of bugs were observed in the four studies.

Suh et al. present a robustness benchmark suite, composed of four primitive robustness benchmarks targeting specific operating system functionalities, such as the file management system, memory access, user application, and C library functions [86]. The authors tested the file management system using random inputs, the memory access functionality was tested by reading from and writing to random addresses, and by attempting to write past the boundaries of a static string, the user application was tested with stuck-at memory faults (i.e., forcing one or more bits to the value 1), and the C library functions were tested using random input values. In a very similar paper, the following year, Siewiorek et al [85] tested these four components from multiple UNIX-like systems (e.g., SunOS, Ultrix, Mach, AIX, HPUX), and the authors encountered several errors resulting from robustness problems, which lead the authors to label the tested systems as having a low level of robustness.

Koopman et al. describes a portable robustness assessment method for evaluating the dependability of COTS operating systems [3], which has been developed in the context of project Ballista. The approach is carried out in a distributed manner, with the target computer running the operating system being tested, a Test process (responsible for carrying

Table 3.3: Techniques for evaluating the robustness of operating systems

Systems	Android	Feng and Shin [72], Maji et al. [73], Sasnauskas and Regehr [74], Ye et al. [75]
	UNIX-like	Acharya et al. [76], Albinet et al. [77], Cong et al. [70], Kanoun et al. [78], Koopman and DeVale [19, 79], Koopman et al. [3], Kropp et al. [80], Miller et al. [68, 81, 82], Montrucchio et al. [83, 84], Shelton et al. [69], Siewiorek et al. [85], Suh et al. [86], Velasco et al. [71], Xiang et al. [87]
	Win32	Durães and Madeira [88], Forrester and Miller [89], Johansson et al. [90], Kanoun et al. [78], Mendonça and Neves [91], Shelton et al. [69]
	WinNT	Durães and Madeira [88], Ghosh et al. [92], Schmid et al. [93], Shelton et al. [69]
Techniques	Code changes injection	Durães and Madeira [88]
	Fault injection	Albinet et al. [77], Cong et al. [70], Durães and Madeira [88], Ghosh et al. [92], Johansson et al. [90], Kanoun et al. [78], Koopman and DeVale [19, 79], Koopman et al. [3], Kropp et al. [80], Mendonça and Neves [91], Montrucchio et al. [83, 84], Schmid et al. [93], Shelton et al. [69], Siewiorek et al. [85], Suh et al. [86], Velasco et al. [71], Xiang et al. [87]
	Fuzzing	Feng and Shin [72], Forrester and Miller [89], Maji et al. [73], Miller et al. [68, 81, 82], Sasnauskas and Regehr [74], Ye et al. [75]
	Interception	Albinet et al. [77], Cong et al. [70], Johansson et al. [90], Kanoun et al. [78]
	Model-based analysis	Acharya et al. [76]
	Static code analysis	Acharya et al. [76], Sasnauskas and Regehr [74]
Targets	Application address space	Siewiorek et al. [85], Suh et al. [86]
	Command arguments	Miller et al. [68, 81, 82]
	Device driver calls	Albinet et al. [77], Johansson et al. [90], Mendonça and Neves [91], Xiang et al. [87]
	Function calls	Koopman and DeVale [19], Siewiorek et al. [85], Suh et al. [86]
	GUI elements	Miller et al. [81, 82], Forrester and Miller [89]
	IPC messages	Feng and Shin [72], Maji et al. [73], Sasnauskas and Regehr [74], Ye et al. [75]
	Kernel address space	Montrucchio et al. [83, 84], Velasco et al. [71]
	Machine code	Durães and Madeira [88], Sasnauskas and Regehr [74]
	Specification	Acharya et al. [76]
	System calls	Cong et al. [70], Ghosh et al. [92], Kanoun et al. [78], Koopman and DeVale [19, 79], Koopman et al. [3], Kropp et al. [80], Schmid et al. [93], Shelton et al. [69], Siewiorek et al. [85], Suh et al. [86], Xiang et al. [87]
Faults	Bit-level faults	Johansson et al. [90], Montrucchio et al. [83, 84], Siewiorek et al. [85], *Suh et al. [86], Velasco et al. [71]
	Boundary inputs	Albinet et al. [77], *Ghosh et al. [92], Kanoun et al. [78], Schmid et al. [93], Siewiorek et al. [85], Suh et al. [86], Xiang et al. [87]
	Invalid inputs	Albinet et al. [77], Cong et al. [70], Ghosh et al. [92], Kanoun et al. [78], Koopman and DeVale [19, 79], *Koopman et al. [3], Kropp et al. [80], Mendonça and Neves [91], Schmid et al. [93], Shelton et al. [69], Xiang et al. [87]
	Invalid return values	*Schmid et al. [93]
	Programming errors	*Durães and Madeira [88]
	Random inputs	Albinet et al. [77], Feng and Shin [72], Forrester and Miller [89], Ghosh et al. [92], Maji et al. [73], *Miller et al. [68, 81, 82], Sasnauskas and Regehr [74], Siewiorek et al. [85], Suh et al. [86], Ye et al. [75]
Classification	5 categories	Kanoun et al. [78], Koopman and DeVale [19, 79], Koopman et al. [3], Kropp et al. [80]
	4 categories	Johansson et al. [90], Mendonça and Neves [91]
	3 categories	Xiang et al. [87]
	Binary	Acharya et al. [76], Albinet et al. [77], Cong et al. [70], Durães and Madeira [88], Feng and Shin [72], Forrester and Miller [89], Ghosh et al. [92], Maji et al. [73], Miller et al. [68, 81, 82], Montrucchio et al. [83, 84], Sasnauskas and Regehr [74], Schmid et al. [93], Shelton et al. [69], Siewiorek et al. [85], Suh et al. [86], Velasco et al. [71], Ye et al. [75]

out the tests), and a Starter process that initiates testing and carries out several startup tasks (e.g., opening sockets). Another computer serves as a watchdog, receiving health check messages from the Test process. The overall setup used is illustrated in Figure 3.1.

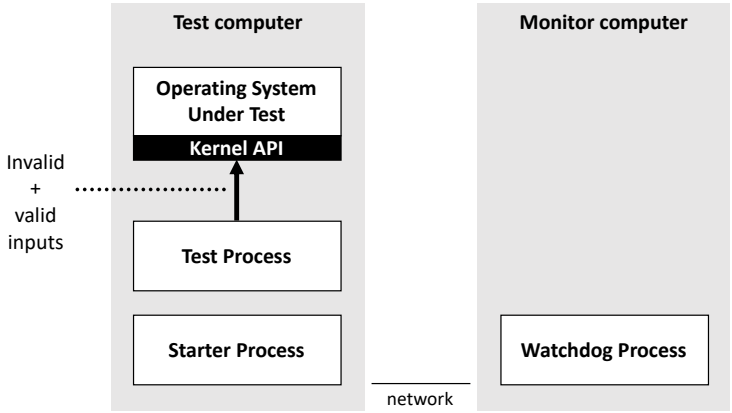


Figure 3.1: Ballista project approach for assessing operating system robustness, adapted from [3].

The approach works by first identifying a set of system calls in the operating system Application Programming Interface (API), and defining a set of valid and invalid inputs for each data type in the set of call parameters. Invalid inputs represent limit conditions or values holding special characteristics (e.g., *NULL*, maximum data type value, 0, 1, -1, values that are equal to or that exceed the valid domain of a certain argument), which tend to be the source of robustness problems.

The proposed approach was applied to a set of UNIX-like operating systems by targeting the kernel functions *read*, *write*, *open*, *close*, *stat* and *fstat*, which mostly take as parameters memory buffers, flag parameters (constant integer values), buffer lengths and file names, and present a predefined set of values to use for each of these data types. For instance, in the case of file handles (internally, integer values), the authors considered the value -1, *NULL* and handles to a i) valid but closed file, ii) valid but opened file or iii) to a deleted file. Clearly, the *NULL* value represents an invalid file handle, while the a handle to an open file represents a valid file handle value. The authors consider that these selected valid, invalid and limit parameter values are more prone to trigger faults.

Finally, the authors propose a scale for qualitative categorization of robustness faults based on the severity of the failure, named *CRASH*. The five categories of this scale are: **C**atastrophic - operating system crashes or multiple tasks affected; **R**estart - process hangs and requires restart; **A**bort - process aborts; **S**ilent - exception was not signaled but should have been; and **H**indering - incorrect exception signaled.

Kropp et al. introduce the Ballista tool in [80]. Ballista is a testing tool that supports automatic generation and execution of black-box robustness tests through the interfaces of COTS software components. This only requires that testers have information regarding function parameters and data types. The approach is composed of the following steps: i) establishing an initial system state, ii) executing a single call (with a test case) to the module under test, iii) assessing any disclosed robustness issue, and iv) restoring system state. Robustness problems found during testing are categorized according to the CRASH robustness scale [3]. The Ballista tool was tested on several Portable Operating System Interface (POSIX) operating systems, and the authors consider the approach to be effective because most of the setup effort revolves around mapping parameter data types to

applicable faults, and test cases are automatically created.

In [19], Koopman and DeVale present the results of a robustness comparison of a set of POSIX operating systems, based on a multi-version comparison technique (i.e., multi- or N-version programming is the concept of using different implementations of a given specification as a way of increasing dependability). If at least one of N systems in the setup signals an exception (e.g., by reacting to invalid input), a robustness fault is attributed to the versions which did not signal it, which the authors call the *one-of-N* version comparison strategy. This approach produces a number of false-positives, which the authors mitigated by automatically excluding all non-exception test cases (i.e., those in which none of the versions signalled an exception), and also by performing a manual verification of a sample of test cases. The experimental evaluation relies on the Ballista tool that is used to target a set of system calls and C library function calls.

The main outcomes of the aforementioned work [19] include the following: i) using multi-version allowed to detect *Silent* failures (i.e., failure to report an exceptional case) and also to filter out non-exceptional cases; ii) conformance issues to POSIX were detected in POSIX-certified systems; iii) The diversity of the systems used would not allow for perfect robustness even if the best systems were to be combined; iv) C library calls were less robust than operating system calls (and also not highly diverse in terms of robustness, despite their different implementations).

Following their work in operating systems robustness, Koopman et al. have also presented a more mature Ballista-based approach in [79], which was used to assess the robustness of fifteen different POSIX implementations.

In [69] Shelton et al. present the results of a robustness evaluation of Linux and several Win32 operating systems, namely Windows 95, 98, 98 SE, NT, 2000 and CE. The authors aim at setting a starting point for comparing certain dependability aspects across different platforms, thus, unlike the Ballista project, the focus is set on similar functionality (even if with different interfaces) across the different systems under test. To assess the robustness of these operating systems, the authors rely on an implementation of the Ballista robustness testing methodology [80].

The authors selected a set of systems calls from the Win32 API and another set of system calls from the Linux API of similar functionality (to allow for comparison). Certain system calls (e.g., graphics-oriented) were discarded in order to avoid targeting driver-specific code. Results showed that the APIs of Linux, Windows 2000 and NT are more resistant to exceptional inputs, with the remaining Windows family operating systems showing low robustness and often crashing. The approach revealed that, over time, developers have become more aware of the need for robust commercial operating systems (i.e., results show older Windows operating systems containing far more robustness problems), as these are the base upon which business, utility and even critical applications are executed, where consequences of robustness faults range from inconvenient to costly.

Ghosh et al. [92] present a black-box approach for assessing the robustness of Windows NT software using the Random and Intelligent Data Design Library Environment (RIDDLE) tool. RIDDLE generates random, invalid and boundary (i.e., around the upper and lower domain boundaries of a data type) inputs from an interface specification of the target system, with the generated inputs being syntactically valid, despite anomalous. The work targeted Windows NT system utilities and a port of GNU utilities, with the former showing fewer failures in presence of anomalous inputs.

Schmid et al. [93] describe two automated approaches for black-box testing the robustness

of Windows NT operating systems. The first is based on using a test data generator to analyze the robustness of Windows NT Dynamic Link Libraries (i.e., the Windows NT API). The data generator can create generic data (i.e., not dependent on the component under test) and intelligent data (i.e., test data targeting the component under test). While generic data can be used for any function, independently of parameter data types, one intelligent data generator only targets a specific data type (i.e., there must be as many intelligent generators as there are data types). These algorithms create valid, invalid and also boundary test input. The other approach uses the Failure Simulation Tool (FST) to evaluate the consequences of operating system robustness issues on running applications. FST sits between the target application and the operating system, and injects faults in return values of operating system calls performed by the application. Returned values include exceptions and error codes specific to the corresponding operating system function that was called.

A paper by Durães and Madeira evaluates the behavior of COTS operating systems in the presence of faulty device drivers through software fault emulation [88]. The authors use the Generic Software Fault Injection Technique (G-SWFIT), presented in more detail in Subsection 3.2.5, which essentially injects faults, in the machine code of the target system driver, that emulate common programming errors (e.g., missing or wrong variable initialization, missing function call). G-SWFIT was evaluated on the floppy disk and CD-ROM drivers of three versions of the Windows operating system (i.e., NT4 SP6, 2000 SP1 and XP).

A work by Albinet et al. [77] assesses the robustness of the Linux kernel to faulty device drivers, by injecting faults on the parameters of the kernel through the Driver Programming Interface (DPI). This works by intercepting driver calls to the target kernel functions and replacing the values of parameters with one of three corrupted values for each data type (i.e., the midpoint and the boundary values). The authors consider the data types integer, unsigned integer, unsigned short, and memory pointers. Mutations for memory pointers are the NULL value, the maximum pointer value or a random value. The testing procedure takes the following steps: i) uninstalling the driver under test; ii) installing it to ascertain successful driver registration; iii) performing a series of calls to test the driver; and iv) uninstalling the driver once more to ascertain successful driver unregistration. The authors evaluate robustness issues according to both the application and the driver perspectives.

In a work by Kanoun et al. [78] robustness is used as a measure in the definition of a dependability benchmark for Windows and Linux operating systems. In what concerns robustness, the evaluation process relies on intercepting system calls, and corrupting call parameters. Parameter mutation is data type-based, and generated values are either out of range or incorrect but within range (i.e., boundary and invalid values), although a set of predefined valid values are also considered. Pointer parameters are mutated to existing but incorrect values (e.g., a valid address with invalid data). The authors use a set of five qualitative dimensions to categorize test results, which cover hanging, error return, exception, panic state (i.e., the operating system is running but not servicing the application) and no-signal results.

Acharya et al. describe a framework for automatically infer system interface specifications from source code based on the combination of model checking and static analysis [76]. A model checker is first used to generate traces related to the interface of the system under test, and from these traces a set of interface-specific details are inferred, such as return values, and success or failure conditions. The resulting specifications are used to identify verifiable robustness properties inherent to the system under test, which is performed through static analysis. The authors implemented the proposed approach and evaluated

it on a set of 60 POSIX APIs, and in 22 the framework successfully inferred the interface specifications and was able to detect a total of 28 robustness problems.

Mendonça and Neves evaluate the robustness of Windows XP, 2003 server and Vista, when faced with erroneous input from faulty device drivers [91]. Unexpected values are injected in functions of the Windows Device Driver Toolkit (DDK), and the behavior of the operating system is observed. The types of values injected are meant to simulate acceptable values, missing initialization of variables, forbidden values (i.e., values explicitly documented as incorrect), out of bounds values, invalid pointers, null pointers and failure to call initialization functions which are required. The test results were classified with one of four failure modes, including *no failure*, *application/system hangs*, *system reboots* and *system reboots and files are corrupted*.

Johansson et al. study the sensitivity and accuracy of robustness evaluation results from injecting faults into calls that operating systems perform on drivers [90]. The approach works by intercepting calls made by the operating system to the target driver. Each available parameter of a given call is chosen, and each of its composing bits is flipped (i.e., all bits of all parameters are eventually flipped once). The call is then finalized the returned to the operating system, and its behavior is analyzed and classified in one of four classes: *no failure* (class NF); *deviation from golden run* (class 1); *specification is violated* (class 2); and *operating system is unresponsive due to crash or hang* (class 3). The authors performed an experimental evaluation of the approach on two device drivers (cerfio_serial.dll and 91C111.dll) on the Windows CE platform, and results show a high number of robustness problems.

Maji et al. evaluated the robustness of the Inter-Process Communication (IPC) mechanism in the Android mobile operating system through fuzz testing [73]. The approach is based on randomly generating *Intent* objects (used by the Android run-time as a data container for passing messages between processes) and assessing how robust Android processes are at handling them. The authors evaluated the approach on more than 800 different application components across 3 versions of the Android system, which reveals low quality exception handling code, the existence of environment-dependent errors, and, from a security perspective, the presence of privileged components with unbounded access.

Xiang et al. [87] describe a multi-layered approach for testing the robustness of Linux-based operating systems. The proposed approach injects faults (e.g., bit-flips considering just one bit, two bits, and 32 bits, minimum integer, illegal file pointer) at three distinct layers in the software under test, namely the API, the DPI and at the kernel level (i.e., system calls). Test results are classified with: *restart* - when a process restarts after a fault is injected); *abort* - when the process throws a fatal exception after a fault is injected); or *pass* - when the process does not behave abnormally after a fault is injected. The authors evaluated the approach on hundreds of functions of Redhat versions 6.0 and 9.0 and Redhat Enterprise Linux version 5.1. About 12% of test cases resulted in system aborts or restarts, with most issues being related to directory management, semaphores (i.e., synchronization), time and string handling system functions.

Ye et al. propose a tool, named DroidFuzzer, to test the robustness of applications in the Android portable operating system [75]. DroidFuzzer parses Activity objects (i.e., activities are Android objects that describe executable processes where user interaction is expected) and extracts input information. This information is used to automatically generate random inputs to use in robustness test cases. This data is provided as input to the respective applications and DroidFuzzer monitors them to detect if a crash occurs (i.e., robustness issues are detected automatically). The authors evaluated DroidFuzzer on three different Android applications. Results include the disclosure of 14 different bugs

across the tested applications, and the detection of one known vulnerability.

Sasnauskas and Regehr describe Intent Fuzzer, a framework for testing the robustness of applications in the Android operating system using a combination of static analysis and randomly generated tests [74]. The approach is based on extracting information regarding Intent objects from the target application, and performing static analysis on its bytecode to understand how these objects are accessed for reading their data. The collected information is then used to randomly generate Intent objects and are sent to the application under test. During testing, Intent Fuzzer monitors crashes and, if code is available (e.g., open-source applications), code coverage is monitored as well. The approach was evaluated on 10 well-known applications, triggering crashes and restarts, and operating system reboots.

Cong et al. present in [70] an approach for automatically testing the robustness of operating system device drivers, which works by running tests on specific drivers and collecting execution traces. The collected trace information is automatically analyzed to identify which of the functions called may cause execution to fail. For each identified function, a fault scenario (i.e., a test case) is generated, containing a set of faults to inject. The proposed method for fault injection is based on intercepting all calls done to the operating system kernel, and returning erroneous results to the calling device driver. The approach implements a feedback technique for increasing test coverage, which is based on actively attempting to disclose new fault scenarios during testing. The authors evaluated 12 different drivers, with focus on three types of functions (memory allocation, memory map and Direct Memory Access (DMA), and Peripheral Component Interconnect (PCI) interface), and were able to disclose a total of 28 severe bugs.

Feng and Shin performed robustness testing on the IPC mechanism [72], a subsystem of the Android mobile operating system for facilitating processes to communicate securely with each other. This occurs via Remote Procedure Calls (RPC) and assuming a client-server logic. The authors exploit the fact that this mechanism trusts client processes by default, and does not properly verify their call parameters. BinderCracker, the tool described by the authors, is capable of fuzzing transactions (i.e., inject input with random values) of this mechanism. The authors evaluated their approach on six major versions of the Android system, covering over 170 services. Results show that BinderCracker disclosed more than 100 vulnerabilities, including cases of privileged code execution and Denial-of-Service (DoS).

Kernel-based fault Injection Tool Open-source (KITO) is a fault injection tool for evaluating the effects of faults in memory containing data structures from UNIX operating systems, specifically the structures related to resource synchronization management [71]. KITO is based on the concepts presented in earlier studies by the authors [83, 84]. Essentially, a module must be first inserted into the kernel (i.e., to obtain higher execution privileges), which activates a timer upon being loaded. When the timer ends, the module performs a bit-flip at a specific kernel-space memory address. The target memory address is customizable, and three options exist: a user-defined address; a UNIX kernel symbol - the module automatically obtains the base address of a UNIX data structure; process descriptor - a *task_struct* data structure with relevant process synchronization information used by the operating system scheduler subsystem. The authors evaluated KITO on four applications with different I/O, memory and processing requirements. The results show that the synchronization module of the Linux kernel is sensitive to a large set of soft errors, resulting in severe performance degradation or crashes of the application issuing the system calls.

In summary, work on operating system robustness assessment has first targeted the kernel, perhaps most notably with the work carried out within the Ballista project [3, 19, 69,

79, 80]. The usual cross-cutting concept is the presence of a testing tool that exercises the kernel via a certain API (e.g., POSIX) and uses a combination of valid and invalid inputs (e.g., boundary value inputs, anomalous, infrequent) to carry out the tests. The first works on robustness testing shared similar components (even if implicitly, and besides the experimental procedure), namely the use of a workload (e.g., calls to the system with valid values to assess its normal behavior), a faultload (e.g., a set of faults to inject on the parameters of system calls), and a way to classify failures (e.g., the CRASH scale). Some difficulties started to became obvious, namely the need for having good quality workload (e.g., with high coverage of functionality or code), or the difficulty in identifying certain failures (which imply using the right observation point and possessing a high quality oracle).

The evolution of robustness assessment for operating systems took a few next relevant steps, such as its application to multi-version software, allowing to evaluate the diversity of implementations of a single standard (in terms of robustness) and also their degree of conformance to standards [19, 80], a concept which has also remained in use throughout the years in several studies [78, 87, 91]. This type of assessment procedure was then evolved with the intention of allowing comparison by considering functional groups [69, 82]. Some focus started to appear on obtaining better workloads [92, 93] and focus started moving out of the kernel to target other parts of the operating systems, such as libraries [93], utilities [81, 82, 89, 92], or drivers [70, 77, 88]. In [82], Miller et al. noted that in over a decade of research, the same programming mistakes were continuously observed [68, 81, 89], at least in what concerns the robustness of operating system utilities.

More recently attention has been shifted to mobile operating systems [72, 73, 74]. This emphasizes the usefulness of robustness evaluation techniques, with each of the presented works showing the challenges and the needs to define specific evaluation techniques required to assess the robustness of new systems.

3.2.2 Communication systems

In this subsection, we describe research that focuses on robustness evaluation of communication systems (e.g., network protocol implementations). Considering that most current software systems make use of network communication, including some cases of truly network-centric devices (e.g., sensor networks), robust communication is an essential asset in many scenarios, for which research has been carried out throughout the years. Table 3.4 presents a short overview of the research found addressing communication systems, including the target TCP/IP layer [94], techniques used and their targets, and the types of faults applied. Some of the fault types relate to moving, adding, changing, or deleting data (e.g., removing an element from an array), which we identify using the Move, Add, Change, Delete (MACD) acronym. Note also that some of the works use just part of the four MACD faults, but for simplicity we refer the whole model.

Kaksonen et al. [20] presented, in 2001, an approach for testing the robustness of Wireless Application Protocol (WAP) implementations through a fault injection method which targets messages exchanged between systems. The approach focuses on robustness, however the authors intend to assess the level of security provided by WAP implementations and link robustness problems to security concerns. The approach requires a specification of the protocol modelled as a Backus–Naur Form (BNF) specification, which allows to understand the available message elements that will be used as fault injection targets. Fault injection is applied through four mutation operators that can *move*, *add*, *change*, or *delete* elements (MACD). These elements refer to message fields as well as to message sequences. The approach was applied to seven WAP gateways, disclosing severe robustness problems,

Table 3.4: Techniques for evaluating the robustness of communication systems

TCP/IP layer	Application	Cavalli et al. [95], Fu and Koné [21], Kaksonen et al [20], Naceur et al. [96], Popovic and Kovacevic [97], Rollet and Salva [98], Vasan and Memon [99], Xiang et al. [100], Xu et al. [101]
	Link	Jing et al. [102]
	Network	Johansson et al. [103]
	Transport	Saad-Khorchef et al. [4]
Techniques	Fault injection	Cavalli et al. [95], Jing et al. [102], Kaksonen et al [20], Vasan and Memon [99], Xiang et al. [100], Xu et al. [101]
	Fuzzing	Johansson et al. [103]
	Interception	Cavalli et al. [95]
	Model-based analysis	Cavalli et al. [95]
	Model-based testing	Cavalli et al. [95], Fu and Koné [21], Jing et al. [102], Johansson et al. [103], Kaksonen et al [20], Naceur et al. [96], Popovic and Kovacevic [97], Rollet and Salva [98], Saad-Khorchef et al. [4], Vasan and Memon [99]
	Mutation testing	Jing et al. [102], Saad-Khorchef et al. [4], Xu et al. [101]
Targets	Message fields	Cavalli et al. [95], Jing et al. [102], Kaksonen et al [20], Rollet and Salva [98], Vasan and Memon [99], Xiang et al. [100], Xu et al. [101]
	Messages	Cavalli et al. [95], Jing et al. [102], Johansson et al. [103], Kaksonen et al [20], Vasan and Memon [99], Xiang et al. [100]
	Specification	Fu and Koné [21], Saad-Khorchef et al. [4]
	System model	Jing et al. [102], Naceur et al. [96], Popovic and Kovacevic [97], Xu et al. [101]
Faults	Boundary inputs	Cavalli et al. [95], *Jing et al. [102]
	Inopportune inputs	Fu and Koné [21], *Saad-Khorchef et al. [4]
	Invalid inputs	Cavalli et al. [95], Fu and Koné [21], Jing et al. [102], Popovic and Kovacevic [97], Rollet and Salva [98], *Saad-Khorchef et al. [4], Vasan and Memon [99], Xiang et al. [100], Xu et al. [101]
	Invalid messages	*Xiang et al. [100]
	Invalid outputs	*Saad-Khorchef et al. [4]
	MACD	Cavalli et al. [95], Jing et al. [102], *Kaksonen et al [20], Vasan and Memon [99], Xu et al. [101]
	Random inputs	*Johansson et al. [103]
	Timing faults	*Cavalli et al. [95], Naceur et al. [96]
Classification	Binary	Cavalli et al. [95], Fu and Koné [21], Jing et al. [102], Johansson et al. [103], Kaksonen et al [20], Naceur et al. [96], Popovic and Kovacevic [97], Rollet and Salva [98], Saad-Khorchef et al. [4], Vasan and Memon [99], Xiang et al. [100], Xu et al. [101]

including several security issues.

Automated Systematic Protocol Implementation Robustness Evaluation (ASPIRE) is an automated approach for assessing the robustness of network protocol implementations [99]. It considers two types of protocol message faults: i) syntactically faulty messages; and ii) semantically faulty messages (e.g., incorrect message ordering). To generate a protocol message of the former type, its fields are mutated. To generate semantically faulty messages, all reordering possibilities are considered. To test stateless network protocols, ASPIRE uses syntactic faults, and to test stateful network protocols, it uses semantic faults (i.e., because altering the order of messages is more prone to erroneously influence protocols that hold state). The authors applied their approach to different HTTP and Simple Mail Transfer Protocol (SMTP) servers, showing the different level of robustness of the different implementations.

Saad-Khorchef et al. [4] propose a framework for generating robustness test cases for communication systems, which is centered around a tool named Robustness Test Cases Generator (RTCG) and is represented in Figure 3.2.

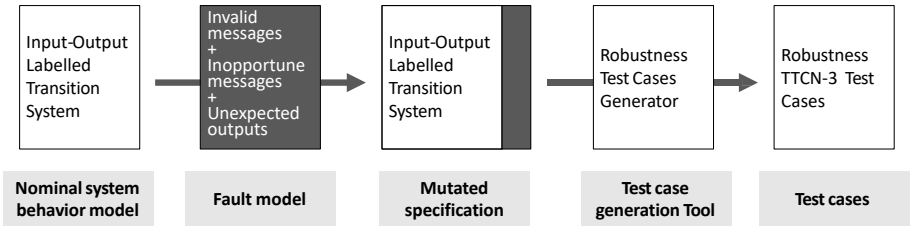


Figure 3.2: TTCN-3 robustness test case generation in [4].

From a specification of the nominal system behavior (i.e., behavior under normal conditions), written in the Specification and Description Language (SDL) and represented in the form of an Input-Output Labelled Transition System (IOLTS), a mutated version is obtained, called the *increased specification*. This mutated specification contains invalid and inopportune inputs and unexpected outputs, and is used as a basis for robustness test case generation. The tool outputs test cases in the Testing and Test Control Notation-3 (TTCN-3) format.

Popovic and Kovacevic [97] present a model-based statistical approach for testing communication protocols through Finite State Machines (FSM) with hidden states and transitions. In FSMs, hidden states and transitions are those which are not observable in the model but can nevertheless occur, and typically represent erroneous behavior. Hidden states represent invalid states of the modelled system, and hidden transitions are the events (triggered by invalid input) that may set the model in either a valid or invalid state. The approach calculates the probability of transiting from a given observable state to a hidden one based on all the possible transitions (i.e., each source-target state pair). This effectively generates a set of probabilities for transiting from valid to invalid states. A non-robust system is thus one whose corresponding FSM model allows such state transitions.

Jing et al. [102] propose a data-driven approach, using invalid inputs, to test the robustness of network protocol specifications and implementations through the Nondeterministic Parameterized Extended Finite State Machine (NPEFSM) model. This model formally describes network protocols, including their messages and respective fields as well as processing rules, and is used to generate robustness test cases by performing mutations on these elements. Message fields are made invalid by using boundary values, injecting formatting errors, or changing length or checksum parameters, and messages are mutated through addition, removal or sequence permutation (i.e., changing message order). The authors consider single and multi-field faults. Finally, test cases are generated in the form of the Protocol Integrated Testing System version 3 (PITSv3), an extension of the TTCN-3 standardized test specification and implementation language.

Cavalli et al. describe in [95] a model-based, fault injection, passive testing approach for testing the robustness of communication protocol implementations. The approach is based on the following steps: i) model the behavior of the system in the form of an FSM from a specification of the system under test; ii) manually extract invariant properties from the FSM model (i.e., properties the specification of the system should satisfy); iii) definition of a fault model, in which the authors consider three types of external faults, namely *omission* faults in which incoming messages are intercepted and removed, *arbitrary* faults based on parameter corruption of received messages (e.g., parameters are replaced with limit values), and *timing* faults, emulated by delaying message delivery; iv) placing instrumentation in the system under test capable of injecting, at run-time, the faults defined in the previous step; v) test execution and trace collection (i.e., traces serve as input to Invariant Analysis,

the passive testing method used, which is based on the invariants identified in step *ii)*); and vi) traces are analyzed resulting in a *pass* or *fail* verdict depending on whether invariant properties are satisfied, with the possibility of an *inconclusive* verdict if a trace lacks information. The approach was tested on an implementation of the WAP protocol, where a hang of the WAP gateway was observed.

Xu et al. [101] present a TTCN-3-based approach for automatic robustness test case generation. As mentioned previously, TTCN-3 is a standardized test specification and implementation language used in communication systems for functional testing, which the authors convert to robustness testing specifications through predefined mutation rules. A test model of the system under test is first obtained by parsing TTCN-3 test suites. This model is then mutated through a set of predefined, data type-specific mutation operators (e.g., replace integer value, delete or reorder elements in a set). The mutated model parameters are those which the authors consider to hold higher probability of disclosing robustness problems, and the operator to use for mutating each parameter is selected from a i) random, ii) round-robin, or iii) weighted random selection. The authors have used their approach to evaluate the robustness of different SIP implementations, triggering several crashes and anomalous behaviors.

Xiang et al. [100] propose a method for testing the robustness of SIP implementations based on TTCN-3 specifications. The approach is based on automatically generating anomalous messages, through message parameter mutation, to deliver to the SIP implementation under test. The types of parameter mutations considered by the authors consist of injecting special characters or invalid bytes, changing numerical values to invalid ones, or even changing the position of *SPACE* and *CRLF* separators (important delimiters in SIP messages). The authors tested a set of 4 different open-source SIP implementations, and only a single test case of one of the tested implementations signaled a robustness issue, revealing the tested software is quite mature and is robust to invalid inputs.

Rollet and Salva present a formal method for assessing the robustness of communication systems [98]. The proposed approach first models the system under test as an IOLTS, which is analyzed to find inconsistent states within the target system (e.g., a known input but which is not expected at a given state, or an unknown input), which should in turn lead to blocking states (e.g., deadlocks or livelocks). Finally, this generates a set of test cases which effectively exercise two behaviors: i) applying inputs known to the system but not expected at particular states; and ii) applying inputs that are unknown to the system.

Johansson et al. [103] introduce T-Fuzz, a model-based framework for testing the robustness of telecommunication protocols through fuzzing. T-Fuzz extends the TITAN conformance testing framework [104], which relies on TTCN-3 specifications of telecommunication protocol implementations to generate and execute C++ test suites. T-Fuzz thus adds another layer to this framework containing the following components: the *model extractor* extracts a model from the TTCN-3 specifications, which hold information regarding the data types and data structures used by the protocol implementation under test; then the *fuzzing engine* creates C++ functions for generating instances of the identified data types as well as complex data structures through fuzzing logic (i.e., generating random values according to their data types), which are fed back to TITAN for generating a test suite; finally, the *observer* component monitors the implementation under test while TITAN executes the test suite. The approach has been applied to a Non-Access Stratum (NAS) protocol implementation triggering several cases of unwanted behavior.

Naceur et al. [96] present an approach for generating robustness test cases that target Wireless Sensor Network (WSN) protocols, specifically the Localized Encryption and Authentication Protocol (LEAP). The authors first model the nominal behavior of the sensor

under test, which uses the LEAP protocol, in the form of a Timed Automata (TA), a type of FSM that includes timing-related data. Non-robust sensors in WSNs are those whose respective TA are unprotected against live or deadlocks, which consequently delay communication between sensors in the network (which is unwanted in real-time systems) or eventually drain the sensor's battery, degrading or ultimately breaking the entire sensor network. Thus, the authors enhance the TA model of the nominal system behavior with tracing capabilities to detect live and deadlocks.

Fu and Koné evaluate the robustness of network protocols in a model-based approach presented in [21]. The network protocol implementation under test is modelled as an IOLTS based on its specification. By default, this specification should cover some input actions, but the approach extends this set in the obtained IOLTS model to include other input types, namely acceptable, inopportune (e.g., acceptable inputs sent in the wrong system state) and invalid inputs. Test cases are automatically extracted from the resulting IOLTS model, and run against the protocol implementation under test. The authors evaluated the approach on an implementation of the Remote Authentication Dial-In User Service (RADIUS) protocol, and found no abnormal behaviors (e.g., blocked execution, crashes).

In summary, the research discussed in this subsection handles mostly network-centric systems or protocols, with most cases actually targeting the Application layer of the TCP/IP reference model. In the application layer, the SIP protocol was found to be a frequent case of study [97, 100, 101], with several approaches applying faults to both messages and message fields. Other approaches focus different protocols such as the WAP protocol [20, 95] (common in portable devices), the NAS protocol [103] (a functional layer in wireless telecommunication protocols), or even the RADIUS protocol [21], for which the robustness and security needs are evident.

The majority of the analyzed studies rely on model-based approaches, where a specification of the protocol under test is modelled (e.g., as an FSM) [96, 97, 98, 102]. This model can then be used to generate test cases [4, 20, 95], or be directly evaluated to find undesired states (e.g., deadlocks) [96]. We found the use of the TTCN-3 specification to be very frequent (i.e., for the definition test cases).

The approaches that use faults applied to protocol message fields tend to use purely invalid values (e.g., out-of-bounds values) [95, 99, 100, 101, 102], whereas the approaches that target messages tend to rely on other types of faults that affect the whole message, namely by applying some variation of the MACD operations (e.g., reordering or omitting messages) [20, 99, 102]. We found timing faults [95] or inopportune faults [4] to be less explored in robustness evaluation research, despite their usefulness in respectively triggering timing failures or in triggering failures that can only be observed after placing the system in a certain state.

3.2.3 Embedded systems

In this subsection, we present research that evaluates the robustness of embedded systems, which are systems that are typically designed to target a specific task (in contrast with a general purpose computer) and are often part of larger systems that serve a more general purpose. Usually, their operation has to fulfill timing constraints (e.g., real-time systems) or deal with safety properties (e.g., in the aerospace domain) [59, 60]. Table 3.5 overviews the main characteristics of the different approaches for embedded systems found in the literature.

Table 3.5: Techniques for evaluating the robustness of embedded systems

Systems	Aerospace	Ait-Ameur et al. [105], Batista et al. [5], Dingman et al. [22], Mattiello-Francisco et al. [24], Yang et al. [106]
	Embedded distributed systems	Alnawasreh et al. [107]
	Microkernels	Arlat et al. [108]
	Reactive	Rollet and Salva [109]
	Real-time systems	Fouchal et al. [110, 111], Mattiello-Francisco et al. [112], Rollet and Saad-Khorchef [113], Tarhini et al. [114], Winter et al. [115]
	RTOS	Cotroneo et al. [116, 117], Madeira et al. [118], Maia et al. [119], Nicolescu et al. [120], Rodríguez et al. [121], Ruiz et al. [122], Shahpasand et al. [123, 124], Zhou et al. [125]
Techniques	Code changes injection	Winter et al. [115]
	Fault injection	Alnawasreh et al. [107], Arlat et al. [108], Batista et al. [5], Cotroneo et al. [116, 117], Dingman et al. [22], Fouchal et al. [110, 111], Madeira et al. [118], Maia et al. [119], Nicolescu et al. [120], Rodríguez et al. [121], Rollet and Saad-Khorchef [113], Rollet and Salva [109], Ruiz et al. [122], Shahpasand et al. [123, 124], Tarhini et al. [114], Winter et al. [115], Yang et al. [106], Zhou et al. [125]
	Interception	Ruiz et al. [122]
	Model-based analysis	Ait-Ameur et al. [105]
	Model-based testing	Cotroneo et al. [116, 117], Fouchal et al. [110, 111], Mattiello-Francisco et al. [24, 112], Rollet and Saad-Khorchef [113], Rollet and Salva [109], Shahpasand et al. [123, 124], Tarhini et al. [114], Yang et al. [106]
	Mutation testing	Mattiello-Francisco et al. [112]
Targets	API calls	Ruiz et al. [122], Winter et al. [115]
	Kernel segments, CPU registers	Arlat et al. [108], Madeira et al. [118], Nicolescu et al. [120], Rodríguez et al. [121]
	Message fields	Batista et al. [5]
	Messages	Alnawasreh et al. [107], Batista et al. [5]
	Specification	Fouchal et al. [111], Mattiello-Francisco et al. [112], Rollet and Saad-Khorchef [113], Rollet and Salva [109]
	System calls	Ait-Ameur et al. [105], Arlat et al. [108], Cotroneo et al. [116, 117], Dingman et al. [22], Maia et al. [119], Rodríguez et al. [121], Shahpasand et al. [123, 124], Zhou et al. [125]
	System model	Fouchal et al. [110], Mattiello-Francisco et al. [24], Tarhini et al. [114], Yang et al. [106]
Faults	Bit-flips	Arlat et al. [108], *Madeira et al. [118], Nicolescu et al. [120], Rodríguez et al. [121], Ruiz et al. [122], Zhou et al. [125]
	Boundary inputs	*Zhou et al. [125]
	Invalid inputs	Batista et al. [5], Cotroneo et al. [116, 117], *Dingman et al. [22], Fouchal et al. [110, 111], Maia et al. [119], Rollet and Saad-Khorchef [113], Rollet and Salva [109], Ruiz et al. [122], Tarhini et al. [114], Winter et al. [115], Yang et al. [106]
	Invalid outputs	*Rollet and Salva [109]
	Random inputs	*Shahpasand et al. [124]
	Random messages	*Alnawasreh et al. [107]
	Round-off errors	*Ait-Ameur et al. [105]
	Timing faults	Alnawasreh et al. [107], Batista et al. [5], *Fouchal et al. [110, 111], Mattiello-Francisco et al. [24, 112], Rollet and Saad-Khorchef [113], Rollet and Salva [109], Shahpasand et al. [123, 124], Tarhini et al. [114], Winter et al. [115], Yang et al. [106]
Classification	8 categories	Nicolescu et al. [120]
	6 categories	Dingman et al. [22]
	5 categories	Cotroneo et al. [116], Madeira et al. [118], Maia et al. [119], Shahpasand et al. [124]
	4 categories	Cotroneo et al. [117], Ruiz et al. [122], Zhou et al. [125]
	3 categories, 7 subcategories	Arlat et al. [108]
	Binary	Ait-Ameur et al. [105], Alnawasreh et al. [107], Batista et al. [5], Fouchal et al. [110, 111], Mattiello-Francisco et al. [24, 112], Rodríguez et al. [121], Rollet and Saad-Khorchef [113], Rollet and Salva [109], Shahpasand et al. [123], Tarhini et al. [114], Winter et al. [115], Yang et al. [106]

Figure 3.3 represents a common case of robustness evaluation of an embedded system. The figure has been adapted from [5], where a real-time aerospace system is evaluated, and has been modified and extended for clarity.

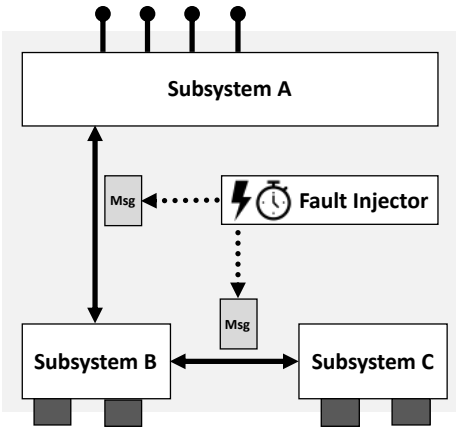


Figure 3.3: Robustness evaluation of an embedded system (inspired by the approach in [5]).

As we can see, Figure 3.3 shows a system composed of three subsystems named A, B, and C. Subsystem A serves as an interface to the outside world and partially relies on subsystem B, which in turn relies on subsystem C. B and C have actuators that interact with the physical world. A fault injector (part of a testing framework) has been added to the system with the purpose of intercepting messages and applying faults. Typical cases of faults are messages holding invalid contents, but in this context, there is special interest timing faults (e.g., delayed messages) so that it is possible to verify if the system fulfills the expected real-time properties (in the presence of a timing fault affecting a subsystem).

In 1995, Dingman et al. presented a tool for evaluating the robustness of Advanced Spaceborne Computer Module System (ASCM), a fault tolerant aerospace system, focusing on file, memory, and IPC subsystems [22]. Each generated test case targets a system call, and for each call valid values and common erroneous (i.e., invalid) values are passed as parameters. A test suite contains one test case for each parameter value combination, and all combinations are tested. The system behavior is classified in terms of severity classes, with class 0 corresponding to correct return codes, class 1 being an unexpected error code being returned by the system call, and so on, with class 5, representing cases where the system must be forcefully restarted. Most robustness problems observed were marked with class 4 (the second most serious).

Madeira et al. evaluate the impact that transient faults in a COTS application have on its operating system (i.e., LynxOS, a Unix-like Real-Time Operating System (RTOS)) as well as on its applications [118]. The authors used Xception [126], a fault injection tool for emulating physical faults, for injecting bit-flip faults at multiple locations of the target system (e.g., processor registers, integer unit, internal processor buses, memory, cache). The authors evaluated the approach on a synthetic workload (i.e., generated specifically for testing) composed of buffer and matrix manipulations, and on a realistic workload (i.e., produced by the execution of real applications) composed of three applications: gravity (Netwon's gravity law calculator), PI (computes the value of π) and Matmult (matrix multiplication program). Results were categorized as: *OS crash*, *application hang*, *abnormal application termination*, *no impact* or *wrong results*.

Arlat et al. present the Microkernel Assessment by Fault injection AnaLysis and Design

Aid (MAFALDA), a tool for assessing the behavior of microkernels in the presence of faults [108]. This approach tests microkernels from both the external and internal perspectives. The former is based on injecting faults in the parameters of calls to the microkernel API, and assessing its robustness at the interface level with respect to external faults. Faults are injected by randomly selecting and flipping one bit among the set of call parameters. The internal perspective relies similarly on bit-flips, but rather on the microkernel code and data segments within its address space, and intends to simulate both physical and (development-related) software faults in order to assess its coverage of internal error detection mechanisms. A fault injection campaign performed by MAFALDA is composed of: fault-less workload execution to assess normal operating behavior; this is followed by the faultload execution where the workload is executed and faults are occasionally injected; finally, the behavior of the target system is observed for robustness issues, and is categorized in three distinct levels: *application level* - one of application failures or application hang; *interface level* - one of error status or exception; and *kernel level* - one of kernel hang or kernel debugger. MAFALDA was evaluated on the Chorus and on the LynxOS microkernels.

Rodríguez et al. have updated the MAFALDA approach to support real-time systems, and propose the MAFALDA-RT (MAFALDA Real-Time) tool for dependability assessment of real-time systems [121]. The major difference from this work to the MAFALDA approach is that of effectively cancelling out the temporal overhead related to fault injection, which does not align well with the temporal constraints of real-time systems. To this end, MAFALDA-RT surpasses this restriction by temporarily stopping system *interrupts* and using a virtually unlimited time window for performing fault injection. This is possible because the state of real-time systems is entirely driven by *interrupts* from the hardware clock, which notify the software system of the passing of time. Thus, their absence is perceived by the system as time having stopped, and its state is effectively frozen. The approach was evaluated on a mine drainage control application running on a Chorus microkernel.

Ait-Ameur et al. present a formal approach for understanding the reaction of a flight control system to inaccurate input values [105]. The authors analyze error propagation in the system under test through the *abstract interpretation* technique, a formal program analysis technique which consists of running, on an abstract domain, a correct and safe approximation (i.e., abstract program) of the program under test. The goal is to analyze the amount of cumulative error that successive function calls produce in numeric parameters (e.g., due to the rounding of values in decimal parameters), and consider the system to be robust if the difference between inputs and outputs is under a certain threshold.

Maia et al. describe an automated fault injection and robustness testing framework for mission and business-critical software systems and components in [119], which is based on Xception, an approach and tool for emulating physical faults in memory and addresses by performing low-level bit-flips [126]. The framework is able to inject both valid and invalid inputs into the interface of the system under test. The injected values depend on the data types of the parameters used by the system. The system behavior observed during testing is categorized with the CRASH scale. Evaluation was performed against the Real-Time Executive for Multiprocessor Systems (RTEMS), which is an operating system designed for embedded systems.

A methodology for assessing the robustness of real-time component-based systems was presented in 2005, by Fouchal et al. [110], a slightly updated version of the concepts the authors presented in an earlier paper of the same year [114]. The approach begins by representing each system component in the form of a Timed Input-Output Automata

(TIOA). Test sequences are generated from these automaton models, and are consequently mutated through fault injection (e.g., replacing an input by another, change input timings, removing or modifying transitions), as a way of simulating faulty component behavior. Test cases assess robustness of each individual component and the correctness of communication between components in the system.

Nicolescu et al. have studied the behavior of the MicroC kernel, a popular RTOS, in the presence of faults in its task scheduling and context switching modules [120]. The proposed approach is based on performing bit-flips in the processor registers at random instants, while these components are active. The authors also propose a fault syndromes scale for categorizing faults observed in safety-critical systems: *effect-less* - no observable effect; *application hang* - system stops responding; *exception* - exception is triggered; *memory access dysfunction* - system tries to access invalid memory address; *system crash* - the system crashes; *incorrect output results* - system provides incorrect results; *real-time problem* - time constraints specified for the system are not respected; *scheduling dysfunction* - incorrect scheduling strategy of tasks by the system. This approach was evaluated on an in-house real-time multitasking application composed of 36 periodic tasks.

Ruiz et al. present an approach for assessing the robustness of System-on-a-Chip (SoC) embedded systems to external faults, using on-chip debugging capabilities [122]. The idea is to emulate external faults by intercepting and corrupting, using fault injection, the data received by the target system through its public interface. Data corruption is performed by identifying one of the parameters of the intercepted interface call, and performing either a bit-flip or replacing the entire value with an invalid, special (e.g., a value representative of a special case) or custom (i.e., user-defined) value. This approach categorizes test results in a four-category scale as *correct output*, *wrong result*, *system hang* and *exception*. The authors evaluated the approach on a component of a real-time operating system, and identified a large number of uncaught exceptions, and some wrong results and system hangs.

Rollet and Saad-Khorchef propose an approach for testing the robustness of embedded systems [113]. The idea is to use two formal specifications of the system under test, a nominal one which describes the expected behavior under normal conditions, and a degraded specification for describing how the system functions under critical conditions. Both specifications are modelled in the form of a Labelled Transition System (LTS) in case the system under test has no timing constraints, or as a TA otherwise. From these specifications (i.e., the nominal and the degraded one), test sequences (i.e., sequences of actions) are extracted, which are then injected with faults. The injected faults are actions which the system under test will perceive as unexpected, and erronous timing constraints, if applicable. The mutated test sequences (i.e., those injected injected with faults) are then executed on the system under test. The authors did not perform experimental evaluation of the proposed approach.

Rollet and Salva present two approaches for testing the robustness of embedded software systems in [109]. The first approach uses a system specification modelled as an IOLTS and a set of unexpected events to generate an *increased specification* which extends the nominal specification to include execution paths to exploit possible robustness gaps (e.g., timeouts, unexpected inputs or outputs). The second approach uses a nominal behavior specification and a degraded one, which describes the minimal system behavior under critical conditions. By integrating faults in test sequences, the authors aim at understanding if the minimal behavior is still respected (even if conformance to the nominal behavior is lost).

Mattiello-Francisco et al. propose a model-based approach for generating and executing robustness test cases on timed embedded systems [112]. The approach is composed of three main steps, namely: i) service modeling, wherein the nominal service model (i.e.,

that which describes expected behavior) is extended with timing deviations (i.e., timing faults) in order to expose timing-related robustness issues in the implementation, and which is modelled as a TIOA; ii) generation of robustness test cases based on the definition of a test purpose (i.e., a specific system characteristic that a tester wishes to assess); and iii) execution of the generated test cases over a real implementation of a timed embedded system. The proposed approach was not experimentally evaluated, but rather showcased on a small case study of a subsystem integration of a real-time space X-ray telescope system on board a scientific astronomy satellite.

Fouchal et al. propose a model-based approach for testing the robustness of a real-time component-based system [111]. The target system is described as a collection of components, each modelled as a TIOA. For each component, a nominal (i.e., describing expected behavior) and degraded specifications are necessary. The nominal specification is used to extract test sequences, which are injected with faults in order to simulate hostile environments. The faulty test sequences of each component are executed (i.e., on the target system) in isolation, and results are recorded. Robustness is then checked by verifying if the recorded results are accepted by the degraded specification of each tested component. The injected faults are actions which the system under test will perceive as unexpected and erroneous timing constraints. The authors did not perform experimental evaluation of the proposed approach.

Cotroneo et al. [116] assess the impact of robustness faults on the system state of a Linux RTOS for safety-critical systems in the avionics domain. The proposed framework is composed of a *test driver* which is responsible for injecting invalid inputs to the interface of the system under test, and a *state setter* which is responsible for driving the system into a certain state (state is previously modelled). Accounting for the state of the file system helped improve test results and reach corner cases with complex interactions with other subsystems (e.g., memory management, caching, scheduling).

Mattiello-Francisco et al. describe InRob, a model-based approach for verifying robustness and interoperability properties of real-time embedded software systems related to timing constraints [24]. The testing process of InRob is composed of three phases: i) service modelling - based on the functional requirements, time constraints and service profile of the system, its nominal behavior is abstracted and modelled as a TIOA. The *augmented model* is then obtained by extending the nominal behavior model with timing deviations (i.e., *timing hazards*), emulating communication channel failures; ii) test case generation - in this phase the augmented model is manually written in a formal language accepted by the test case generator, allowing test cases to be generated automatically; iii) test case execution - finally, the test cases generated in the previous phase are executed on the target system. The authors evaluated InRob in a real satellite subsystem.

Yang et al. focus on the robustness of safety-critical avionics-embedded software, by proposing a model-based robustness testing approach [106]. It is based on the conversion of communication protocols of avionics systems (e.g., interface, data types, timing constraints) to an abstract testing model. This testing model is used to generate both normal and boundary testing data, constituting a workload. This workload is then injected with timing, state, data and protocol failures, which simulate abnormal behavior of the components in the system. The authors implemented this approach in a tool called Model-based Robustness Testing Environment (MbRTE), and evaluated it on two flight control and two inertial navigation software systems, all of which are real world avionics applications.

In [117], Cotroneo et al. introduce the State-Based Robustness Testing of Operating Systems (SABRINE) approach, which automatically extracts models of operating system

states from execution traces and generates a set of test cases covering multiple system states. The SABRINE approach consists of the following steps: i) system behavior data collection through execution of a workload with valid inputs; ii) pattern identification through grouping of identical event sequences (i.e., sets of interactions that occur during the execution of individual system calls); iii) clustering of similar patterns (i.e., event sequences) identified in the previous step; iv) for each pattern cluster, generate a behavioral model in the form of an FSM and identify state transitions (i.e., function calls) wherein faults can be injected; and v) generate test programs from test cases and execute them. Furthermore, the authors present a four category robustness classification scheme based on observed results from applying SABRINE to a Linux-based RTOS used in the aviation domain. Test outcomes were categorized as one of *kernel failure*, *workload failure*, *file system corruption* or *no impact*.

Zhou et al. test the robustness and fault tolerance of three popular RTOS through a Ballista-based approach, which works by injecting faults at the RTOS API. [125]. The authors focus only on the API functions that relate to four basic components of an RTOS, namely task management, task synchronization, memory management, and timer management. The proposed approach injects one of two fault types in the parameters of these functions: bit-flips and data type-based parameter corruption. The latter occurs only for bounded data types (i.e., numerical parameters), and injects boundary values, the original value plus or minus 1, or the values 1, -1 and 0. Observed faults are categorized with: *detected failure* - the RTOS detects an error and stops it from spreading; *silent failure* - a fault is triggered but the system does not detect it; *hang failure* - the task calling that specific API function hangs while other tasks operate normally; *crash failure* - the whole RTOS crashes and requires a restart.

Winter et al. present GeneRic fault INjection tool for DEpendability and Robustness assessments (GRINDER), an open-source fault injection tool for testing the robustness of software systems [115]. GRINDER uses a probe-based approach, which requires instrumenting the application under test with communication channels to allow the tool to inject faults as well as monitor the behavior of the application. Additionally, this can only be done at compile time, with access to the source code of the target application. Regarding the fault model, GRINDER does not impose any restrictions, and users of the tool may employ any fault types and triggers they wish. The approach was evaluated on an AUTomotive Open System ARchitecture (AUTOSAR) adaptive cruise control system and the Android mobile operating system, with the authors reporting on the testing effort required to use GRINDER.

In 2016, Shahpasand et al. [123] propose the TIMEOUT approach for robustness testing of RTOS through state-aware fault injection. First, a workload is executed several times for obtaining behavioral data from the target system, and input-output interactions as well as specific time-related metrics are logged. Logged calls to the same kernel function are grouped into clusters, based on the assumption that individual calls may relate to different event sequences (i.e., for tracing different call sequences). For each cluster, the logged execution start and end times of the kernel function calls are used to infer a TA, whose states represent inferred states of the operating system, which are connected by the observed events labelled with timing data. Test cases are then generated by injecting faulty transitions into the generated TA. Faulty transitions are those that erroneously influence execution path by changing system state to an unexpected one (i.e., as a consequence of delays, for example). In a recent paper, the authors consider additional types of faults [124], such as value faults (e.g., random values depending on the data type of the target parameter). Also, the authors identify the critical states of the system under test (i.e., states that are important for system dependability) and attempt to quantify its minimum

and maximum level of robustness with the aid of weights attributed to the CRASH scale [3].

A work by Alnawasreh et al. [107] describes a fault injection approach, named Postmonkey, for testing the robustness of industrial distributed embedded systems with limited computational power. The proposed fault types are: i) sending messages with random contents; and ii) delaying transiting messages. The former fault type is intended to assess the fault tolerance level of implementations of the message protocol used for communication, by targeting the message origin and contents. The latter fault type has the goal of testing how well the overall system handles delays. The authors implemented the Postmonkey approach in C++, and evaluated it on the embedded distributed system of a Radio Based Station.

Batista et al. present the Failure Emulator Mechanism (FEM) framework for robustness testing of interoperable software-intensive subsystems of nanosatellites in [5]. The framework integrates with communication channels of nanosatellite subsystems and tests them through fault injection using invalid values in messages, delays, and physical faults. For carrying out robustness testing using this framework, a tester must define a fault model (i.e., the types of faults to inject) a fault library (i.e., the implementation of the faults) and a fault script to support the fault injection process. The authors tested two communication subsystems of the NanoSatC-BR2 nanosatellite, a real world nanosatellite used in scientific mission programs by the Brazilian National Institute for Space Research.

In summary, existing works on the robustness of embedded systems cover a quite wide array of different types of systems, including aerospace software [5, 22, 24], autonomous systems (typically used in robot controllers) [23, 127, 128], real-time software components [110], and real-time operating systems [117, 118, 120, 124], just to name a few. Aside from the traditional fault injection approaches [116, 117, 121], many authors gave preference to model-based techniques for generating test cases [106, 109, 123], and one work used a particularly uncommon testing technique named abstract interpretation [105].

These techniques were used both at a high abstraction level (e.g., system interfaces) [119, 121, 123, 125] and at a quite low level (e.g., processor registers) [118, 120]. The faults used also present a wide range of diversity, and encompass the typical interface parameter mutations (e.g., invalid or boundary values [5, 22, 23, 106]), bit-flips on processor registers [118, 120], message-level faults (e.g., reordering messages [23, 127]), or even timing-related faults [24, 107, 109, 110]. The latter fault type, in particular, is quite common in embedded system robustness testing, which is expected given the large focus on real-time systems.

3.2.4 Middleware

This subsection discusses robustness evaluation of middleware. The term middleware refers to software that supports the operation of a certain software system and, at the same time, works on top of other software [61], which is a broad definition in which we include classic middleware, such as Java Message Service (JMS) implementations, as well as software systems that provide facilities or resources for supporting higher-level services, e.g., cloud management systems like OpenStack or Docker. Table 3.6 presents a summary of the main types of systems, testing techniques and their targets, the types of faults and the classification schemes used to evaluate the robustness of middleware.

Figure 3.4 represents an example of robustness evaluation of the (message-oriented) middleware supporting a distributed system, and is inspired in the approach presented in [6].

Table 3.6: Techniques for evaluating the robustness of middleware

Systems	Cloud platforms	Cardoso and Martins [129], Cardoso et al. [27], Chauvel et al. [130], Cotroneo et al. [131]
	CORBA	Pan et al. [132]
	HA middleware	Azevedo et al. [26], Fernsler and Koopman [133], Kövi and Micskei [134], Micskei et al. [25, 135]
	Message-oriented middleware	Laranjeiro et al. [6], Napolitano et al. [136]
Techniques	Code changes injection	Laranjeiro et al. [6]
	Fault injection	Azevedo et al. [26], Cardoso et al. [27], Chauvel et al. [130], Cotroneo et al. [131], Fernsler and Koopman [133], Kövi and Micskei [134], Laranjeiro et al. [6], Micskei et al. [25, 135], Napolitano et al. [136], Pan et al. [132]
	Interception	Kövi and Micskei [134], Micskei et al. [135]
	Model-based testing	Cardoso and Martins [129], Cardoso et al. [27], Cotroneo et al. [131], Micskei et al. [25]
Targets	API calls	Azevedo et al. [26], Cardoso and Martins [129], Cotroneo et al. [131], Fernsler and Koopman [133], Kövi and Micskei [134], Micskei et al. [25, 135], Napolitano et al. [136], Pan et al. [132]
	Cloud network components	Chauvel et al. [130]
	JMS Message fields	Laranjeiro et al. [6]
	Messages	Cardoso et al. [27]
Faults	Boundary inputs	Azevedo et al. [26], Cardoso et al. [27], Cotroneo et al. [131], *Fernsler and Koopman [133], Laranjeiro et al. [6]
	Change cloud topology	*Chauvel et al. [130]
	Inopportune inputs	*Cardoso and Martins [129]
	Invalid inputs	Azevedo et al. [26], Cardoso and Martins [129], Cardoso et al. [27], Cotroneo et al. [131], *Fernsler and Koopman [133], Laranjeiro et al. [6], Micskei et al. [25], Napolitano et al. [136], Pan et al. [132]
	Invalid return values	Kövi and Micskei [134], *Micskei et al. [135]
	MACD	Kövi and Micskei [134], *Micskei et al. [135]
	Shut down components	*Chauvel et al. [130]
Classification	7 categories	Fernsler and Koopman [133]
	6 categories	Pan et al. [132]
	5 categories	Azevedo et al. [26], Cardoso and Martins [129], Laranjeiro et al. [6], Napolitano et al. [136]
	Binary	Cardoso et al. [27], Chauvel et al. [130], Cotroneo et al. [131], Kövi and Micskei [134], Micskei et al. [25, 135]

As we can see in this particular case, the message sender middleware is instrumented so that faults are injected into messages that are about to be sent to a provider (and fetched later from the provider, whenever the receiver is online). The idea is that a faulty (or even malicious) client may send invalid messages and the provider middleware should be able to handle such cases. If an invalid message arrives, the provider should detect it and somehow reject the message. Even if the message is silently stored at the provider side, the receiving peer middleware should be robust to this invalid message.

In 1999, Fernsler and Koopman use the Ballista methodology to test the robustness of the High Level Architecture (HLA) Run-Time Infrastructure (RTI), a standard distributed simulation backplane that is expected to provide completely robust exception handling [133]. The only required setup for adapting Ballista to the target system (i.e., RTI) was to create object-oriented representations of the data types it uses (e.g., the *AttributeHandleSet* data type). Then, for each identified parameter type, a set of valid, limit (i.e., boundary) and invalid values are defined (e.g., zero or *MaxInt* for the integer type). Test outcomes are classified according to an extension of the CRASH scale, comprising of *pass*, *pass with exception* (i.e., RTI throws an exception stating that it has gracefully caught and handled a fault), *RTI internal error*, *unknown exception* (i.e., similar to *RTI internal error* but with an unknown exception), and finally *abort*, *restart* and *catastrophic* which are borrowed

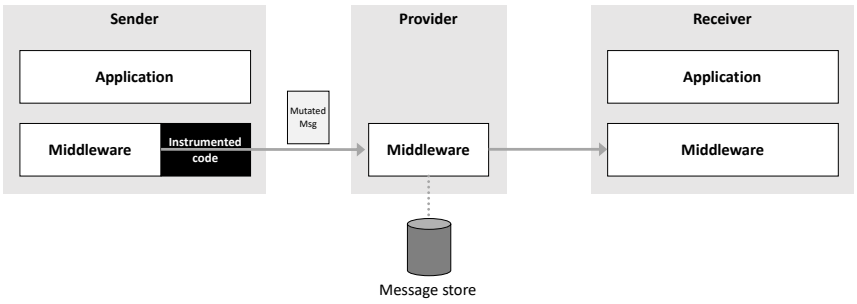


Figure 3.4: Robustness evaluation of the middleware supporting a distributed system, inspired by [6].

directly from the CRASH scale [3]. The authors evaluated the approach on three versions of HLA RTI, which displayed a failure rate of 6% for one version and 10% for the other two, including aborts, unknown exceptions and RTI-specific exceptions.

Pan et al. [132] assess the robustness of Common Object Request Broker Architecture (CORBA) client implementations. CORBA is mostly a way for integrating heterogeneous systems that is based on the use of RPC. In practice, the authors have adapted the Ballista technique [80] to the CORBA context by using the Object Request Broker (ORB) interface provided by these systems. The authors analyzed C++ ORB implementations to disclose several robustness problems. Behavior was categorized in robust (successful return or CORBA exception raised) and non-robust using the following adaptation of the CRASH scale: Computer crash (Catastrophic); Robust behaviors Thread hang (Restart); Thread abort (Abort); Raise unknown exception; False success (Silent); Misleading error information (Hindering). The authors analyzed three C++ ORB implementations for the Solaris and Linux operating systems, disclosing significant robustness vulnerabilities.

Micskei et al. analyzed, in 2006, different methods to generate input for testing the robustness of High Availability (HA) middleware compliant with the Application Interface Specification (AIS) [25]. These are: generic input testing - a set of 32-bit values (0, -1, 1, random, address of a variable), which can be cast to any primitive data type of the C programming language; type-specific testing - values are randomly generated depending on the data type of the parameter; and scenario-based - follows a model-based approach to consider the state of the target system, and generates values that are out of context for the current system state. The authors evaluated the three methods on two modules of OpenAIS, by injecting faults directly on the parameters of calls to the AIS of this middleware. In a later work [135], the authors extended the previous approach to include two new testing methods: mutation-based sequential testing - omits, swaps or modifies valid calls (i.e., Move, Add, Change, Delete (MACD) operators) to the AIS of the middleware under test; and the operating system call wrapper - intercepts systems calls made by the middleware and injects exceptional values into the returned objects. The authors tested this new approach on two real systems that use HA middleware. Kövi and Micskei [134] further evaluate the previous approach on a system with the open-source OpenSAF middleware. The results showed the effectiveness of the proposed test data generation methods, and revealed an increase in the robustness of the tested HA middleware throughout its development.

An experimental approach for assessing the robustness of JMS middleware is proposed by Laranjeiro et al. [6]. It is based on mutating messages immediately before they are sent by a producer/publisher. Mutations of JMS message fields are based on boundary and

exceptional values (e.g., null or empty values, maximum or minimum value of a data type), and are implemented by instrumenting the code (using aspect-oriented programming) of the Java methods used to send messages. Test execution is composed of three phases: i) observing normal behavior by sending valid JMS messages; ii) exchanging mutated messages between client and provider; and iii) sending valid messages again and attempt to disclose problems caused by the previous phase. Failures are classified with a CRASH [80] scale adaptation, and compliance with the JMS specification is also checked. A few cases of robustness problems were disclosed, including severe DoS vulnerabilities.

Napolitano et al. [136] have presented a tool for performing robustness tests on Data Distribution Service (DDS) middleware, which is an Object Management Group (OMG) standard for supporting Quality of Service (QoS) properties in publish-subscribe services. The proposed tool is capable of automatically injecting exceptional or invalid inputs through the AIS of a given DDS middleware implementation (OpenSplice, in the case of this work). Robustness issues observed during testing are manually categorized according to the CRASH scale. Results revealed a number of Restart and Silent failures, the latter also causing performance issues as a side-effect.

In [26], Azevedo et al. evaluate the robustness and scalability of two open-source HLA libraries (i.e., poRTIco and CERTI), which are used for simulating satellites. HLA is a software architecture for building simulators based on computational simulation components. Using the Ballista approach [80], invalid and boundary values are injected into the input parameters of the AIS calls of the libraries. These include null or empty values, non-printable characters, values that may cause data type overflow, among others. The authors used the CRASH scale to categorize the results of tests on poRTIco and CERTI, and observed a large number of robustness problems on both libraries, including Hindering, Silent and most often Abort failures, with also a few Catastrophic failures having occurred on CERTI.

Cotroneo et al. propose a method for testing the robustness of software running in Infrastructure as a Service (IaaS) clouds [131]. Test cases are generated in four steps: i) decomposition, where the set of testable functionalities is identified; ii) input modelling, for each input parameter of each functionality, valid, boundary valid and invalid input values are generated; iii) state modelling, wherein the state of the software under test is identified and modelled; and iv) application of constraints, for eliminating useless or redundant test cases. The authors applied their approach to the Apache Virtual Computing Lab (VCL), an open source cloud platform, and compared the results from three different perspectives: i) considering only inputs, which triggered only a few failures; ii) considering stressful environmental conditions (e.g., emulating concurrency from many parallel user requests), which was more successful in exposing failures; and iii) forcing the state of the system under test, which triggered specific faults which are entirely dependent on system state (i.e., and would not have been triggered by the other methods).

Chauvel et al. [130] evaluate the robustness of cloud-based systems by resorting to models used in the field of Biology for quantifying ecosystem robustness. The approach is based on an analogy between the propagation of extinctions of biological species and the propagation of failures between components of a software system. The approach extracts three key indicators from the cloud system under test: i) a robustness indicator (i.e., a value between 0 and 1) reflecting the impact that failures in the system have on other systems; ii) the most sensitive components in the system, whose failure brings down most of the functionality in the cloud topology; and iii) the most threatening failure sequences, which describe the most likely ordering of failures with a strong impact. These are extracted through testing, where each component of the cloud system under test is injected with faults, whose propagation

is monitored to identify failure sequences. Two fault types are considered: i) shutting down the components of interest; and ii) changing the configuration of the cloud network so that calls do not reach their expected endpoints.

Cardoso and Martins [129] evaluate the robustness of cloud platforms (e.g., OpenStack) using a combination of both search-based and model-based testing. The first step of the approach is *system decomposition*, wherein testers analyze the cloud platform under test by breaking down system functionality into individual testable units. Functionality of interest is then modelled in the form of Unified Modeling Language (UML) class and state diagrams. The authors apply both invalid and inopportune input values (i.e., unexpected but valid in the global context of the system) for evaluating robustness. The approach uses the Multi-Objective Search-based Testing algorithm for test case generation, which helps avoiding state explosion and reduces the amount of generated tests. The OpenStack cloud platform was used for the evaluation of the approach, and results were classified using an adaptation of the CRASH scale [3]. The experiment failed to disclose new robustness issues on OpenStack, but helped show that the platform is well protected against inopportune inputs.

The robustness of OpenStack is evaluated in [27], where the authors combine classic robustness testing techniques with model-based testing. The main idea is to set the system in different valid states (that are previously modeled) and apply robustness tests at the server side (at OpenStack Nova's Compute component, in this case). Behavior is categorized using information about the system states (e.g., if a faulty request places the system in an incorrect state) and the reported exceptions. The authors disclosed various cases of failures that have been reported to the OpenStack community.

In summary, middleware robustness assessment is mostly composed of three main cases of software being tested. Mainstream middleware like CORBA [132], JMS [6], or even DDS [136] is one of the targets. The second case concerns less popular middleware like HA middleware [25, 26, 134, 135] or HLA [26]. The third case concerns software that is designed to support other systems [129, 130], in particular cloud management platforms like the Apache VCL [131] or OpenStack [27].

Most of the studies use traditional robustness testing methodologies with adaptations to fit specific contexts (e.g., the works in [134, 135] apply a less usual fault model that includes swapping calls), however we do find a few applying model-based approaches where the behavior or a specification of the system is modelled and then used to generate test cases [25, 129]. The work by Cardoso and Martins goes one step further and applies a search-based test case generation technique [129].

A common case is the injection of faults in the parameters of AIS calls [133, 136] (e.g., boundary or invalid inputs, inopportune inputs - those used at the wrong system state). Some approaches ignore the parameters and just target the calls themselves by omitting or swapping them [134, 135]. A few studies also injected faults in the return values of AIS calls, replacing them with invalid values [134, 135]. More recently, we find approaches that explore unusual types of faults (i.e., when compared to the remaining middleware research), namely in [130] where the authors shut down components of cloud networks or change the network topology during execution.

3.2.5 Software components

This subsection mostly targets COTS software components and applications and, in general, software that is built to be reusable (e.g., software libraries). The increasing use of

COTS in dependable systems makes robustness of special importance, especially in cases like mission-critical systems. Robustness testing of software components has remained an active area of research since the early nineties, and existing studies are quite diverse as summarized in Table 3.7.

Costa and Madeira [156] assessed, in 1999, the behavior of a COTS Database Management System (DBMS), i.e., Oracle server, in the presence of faults injected using XceptionNT, a Windows NT/Premium port of the Xception dependability assessment tool [126]. Xception is a fault injection tool for emulating physical faults by performing low-level bit-flips. The authors injected faults in the processes and threads related to the DBMS (i.e., Oracle server) and only targeted processor registers with single bit-flips in randomly chosen bits. Test results were classified with one of *correct*, *abort* or *hang*, and more 10% of faults injected produced either an abort or a hang in the DBMS process. Costa et al. [157] extended this approach to also support higher-level software faults (i.e., rather than just processor register-level faults, as in [156]). The authors consider four types of software faults, namely assign (missing/wrong initialization), check (missing variable check, off-by-one or negation), overlay (repetition of instructions to store strings) and pointer (corruption of target or source address). Both assign and pointer fault types hold a predetermined value distribution, with 70% of generated faults relating to boundary values, and the remaining 30% equally distributed among changing the value to zero, to one, or to a random value. The authors evaluate the approach on a more recent version of the Oracle DBMS, and maintain the previous 3-level result classification scale.

Ghosh and Schmid [7] present an approach for assessing the robustness of COTS applications against faults in external components (e.g., faulty operating system functions), requiring only access to the interface of the application under test. Figure 3.5 shows a graphical representation of this method.

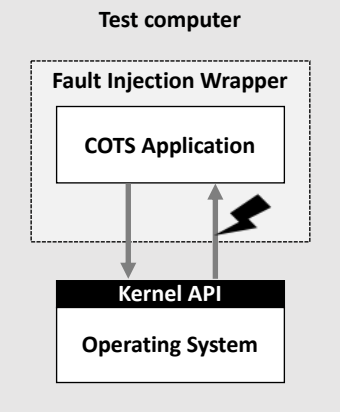


Figure 3.5: Robustness evaluation of a COTS application with emulation of a faulty Operating System [7].

The approach works by implementing a wrapper between the application interface and, for example the operating system (i.e., in case that is the component to test the target application with), which is then used to emulate faulty behavior (e.g., memory errors, I/O issues). When the target application performs calls to the external system, the wrapper either replaces call parameters with invalid values which are known to cause exceptions, or it directly replaces return values (from the original, valid call) with exceptions or error codes. The proposed approach has been implemented by the authors in the form of a Windows NT tool named the Failure Simulation Tool (FST).

Table 3.7: Techniques for evaluating the robustness of software components

Systems	COTS applications	Ahmad et al. [137], Belli et al. [138], Fetzer and Xiao [139], Ghosh and Schmid [7], Giuffrida et al. [140], Jin et al. [141], Quing-He et al. [142], Shahrokni and Feldt [143], Ufuktepe and Tuglular [144], Zamli et al. [145]
	COTS components	Csallner and Smaragdakis [146], Durães and Madeira [147, 148, 149, 150], Fernandez et al. [151], Lei et al. [152, 153], Moraes et al. [154], Oláh and István [155]
	COTS DBMS	Costa and Madeira [156], Costa et al. [157]
	GUI applications	Bauersfeld and Vos [158]
	Stateful components	Heckeler et al. [159]
Techniques	Code changes injection	Ahmad et al. [137], Costa et al. [157], Durães and Madeira [147, 148, 149, 150], Giuffrida et al. [140], Jin et al. [141], Moraes et al. [154], Oláh and István [155], Ufuktepe and Tuglular [144]
	Fault injection	Ahmad et al. [137], Bauersfeld and Vos [158], Costa and Madeira [156], Costa et al. [157], Csallner and Smaragdakis [146], Durães and Madeira [147, 148, 149, 150], Fetzer and Xiao [139], Ghosh and Schmid [7], Giuffrida et al. [140], Jin et al. [141], Lei et al. [152, 153], Moraes et al. [154], Oláh and István [155], Quing-He et al. [142], Shahrokni and Feldt [143], Zamli et al. [145]
	Model-based analysis	Ufuktepe and Tuglular [144]
	Model-based testing	Belli et al. [138], Fernandez et al. [151], Heckeler et al. [159], Lei et al. [152, 153], Oláh and István [155]
	Mutation testing	Belli et al. [138], Fernandez et al. [151]
	Static code analysis	Ufuktepe and Tuglular [144]
Targets	API calls	Fetzer and Xiao [139], Ghosh and Schmid [7], Lei et al. [152, 153], Moraes et al. [154], Shahrokni and Feldt [143]
	Application address space	Quing-He et al. [142]
	CPU registers	Ahmad et al. [137], Costa and Madeira [156], Costa et al. [157], Quing-He et al. [142]
	Function calls	Csallner and Smaragdakis [146], Oláh and István [155], Zamli et al. [145]
	GUI elements	Bauersfeld and Vos [158]
	Machine code	Ahmad et al. [137], Costa et al. [157], Durães and Madeira [147, 148, 149, 150], Jin et al. [141], Moraes et al. [154], Oláh and István [155]
	Source code	Giuffrida et al. [140], Ufuktepe and Tuglular [144]
	Specification	Heckeler et al. [159]
	System model	Belli et al. [138], Fernandez et al. [151]
Faults	Bit-level faults	Ahmad et al. [137], *Costa and Madeira [156], Costa et al. [157], Quing-He et al. [142]
	Boundary inputs	*Costa et al. [157], Moraes et al. [154], Zamli et al. [145]
	Communication failures	*Fernandez et al. [151]
	Invalid inputs	Costa et al. [157], Fetzer and Xiao [139], *Ghosh and Schmid [7], Giuffrida et al. [140], Heckeler et al. [159], Lei et al. [152, 153], Moraes et al. [154], Oláh and István [155], Ufuktepe and Tuglular [144], Zamli et al. [145]
	Invalid outputs	*Ghosh and Schmid [7], Oláh and István [155]
	MACD	Ahmad et al. [137], Belli et al. [138], *Oláh and István [155]
	Programming errors	*Durães and Madeira [147, 148, 149, 150], Jin et al. [141], Moraes et al. [154]
	Random inputs	Bauersfeld and Vos [158], *Costa et al. [157], Csallner and Smaragdakis [146]
	Timing faults	*Shahrokni and Feldt [143]
Classification	9 categories	Lei et al. [152, 153]
	5 categories	Shahrokni and Feldt [143]
	4 categories	Ahmad et al. [137], Durães and Madeira [147, 148, 149, 150], Moraes et al. [154]
	3 categories	Costa and Madeira [156], Costa et al. [157]
	Binary	Bauersfeld and Vos [158], Belli et al. [138], Csallner and Smaragdakis [146], Fernandez et al. [151], Fetzer and Xiao [139], Ghosh and Schmid [7], Giuffrida et al. [140], Heckeler et al. [159], Jin et al. [141], Oláh and István [155], Quing-He et al. [142], Ufuktepe and Tuglular [144], Zamli et al. [145]

In 2002, Durães and Madeira propose Generic Software Fault Injection Technique (G-SWFIT), a method for emulating software faults [147]. G-SWFIT analyzes the machine code of the target system, finds specific programming structures, and injects faults which emulate high-level software issues (e.g., programming errors). The faults considered by the authors include, for instance, missing or wrong variable initialization, replacing operators in conditional instructions, or missing function calls. The approach was evaluated on three benchmark applications (e.g., file compression utility), and the authors characterized results with one of *correct*, *error*, *erratic* or *timeout*, depending on the behavior displayed by the tested applications. G-SWFIT has been extended in [148], where the authors analyzed the most common types of software defects (i.e., software bugs) introduced by developers and extended the fault set used by G-SWFIT. In [149] the authors evaluated the suitability of using the G-SWFIT methodology in dependability benchmarks, focusing on essential properties such as repeatability, portability, and scalability. Finally, in [150] the authors formalize the concepts previously presented in [147, 148, 149]. Moraes et al. compare G-SWFIT to the classical robustness testing method of injecting faults at the component interface [154]. The authors used Jaca [160] and Xception [126] to inject interface-level faults, and used boundary values (e.g., maximum integer), and values with well associated meanings (e.g., NULL; 0, -1). The experiment was carried out on the Ozone object-oriented Java database, and on a real-time space application implemented in C. Results show quite different behavior between both fault types, and the authors conclude that interface faults do not represent well residual faults in component code.

Csallner and Smaragdakis present JCrasher [146], a tool for automatic robustness testing of code written in Java. JCrasher analyzes Java classes and generates test cases that exercise their methods by calling them with randomly generated input. Because JCrasher requires access to source code, it is a white-box approach. Two disadvantages of this tool are that it is only capable of testing Java code and that it only allows randomly generated input (e.g., users would benefit from being able to define custom domains for parameters). The authors evaluated JCrasher on a set of eight Java classes, the majority of which often crashed during testing.

Fetzer and Xiao [139] introduce HEALERS, a tool for automated black-box robustness testing of C/C++ component-based systems by detecting programming errors and security vulnerabilities. The approach is based on detecting potential robustness violations by computing robust argument types. These restrict the set of possible values in such a way that no robustness issue may occur in the function. For instance, a C/C++ function taking a *char** (i.e., pointer to a character array) is expected to also take an integer argument representing its length. The robust argument type of these arguments should reflect that the first argument is an array of length equal to the value of the second argument. Robust argument types are determined through fault injection experiments, wherein each target function is called with both valid and exceptional values. These experiments are performed in an adaptive manner, in the sense that the range of generated input values is iteratively adapted to the set of values which allows the function to execute properly (i.e., values not in the final set are those that trigger robustness issues).

In [151], Fernandez et al. present a model-based framework for robustness testing of software systems, which is applied to a Controller component of a ticket machine. The approach requires the existence of a specification of the software component under test, from which it produces a mutated version. A set of robustness requirements are defined by the tester, which are used with model-checking against the generated mutated specification. This process results in what the authors call *diagnostic sequences* - abstract models from which test cases can be generated. This approach tests software from a black-box perspective, but it still requires that implementation specifications with functionality details be

provided, and is often not the case in this type of testing.

Zamli et al. [145] introduce SFIT, an automated software fault injection tool for testing the robustness of Java COTS systems. The tool relies on the Java Reflection API to obtain the set of existent functions for each Java class in the system under test, and thus does not need direct access to source code (only to the compiled Java bytecode). This allows for automated discovery of method interfaces over any Java class. Test cases are defined in the form of *fault setting* files, which contain information regarding fault injection locations, how faults are to be injected, which faults to inject and how many. Unfortunately, the tool does require a large amount of manual work in the case of testing complex systems with many Java classes (which is often the reality), because testers must manually define test cases in *fault setting* files.

Jin et al. introduce the PIN-based Dynamic Software Fault Injection System (PDSFIS), a fault injection method based on the PIN framework provided by Intel [141]. PIN is a tool that allows users to scan specific Assembly code instructions in an executable file and inject code at those positions. PDSFIS uses this to scan for specific code patterns (e.g., *mov* instructions) and inject faults that represent high-level programming errors (e.g., missing variable initialization). The authors evaluated PDSFIS experimentally on the Apache web server, and observed many cases of erroneous behavior including delayed responses, uncaught exceptions and lack of connection.

Oláh and István present a model-based tool for performing robustness testing, through fault injection, on software components written in Java [155]. The tool reads the Java bytecode from the component under test, and presents the tester (through a graphical interface) with a UML-based model of the internal structure of the component. The tester then configures the faults to inject (including their locations, time or rate of activation, and parameters) and the monitoring aspects (e.g., parameter monitoring, return value monitoring, method call monitoring) of the test suite to carry out. The tool supports definition of custom faults, but by default the following faults are provided: omit return statement; omit method call; return NULL; return predefined number (i.e., set by the tester); return predefined string (i.e., set by the tester); and use predefined call parameters (i.e., set by the tester). The authors did not carry out any experimental evaluation of the presented tool.

Lei et al. [153] present a stateful robustness testing approach for reusable software components. The approach begins by generating a state machine representation of the component under test using the refinement of Object and Component Systems (rCOS) model. From this model robustness test cases are automatically generated, containing randomly generated valid as well as invalid inputs. The authors classify disclosed robustness faults in one of nine different categories, based on a set of predefined rules. These depend on the state before and after test execution and whether or not exceptions were raised during execution. Taking internal component state into account increases the coverage of the robustness tests. A drawback of this approach is the requirement of the rCOS model, which is not exactly a widely used software component description format and may, in this sense, limit the applicability of this approach. In another paper of the same year [152], the authors describe the same approach again, with negligible changes.

Belli et al. [138] describe a model-based approach for testing the robustness of software systems using Event Sequence Graphs and Decision Tables. Event Sequence Graphs are used to represent system behavior and the interactions between the user and the system as events, while Decision Tables model user inputs and control flow (e.g., different user inputs lead to different event flows). Test cases are represented by sequences of events and are mutated through one of three mutation operators: i) insertion of an element; ii)

omission of an element; and iii) combination of two elements. Mutated event sequences (i.e., test cases) are meant to represent erroneous or undesirable system behavior. This approach requires that Event Sequence Graphs be manually designed by testers, which is a time consuming activity. Experimental evaluation was carried out on Isik's System for Enterprise-Level Web-Centric Tourist Applications (ISELTA), resulting in the detection of 18 robustness faults.

Quing-He et al. [142] describe Memory-Oriented Fault Injector (MOFI), a Windows NT tool for testing the robustness of applications through the injection of software faults that emulate the effects of radiation-induced hardware faults. MOFI injects faults into processor registers and memory locations of a running application, and only requires the PID (i.e., process ID) of the target application to begin testing. Upon identifying the target registers and memory locations (i.e., the tool only considers the memory space allocated to the application under test), MOFI injects bit-flip faults, which can occur in single as well as multiple bits.

Shahrokni and Feldt [143] present RobusTest, a Java-implemented framework for testing the robustness of software systems, with focus on timing issues. The user first manually identifies testable properties of the target system (i.e., what the system should or should not do, under defined conditions) as well as its expected behavior (i.e., test oracle), and RobusTest then uses this information to automatically generate test cases. A test is composed of a set of randomly generated time delays between the calls to the system under test. The authors integrated the CRASH robustness scale directly into the verdict component of the RobusTest framework, which essentially automates the process of result classification. The proposed framework was evaluated on two open source implementations of the Extensible Messaging and Presence Protocol (XMPP) instant messaging protocol, namely Vysper 0.7 and Ejabbered 2.1.8, where it was able to disclose several robustness problems, including a few Catastrophic and Restart failures.

In [158], Bauersfeld and Vos describe GUITest, a Java library for automated robustness testing of GUIs. GUITest detects control elements of a graphical interface, along with their corresponding properties (e.g., size, position, content or type). GUITest also takes into account interface element hierarchies, which are structured through *widget trees*. Testing works by emulating user actions over GUI elements (e.g., mouse clicks or drags, keystrokes from the keyboard), which in turn activate control elements (e.g., hyperlinks, sliders) and, if possible, change their content (e.g., tapping a text field and editing its text). The tool is capable of generating random input sequences which target the detected interface elements. Robustness issue detection works by checking if the generated input test sequences make the application crash or freeze (i.e., halt its execution). The authors tested GUITest on Microsoft Word for Mac, and a few test cases were able to effectively crash the program.

Heckeler et al. [159] describe an approach to test stateful components written in C++. It requires a UML Statechart specification of the component under test, which describes defined and undefined state transitions, respectively representing its normal and abnormal working conditions. The Z3 constraint solver is used to generate robustness tests from the UML Statechart (essentially, a state machine model) based on transition coverage criteria. When a test is finished the component state is driven back to the starting state required by the next test, rather than the entry state of the component itself. The goal of this is to reduce the number of states to execute over a test suite, thus discarding irrelevant state transitions and reducing overall testing time. An evaluation of the approach was carried out against three C++ components, in which a few faults were detected.

EDFI is a general-purpose fault injection tool, which combines static and dynamic program instrumentation to perform execution-driven fault injection [140]. The approach is based

on generating multiple heterogeneous faulty and fault-free versions of the target application at compile time (static instrumentation), and seamlessly interleave them in a controlled way during execution (dynamic instrumentation). The proposed fault model supports time-based as well as probability-based fault injection, and the user of EDFI is free to choose which faults will inject (e.g., bit-flips, data type-based mutations). The authors evaluated EDFI on an instance of the MySQL database management system and on the Apache httpd web server and were able to detect a few faults.

Ufuktepe and Tuglular [144] propose a method for estimating the robustness of software applications in relation to input validation vulnerabilities through Bayesian Networks. The approach searches the source code of the application under test for input validation code (e.g., *if* statements), and checks whether this code is exploitable. The authors focus on a set of six known input validation vulnerabilities, namely: i) path transversal (CWE-22); ii) OS command injection (CWE-78); iii) cross site scripting (CWE-79); iv) SQL injection (CWE-89); v) buffer overflow (CWE-120); and vi) uncontrolled format string (CWE-134). The probabilities of finding these six vulnerabilities in a software application (based on 15 years of data) are preloaded into a Bayesian Network. To test if these vulnerabilities exist in the target application, the source code is statically analyzed with a tool that checks, for each function, if its input parameters are validated before being used. A metric called the validation ratio (i.e., a number between 0 and 1) is obtained by counting the number of validated vulnerabilities out of the applicable vulnerabilities. The Bayesian Network then uses this value to estimate the robustness of the target function.

Ahmad et al. [137] introduce Lightweight Dynamic Software-based Fault Injection (LDSFI), a technique for injecting faults into the binary code of software applications at runtime. The approach uses the GNU Debugger to periodically interrupt execution of the target program through traps, and inject faults. The fault model consists of bit-flips in the program counter register, random data registers, random instructions or branch instructions, and removing branch instructions or replacing them with non-branch instructions. The authors evaluate the LDSFI method on three benchmark programs (i.e., bubble sort, quicksort and matrix multiplication), and categorized the results with a four-level scale: *operating system* - the operating system signals a fault through an exception; *timeout* - the program consumes more time than expected (e.g., infinite loop); *correct result* - program behaves as expected; and *silent data corruption* - result produced by the program is incorrect. Experimental results show that LDSFI is a highly portable, non-intrusive technique for performing instruction-level fault injection without requiring source code.

In summary, research on COTS robustness testing covers a large time period, ranging from the late nineties [7] to very recent years [137]. The rapid increase in the quantity and diversity of approaches resulted in the appearance of less usual robustness evaluation techniques, such as the use of mathematical constructs to model certain system properties [151, 152], or the use of Bayesian Networks for estimating system robustness [144]. Fault injection, however, still remained as a quite common technique for assessing robustness in this context. It has been applied in the values of call parameters [143, 145, 146], in source code [140] as well as machine code (i.e., compiled) [141, 147], or even in the processor registers [137, 142, 156, 157].

The types of faults applied are relatively usual, encompassing invalid, random or boundary input values [7, 139, 146, 152], bit-related faults (e.g., bit-flip, bit-masking) [156] and also invalid return values [7]. Some studies explore solutions which are conceptually different from traditional bit or parameter-based tampering, such as removing machine code instructions [137] or even injecting machine code that emulates high-level programming mistakes [141, 147].

3.2.6 Web services

This subsection discusses robustness evaluation of web services (in *latu sensu*) and mostly covers: i) web applications (i.e., client-server applications that communicate using the HTTP protocol, usually for transporting Hypertext Markup Language (HTML) documents and related objects) [161]; ii) SOAP web services, which have the goal of allowing interoperable communication between heterogeneous systems [162]; and iii) SOAP web service compositions, which are built based on the aggregation of individual composing units [163]. Table 3.8 summarizes the main characteristics of the research identified that targets web services.

Fu et al. [8] presented, in 2004, an approach for testing the robustness of Java-based web applications, which takes advantage of Java's exception handling mechanisms. The approach, depicted in Figure 3.6, is based on instrumenting the target application bytecode so that the execution path of a certain request is forced towards specific error handling code paths (i.e., *catch* statements in *try-catch* blocks). This allows developers to evaluate the error handling mechanisms and overall robustness of their applications. For instance, a non-robust application would throw an unexpected exception and abort execution, whereas a robust one should be able to comply with the specification and, for instance, proceed execution.

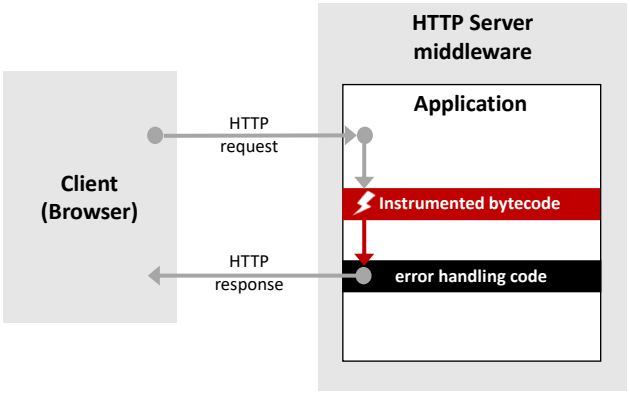


Figure 3.6: Robustness evaluation of a web application, by triggering error handling code [8].

SOAP web services rely on a set of eXtensible Markup Language (XML)-based specifications to support their operation [162]. Among these, it is worthwhile mentioning the SOAP protocol, which allows clients and servers to exchange messages in a platform independent way; and the Web Services Description Language (WSDL) which allows describing a service interface (e.g., operations, operation arguments, data types) in a platform independent manner [162].

Looker et al. [168] introduced, in 2004, the Web Service Fault Injection Technology (WS-FIT), further detailed in [169], a method and tool for testing the robustness of SOAP web services through fault injection. WS-FIT is based on the concepts first devised by Looker and Xu in [181], in which the authors described a method for assessing the dependability of SOAP web services through fault injection. WS-FIT requires instrumentation of the code of the target SOAP service to allow interception of both outgoing and incoming SOAP messages. Using the WSDL document of the service under test, WS-FIT maps intercepted SOAP messages to the service operations, their parameters and respective data types, and generates test cases. The approach considers multiple fault types, such as delay and

Table 3.8: Techniques for evaluating the robustness of web services

Systems	BPEL compositions	Ilieva et al. [31], Kuk and Kim [29]
	SOAP web services	Carrozza et al. [164], Hanna and Munro [165], Laranjeiro et al. [166, 167], Looker et al. [168, 169, 170], Martin et al. [171, 172], Rabhi [173], Rychlý and Žouželka [174], Salas et al. [175], Salva and Rabhi [30, 176], Siblini and Mansour [9], Vieira et al. [177, 178]
	Web applications	Calori et al. [179], Fu et al. [8], Mendes et al. [28], Pattabiraman and Zorn [180]
Techniques	Code changes injection	Fu et al. [8], Mendes et al. [28]
	Fault injection	Carrozza et al. [164], Hanna and Munro [165], Ilieva et al. [31], Kuk and Kim [29], Laranjeiro et al. [166, 167], Looker et al. [168, 169, 170], Martin et al. [171, 172], Mendes et al. [28], Pattabiraman and Zorn [180], Rychlý and Žouželka [174], Salas et al. [175], Salva and Rabhi [30, 176], Vieira et al. [177, 178]
	Interception	Looker et al. [168, 169, 170], Rychlý and Žouželka [174], Salas et al. [175]
	Model-based analysis	Calori et al. [179]
	Model-based testing	Rabhi [173], Salva and Rabhi [30]
	Mutation testing	Rabhi [173], Siblini and Mansour [9]
	Service emulation	Kuk and Kim [29]
Targets	Error handling code	Fu et al. [8]
	Machine code	Mendes et al. [28]
	Service calls	Pattabiraman and Zorn [180]
	SOAP message fields	Carrozza et al. [164], Hanna and Munro [165], Ilieva et al. [31], Kuk and Kim [29], Laranjeiro et al. [166, 167], Martin et al. [171, 172], Salva and Rabhi [176], Vieira et al. [177, 178]
	SOAP messages	Looker et al. [168, 169, 170], Rychlý and Žouželka [174], Salas et al. [175], Salva and Rabhi [176]
	Specification	Calori et al. [179]
	System model	Rabhi [173]
	WSDL fields	Salva and Rabhi [30], Siblini and Mansour [9]
	WSDL operations	Salva and Rabhi [30]
Faults	Boundary inputs	Carrozza et al. [164], Laranjeiro et al. [166, 167], *Siblini and Mansour [9], Vieira et al. [177, 178]
	Command injection	*Laranjeiro et al. [167], Rychlý and Žouželka [174], Salas et al. [175]
	Invalid inputs	Calori et al. [179], Carrozza et al. [164], *Fu et al. [8], Hanna and Munro [165], Ilieva et al. [31], Kuk and Kim [29], Laranjeiro et al. [166, 167], Martin et al. [171, 172], Pattabiraman and Zorn [180], Rabhi [173], Rychlý and Žouželka [174], Salva and Rabhi [30, 176], Siblini and Mansour [9], Vieira et al. [177, 178]
	Invalid operation name	*Salva and Rabhi [30]
	MACD	*Looker et al. [168, 169, 170], Salva and Rabhi [176], Siblini and Mansour [9]
	Programming errors	*Mendes et al. [28]
	Random inputs	Ilieva et al. [31], *Martin et al. [171, 172], Rabhi [173], Salva and Rabhi [176]
	Timing faults	*Looker et al. [168, 169, 170], Rychlý and Žouželka [174]
Classification	5 categories	Carrozza et al. [164], Laranjeiro et al. [166, 167], Mendes et al. [28], Vieira et al. [177, 178]
	3 categories, 5 subcategories	Ilieva et al. [31]
	Binary	Calori et al. [179], Fu et al. [8], Hanna and Munro [165], Kuk and Kim [29], Looker et al. [168, 169, 170], Martin et al. [171, 172], Pattabiraman and Zorn [180], Rabhi [173], Rychlý and Žouželka [174], Salas et al. [175], Salva and Rabhi [30, 176], Siblini and Mansour [9]

reordering of SOAP messages, and duplication, omission and addition of both messages and message parameters. In [170] the authors compared WS-FIT with code insertion, a well-established fault injection method, and concluded that WS-FIT holds comparable performance and fault injection capabilities and is less invasive.

The work by Siblini and Mansour [9] rely precisely on the WSDL documents associated with SOAP services and generate mutants, with the goal of generating clients that comply with each mutated specification. However, the approach sends requests to the genuine

service implementation, with the goal of disclosing robustness problems. This concept is represented in Figure 3.7 and described in further detail in the next paragraphs.

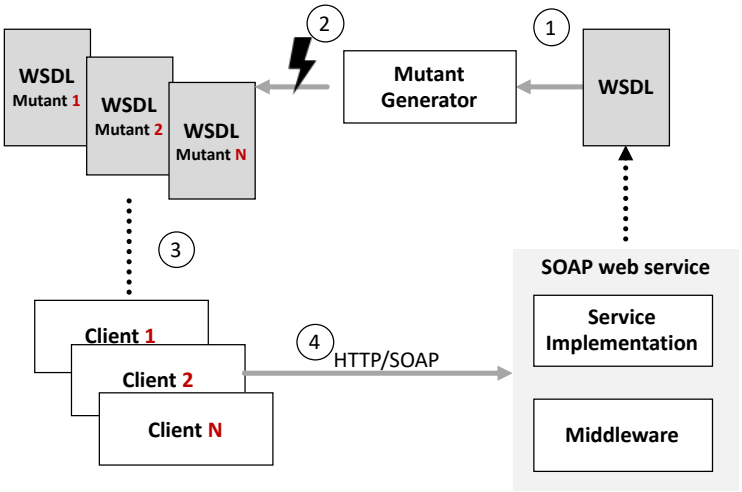


Figure 3.7: Approach for robustness evaluation of SOAP web services based on mutated WSDL documents [9].

The approach depicted in Figure 3.7 relies on a set of three WSDL-oriented mutation operators, which target three different XML elements in the WSDL documents, namely the *types*, *message* and *port type* elements. The mutation operators are *switch*, which replaces elements in pairs, the *special* operator modifies an element's value to a boundary value, the next value in the domain or null (if applicable for the data type), and *occurrence* which deletes or adds an occurrence of a given element. The authors evaluated the approach on an example credit card checker web service, which showcased its usefulness in revealing interface errors in WSDL documents and logical errors in web services.

In 2007, Vieira et al. proposed an approach for assessing the robustness of SOAP web services based on the combination of valid and invalid input values [177]. The approach was adapted in [178], where the authors aim at a method for evaluating and comparing web services robustness, which was later implemented in *wsrbench* [166] – a public web application for testing the robustness of SOAP web services. As a result of these endeavors, the concepts regarding robustness testing of SOAP services were matured and formalized in [167], in which a robustness testing approach, based on invalid and malicious inputs, is described in detail. The inputs used are mostly based on limit conditions, and are generated by applying a set of predefined rules that are data type-dependent. The authors propose a set of steps for testing the robustness of SOAP web services, which include extracting information from the WSDL file describing the service, generating and executing a workload, i.e., a set of valid calls to the service (for understanding the service normal behavior), defining and executing a faultload (i.e., a set invalid and malicious inputs that is integrated into the workload), and classifying the observed behavior. The behavior is classified using an adaptation of the CRASH scale [3] and a set of behavior tags proposed by the authors. The authors applied their approach to 250 online services, disclosing numerous robustness issues, including security vulnerabilities.

Martin et al. [171] define a framework for automated generation and execution of web service robustness tests based on the information extracted from the service's WSDL files. A set of Java classes is first generated based on the service interface, each class targeting a specific service operation. Afterwards, a test generation tool (e.g., JCrasher [146]) is used to automatically create robustness tests for the Java classes obtained in the previous step.

Finally, the Java testing tool JUnit is used to execute the robustness tests against the web service under test. The described framework was later implemented by the authors in the form of a tool named WebSob [172]. The authors evaluated WebSob on 35 public web services, and disclosed a considerable number of robustness problems with no knowledge regarding the service implementations.

Calori et al. [179] propose a framework for conducting robustness analysis of web applications at early development stages. The approach is comprised of four steps: i) define a severity-based ranking of failure scenarios (e.g., critical, not critical); ii) use Jacobson's analysis method (a combination of Failure Modes and Effects Analysis (FMEA), and Bayesian Belief Network (BBN)) to capture system behavior aspects at early development stages when little information about system structure is known, and to identify interface objects, entity objects (e.g., databases) and control objects (e.g., application logic); iii) apply FMEA (a technique used to examine potential failures in processes and identify corresponding causes and effects) for each identified control object; iv) represent the system in the form of a BBN model, with arcs representing the cause-effect relationships identified by FMEA and nodes representing identified variables. Finally, variables are assigned concrete states (i.e., values) and the Bayesian Network calculates an estimation of the severity of a robustness failure in the system, which allows to identify critical components in the system (which should be focal points for developers).

A fault injection approach for testing SOAP web services is described by Hanna and Munro in [165]. The proposed method automatically generates robustness test cases, but requires a previous (manual) analysis of the service's WSDL document. This is used to identify possible robustness faults in the service, allowing the tester to build a set of rules for automatic test case generation. A rule is composed, among other fields, by the target fault, the testing technique to use (e.g., boundary value analysis, robustness testing, mutation testing), the target WSDL parameter and the test data (e.g., value to inject in the target parameter). The authors implemented the approach in a tool named Web Services Testing Framework (WSTF), which was evaluated against a single-operation in-house web service running on top of Axis middleware [182] and the Tomcat web server [183], where the results only revealed a single minor robustness issue.

Kuk and Kim [29] detail a method for assessing the robustness of web service compositions based on Business Process Execution Language (BPEL). BPEL documents contain information regarding the service composition structure, the set of participating services and the exception handling characteristics. By analyzing the participant services list, it is possible to obtain the corresponding WSDL files. The approach does not invoke the actual services in the target composition, but rather generates virtual services which mimic the real ones and contain only the necessary logic to conform to the behavior specified in the respective WSDL files (e.g., return valid or invalid results, return exceptions described in the WSDL). Valid and exceptional test cases are thus automatically generated from this data. This approach depends greatly on the completeness of BPEL and WSDL specifications, which may often not be the reality. The proposed approach was evaluated on a pump design engineering application, uncovering a few issues related to uncaught exceptions being thrown due to parameter mishandling.

A 2009 work by Salva and Rabhi [176] discusses an automated robustness testing method for web services based on information gathered from their interface descriptions documents. The automatically generated tests comprise of random calls to the service operations using random and invalid values as input parameters. The authors also propose a method for reducing the number of tests by identifying and removing test cases that contain faults which are likely to be consumed by SOAP processing middleware, as these faults would

never actually affect the web service implementation. The same authors proposed an extension to this approach in [30], which focuses on stateful SOAP web services and works by modelling the service state as a Symbolic Transition System (STS). The test case generation method used relies on the addition, modification and replacement of web service operation names and values for operation parameters. The state of the service is actively modelled by an STS, by mapping each service state to an STS state and its transitions to the service operations. The resulting model should represent the overall web service behavior and state transitions, including calls to incorrectly named operations and also state transitions resulting from incorrect service responses. The set of invalid inputs considered (i.e., invalid operation names and unusual parameter values) is relatively small, when compared to most other robustness evaluation approaches.

Pattabiraman and Zorn describe an approach for assessing the robustness of Web 2.0 applications which relies on the identification of the invariant elements of the Document Object Model (DOM) in [180]. DoDOM is a tool that is able to automatically identify which of the elements in a service's DOM tree remain constant by recording the outputs from the service as a user performs a series of inputs (e.g., navigates through the web application). Upon obtaining a model of the invariant elements in the DOM tree, the service is tested with invalid inputs, and the resulting DOM trees are compared with the tree of the original model. Major deviations in structure constitute erroneous behavior, and thus represent robustness issues. This approach has the advantage of allowing service consumers (i.e., users) to easily apply it, as access to source code is not necessary. However, the approach assumes that the many views of a web application hold some similarity in structure between each other. In cases where this is not true, one can only expect that the baseline DOM tree contains root HTML elements such as *head* and *body* (i.e., because these would, hypothetically, be the only invariant elements in such a dynamic environment). Authors evaluated DoDOM on a set of three web applications, with results showing a high coverage of known faults.

Mendes et al. [28] assess the impact that web application faults have on web servers, by injecting faults in the server-side applications and monitoring the behavior of the web server from both server and client sides. The approach has two phases, a first one for establishing the baseline web server behavior and a second phase where fault injection is used. The workload (used in the first phase) is comprised of a set of representative operations which aim to exercise the web server under normal operation. The faultload (used in the second phase) is composed of faults that emulate typical high-level programming mistakes (e.g., missing function call, wrong assignment value, missing *if* statement), which are injected using the Generic Software Fault Injection Technique (G-SWFIT), presented in an earlier paper [150] (and previously described in Subsection 3.2.5). The authors use a qualitative failure mode scale classification to label each fault injection outcome. The approach was experimented on a realistic e-commerce web application running on three widely known web servers, and results show a small number of problems, such as calls returning wrong results and excessive use of system resources.

Carrozza et al. propose a tool for assessing the robustness of web services in [164], which is named WSRTesting. The main motivation is to handle wsrbench's limitations [166] and create a tool that complies with industry requirements. These requirements include the capability of recognizing complex data types in WSDL files, taking into account the semantics of specific operations within different web services (e.g., some operations require data from others, thus test cases should allow for proper operation sequencing), or even automatically managing test results. The proposed approach, WSRTesting, aims to fill in these gaps by presenting testers with interfaces to configure and properly setup the desired testing environments, which include the input of admissible testing ranges for each

operation parameter (or using the tool's default values when no ranges are specified), or even providing the tool with a description of XML-formatted custom parameter types.

A paper by Ilieva et al. [31] presents the TASSA framework for automated generation and execution of robustness test cases against BPEL web service compositions. The test cases generated by TASSA are based on fault injection using invalid, unexpected or random data. In order to test a web service composition, the TASSA framework requires a BPEL specification document. Robustness faults disclosed during testing are classified into one of several categories of three fault types, namely physical, interaction and development. The authors validated the TASSA robustness testing framework on a case study service composition comprised of three separate SOAP services, and around incorrect behavior was observed in around 38% of test cases.

Rabhi presents a model-based approach for testing robustness of web service compositions operations in [173]. First, an STS specification is generated, which models the behavior of the web service composition. An STS is an extended automaton model that associates behavior with a specification, and is composed of transitions labelled by actions as well as internal and external variables of a system. From the composition STS, a symbolic execution tree is automatically generated per service operation. Symbolic execution trees describe execution paths of each operation and serve as a per operation sub-specification. Finally, the set of symbolic execution trees (one per service composition operation) are mutated to represent incorrect (i.e., unexpected or invalid) behaviors.

Rychlý and Žouželka describe Fault Injector for Web Services (FIWS), a tool for testing the robustness of SOAP web services through fault injection experiments [174]. FIWS proxies between a SOAP web service and its client application, and intercepts traffic being sent to the service. Every intercepted message is dispatched to a fault injector component, which tests the contents of the message against a set of user-defined rules. A rule is composed of a set of conditions (e.g., based on message headers or payload), and a set of faults related to message content (e.g., string replacement, XPath corruption or multiplication, corruption of headers, removing payload) or the message itself (e.g., delay message transmission). When an intercepted message meets all the conditions of a rule, each of its faults is injected into it and each faulty message is then forwarded to the service under test. The authors evaluated the FIWS tool on a set of sample web service projects included in the Netbeans Integrated Development Environment (IDE), and on a set of four public web services, with some fault injections resulting in 400 Bad request and 500 Internal server error status code responses.

Salas et al. present WSInject, a fault injection tool for assessing the robustness of SOAP web services that implement the WS-Security extension [175]. WSInject works as a proxy between the client application and the target web service, and intercepts the SOAP messages sent from the client to the service. The faults injected into SOAP messages emulate different injection attacks, including XML code injection, cross-site scripting, and XPath injection. The existence of security vulnerabilities (strongly related with input-level robustness) is diagnosed according to eight rules, which rely on verifying the HTTP status code of the service response (to a faulty request) as well as whether or not private information was disclosed by the service. The proposed approach was evaluated on a set of ten web services, and many XML Injection-related vulnerabilities were disclosed.

In summary, we found research on robustness evaluation of web services targeting pure web applications [8, 28, 179, 180], individual SOAP services (e.g., [165, 167, 171, 173]) and service compositions (i.e., [29, 31]). The work found on web applications is relatively scarce, with some works applying less usual methods (e.g., code instrumentation [8] or Jacobson's analysis method [179], programming errors being introduced into the system

[28]).

Regarding the SOAP web services, it is worthwhile mentioning that the targets of the techniques now also aim at a key component of these services, the interface description [9, 30, 164, 177].

Some approaches rely on parameter-based tampering methods by directly mutating the input values in calls to service operations [30, 31, 176]. These mutations may be based on boundary, invalid or purely random values [9, 167, 171], or even values that exclusively aim to detect security vulnerabilities [175]. Others work by intercepting SOAP messages and delaying, reordering, removing or duplicating them [168, 174], or instead mutate the fields of the WSDL specification to replace, add or delete fields [9] or changing names of service operations [30] to evaluate how the service responds. It is worthwhile mentioning that we also observed a few cases of research focused on delivering a tool that could be effectively used by practitioners [31, 164, 166, 170].

As a final comment to the research carried out in SOAP web services, it is interesting to notice that the last work dates back to 2015, which signals the known decreasing interest in SOAP services. At the same time, we were not able to find any work targeting Representational State Transfer (REST) services, which posses different characteristics, namely less rigid service invocations, possible absence of interface descriptions, unstructured or semi-structured service descriptions [15], which may be relevant information for future research.

3.2.7 Autonomous and Adaptive systems

This subsection discusses research that targets the robustness evaluation of autonomous and adaptive systems, i.e., software systems that have to operate autonomously and may have capabilities of autonomous adaptation to changing environments [66]. A plain autonomous system may be a robotics system or the core software system in an autonomous vehicle, whereas an adaptive system is one whose configuration can be (self-) adjusted during execution, to maintain a certain level of QoS or to fulfill operational goals. These systems are usually based on a feedback loop mechanism (e.g., MAPE-K [184]), where the system constantly receives data regarding the sensed environment and in the end actuates to adjust itself, in a certain manner, to the changing environment. Table 3.9 summarizes the research identified in this topic.

In 2004, Bennani and Menascé [186] proposed an approach for assessing the robustness of self-managing software systems. The approach relies on using workloads with highly variable time requirements. That is, each workload requires that the system use a certain amount of computational resources that depends on how large or small its set of requests is. By using highly variable workloads, the authors intend to stress the underlying self-adjustment modules of these systems and attempt to disclose robustness problems. This is a black-box approach, but it requires feedback regarding the behavior of the system under test, which may be impossible in some situations (e.g., when the system under test is remote).

A work by Chu et al. [23] describe a fault injection-based approach to test the robustness of the LAAS architecture, a software architecture for real-time control of mobile robots. The approach works by injecting faults in the real robot controller software, which is itself attached to a robot simulation tool. Faults are injected in messages passed between the *functional* (i.e., the fault injection target) and *decisional* layers of the architecture. Three types of faults are considered: i) unforeseen messages (i.e., message flooding); ii) message

Table 3.9: Techniques for evaluating the robustness of autonomous and adaptive systems

Systems	Autonomous	Chu et al. [23], Hutchison et al. [128], Katz et al. [185], Powell et al. [127]
	Self-adaptive	Bennani and Menascé [186], Cámara et al. [10, 187, 188, 189]
Techniques	Fault injection	Cámara et al. [10, 187, 188, 189], Chu et al. [23], Hutchison et al. [128], Powell et al. [127]
	Interception	Hutchison et al. [128]
	Mutation testing	Katz et al. [185]
	Stress testing	Bennani and Menascé [186]
Targets	API calls	Bennani and Menascé [186], Katz et al. [185]
	Message fields	Chu et al. [23], Cámara et al. [10, 187, 188, 189], Hutchison et al. [128]
	Messages	Chu et al. [23], Hutchison et al. [128], Powell et al. [127]
Faults	Boundary inputs	*Cámara et al. [10, 187, 188, 189], Katz et al. [185]
	Invalid inputs	*Chu et al. [23], Cámara et al. [10, 187, 188, 189], Hutchison et al. [128], Katz et al. [185]
	MACD	*Chu et al. [23], Powell et al. [127]
	Stressload	*Bennani and Menascé [186]
	Timing faults	*Chu et al. [23]
Classification	5 categories	Cámara et al. [10, 187, 188, 189]
	4 categories, 12 subcategories	Chu et al. [23]
	Binary	Bennani and Menascé [186], Hutchison et al. [128], Katz et al. [185], Powell et al. [127]

transmission disruption, which includes removing, delaying or repeating messages, as well as inverting the order of message sequences; and iii) message parameter corruption similar to the one carried out in [3]. Each test outcome is classified considering the following four dimensions: *interface* (i.e., the types error messages observed); *system* (i.e., what is observed from a system point of view, such as an operating system crash); *application safety* (i.e., whether safety is violated); and *application mission* (i.e., whether the mission accomplished or not).

Powell et al. describe in [127] a method for assessing the timing robustness of real-time control software of autonomous systems. The approach is based on fault injection, in which random mutations are applied to valid sequences of requests issued to the functional layer of such systems. The faults used consisted of deleting, inserting, and re-ordering requests and proved to be useful in detecting several robustness violations. The approach was evaluated on an experimental planetary exploration robot, and only a few cases of incorrect behavior were observed.

Cámara et al. [10] proposed an approach, in 2013, for evaluating the robustness of controllers for self-adaptive software systems, depicted in Figure 3.8. The state of the system taken into account and is tracked through scenarios. Scenarios are sequences of events, and each scenario is described by the workload assigned to the system at a given time, its then operational conditions, and a set of changes to apply to the controller input. A change is an instance of the *changeload* model, which is used to describe mutation rules to apply to the inputs of the controller under test. These mutation rules are essentially the application of invalid inputs to the data types in messages that reach the controller, similar to the ones used by Ballista [80]. Disclosed robustness issues are classified according to an adapted version of the CRASH scale [3]. These concepts have been matured and formalized in [187]. The authors have also tried later to understand to what extent the use of a different target system, within a self-adaptive architecture, may impact the robustness of the controller [188]. In [189] the authors propose an approach that is based on the use robustness testing, with the goal of assessing the resilience of self-adaptive systems. Results show that the stateful properties of the controller strongly influences the resilience of the target system.

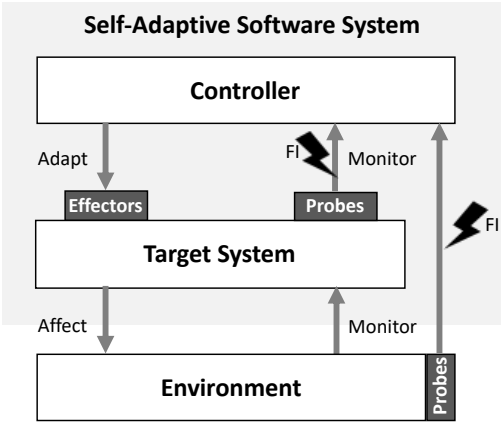


Figure 3.8: Evaluating the robustness of a self-adaptive system controller (adapted from [10]).

Thus, testing the robustness of the controller must be a priority for achieving resilience.

Hutchison et al. present, in [128], the Automated Stress Testing for Autonomy Architectures (ASTAA), a tool for automating robustness testing activities for autonomous systems (e.g., systems used in robotics or automation). In ASTAA, a tester must first specify how the target system works, including system interfaces, startup and shutdown sequences and safety invariants, which are the safety properties the system must respect (e.g., autonomous vehicles should not exceed speed limits). Test cases are then generated from this system specification, which intercept messages between distributed components and inject invalid values in fields and messages (multiple injections within the same message are also used). The authors evaluated the approach on 17 real-world autonomous systems, robots and robotics-oriented libraries (both commercial and academic), and disclosed numerous robustness problems, including severe cases where safety could be compromised.

Katz et al. propose a method for detecting execution anomalies in robotics and autonomous software, through robustness testing, by monitoring low-level execution behavior and performing anomaly detection on the results [185]. The first phase of the approach is based on running an instrumented version of the target system on nominal inputs and collecting summaries of execution (i.e., execution traces). The affinity propagation algorithm is then used to automatically cluster the collected execution traces, thus allowing to identify and categorize nominal system behavior. In the following phase, new execution traces are collected by running the target system with invalid and boundary inputs likely to induce safety failures (e.g., MAX_INT, NAN, -1), and these are compared against the previously obtained clusters of nominal behavior in order to detect anomalies. Large differences between these traces may be indicative of non-robust system behavior, and thus violation of safety properties. The authors evaluate the presented technique in a simulation on two robotics systems, one of which is a real-world industrial system, leading to the detection of robustness issues representative of unsafe system behavior.

To summarize, some of the approaches found on autonomous and adaptive systems do not take the state of the system into consideration [23, 127, 186], whereas the works by Cámara et al. [10, 187, 188, 189] explicitly consider the state of the system. Most of the approaches use fault injection as a testing technique [23, 127, 189], but one particular work relied on stress testing [186], an entirely different method which attempts to force the target system into an erroneous state through, for example, excessive resource consumption.

The fault models used are quite typical (e.g., invalid or boundary inputs [10, 128, 188]), although in some cases less usual faults are taken into consideration, such as timing-related faults [23] and inserting, re-ordering or deleting messages [23, 127]. Considering the increasing use of this kind of systems (e.g., in autonomous vehicles), we find robustness evaluation approaches to be scarce, especially considering the case of systems where the lack of robustness may affect other important properties of the system (e.g., safety).

3.3 Discussion

In this section, we discuss the main findings identified during our analysis of the state of the art. We do so while placing emphasis on answering the research questions presented earlier, in Section 3.1.2.

We begin by discussing research question **RQ-1 Which types of software systems are the subject of robustness evaluation?**, for which we found relatively diverse target systems being evaluated. However, we were able to fit the different systems into seven groups that we consider to be widely accepted classes of software. Figure 3.9 depicts the distribution and count of publications per system type, over the whole period where research on software robustness was found.

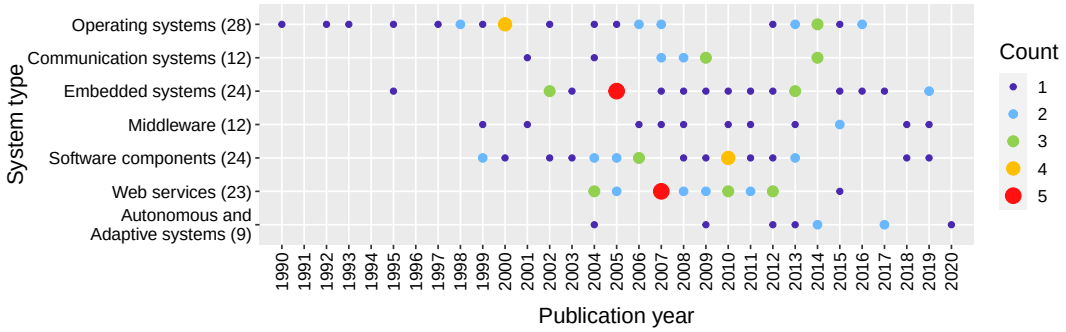


Figure 3.9: Distribution of publications per system type over the years.

As we can see in Figure 3.9, there has been a relatively large interest in the robustness evaluation of operating systems, embedded systems, web services, and software components, although, by nature, this latter type of software is a broad group. The evaluation of *operating systems* robustness has received some exploratory attention in the early nineties, but has mostly developed in the late nineties, due to the works on robustness conducted within the Ballista project [80]. Afterwards, a few works appeared, with authors benefiting from the lessons learned from the work carried out within the Ballista project and also proposing new approaches for operating system robustness evaluation. Recently, the attention has shifted to mobile operating systems, which is also a visible trend in Figure 3.9.

Communication systems have seen research being carried out in a few short bursts, spanning about a decade and a half. Considering their long existence, they do not seem to be the typical case of interest in robustness evaluation. Regarding *embedded systems*, the use of robustness evaluation techniques has also been traditionally important (e.g., in aerospace systems). With the increasing use and complexity of software in these systems (e.g., in modern vehicles) [190], it is not surprising that the area is still active even when considering the high specificity of this type of systems. Research on *middleware* has concentrated

mostly around the late 2000's, although we are also considering middleware management systems, in which we also observe recent work being carried out, especially in cloud platforms. As mentioned, the *Software Components* category is a typical target of research, with particular expression in the popular years of COTS usage [150].

Web services robustness evaluation has seen a peak of research being carried out in the late 2000's, with some work on Web Applications but with the majority focusing on SOAP web services. Research interest has clearly stopped and, at the time of writing, we were also not able to identify robustness evaluation research of more recent web service implementations, such as REST services. Finally, and more recently, research has targeted *adaptive and autonomous systems*, with most of the works being published in the last decade. This is somewhat expectable, given the recent interest, for instance, in autonomous and self-driving vehicles [191].

Robustness evaluation techniques have been applied to highly diverse systems throughout the years, with the heterogeneity and specificities of the target systems justifying the need for adaptations of existing techniques or the need for the definition of new techniques. At the same time, the open space for research is visible, with many specific subtypes of systems not really being evaluated for robustness. This is the case of specific systems like iOS or Windows mobile operating systems (for which close work on security evaluation already exists [192, 193, 194]), different types of middleware like streaming middleware (e.g., Apache Kafka [195] or Amazon Kinesis [196]), or REST services, which are now pervasive in the industry with major companies providing access to their services via a REST API (e.g., Twitter, Instagram, Facebook) [15]. More importantly, new types of systems have not yet been the focus of robustness evaluation. Thus, we were not able to identify research on topics involving, for instance, Cyber-Physical Systems [197] or Blockchain systems [198].

The second question **RQ-2 Which techniques are used to evaluate software robustness?** led us to focus on the techniques used in the robustness evaluation literature. Again, we found a large diversity of specific techniques being used which, however, fit into a few general cases. Figure 3.10 shows the total number and application of the different techniques over time.

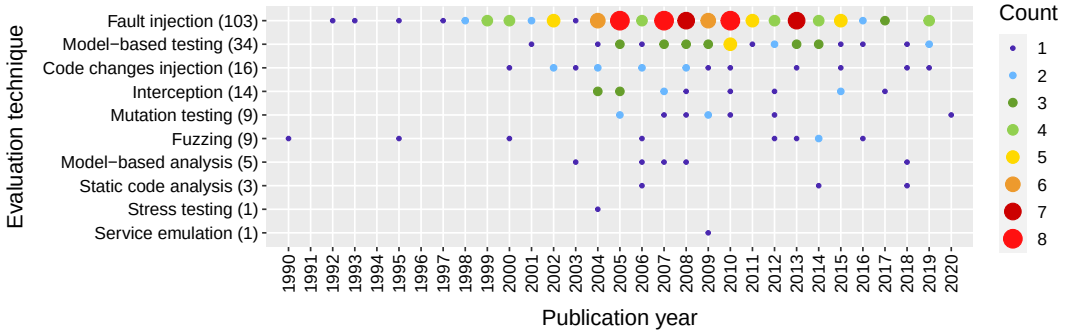


Figure 3.10: Distribution of publications per technique over the years.

Figure 3.10 makes it clear that robustness evaluation research tends to be mostly of experimental nature, with formal techniques being less frequently used. The distribution is essentially dominated by two groups of techniques, with most of the works using fault injection (i.e., about three quarters of the works) and the other major group including model-based techniques (e.g., model-based testing) and accounting for about one fourth of the works. Less frequently, some works use formal analysis techniques, such as invariant

analysis and abstract interpretation. The distribution of the most popular techniques is fairly regular, despite the presence of a few moments where we observe a higher number of papers being published (e.g., fault injection during the late 2000's).

Figure 3.11 shows the distribution of the different techniques in perspective with the type of system being evaluated.

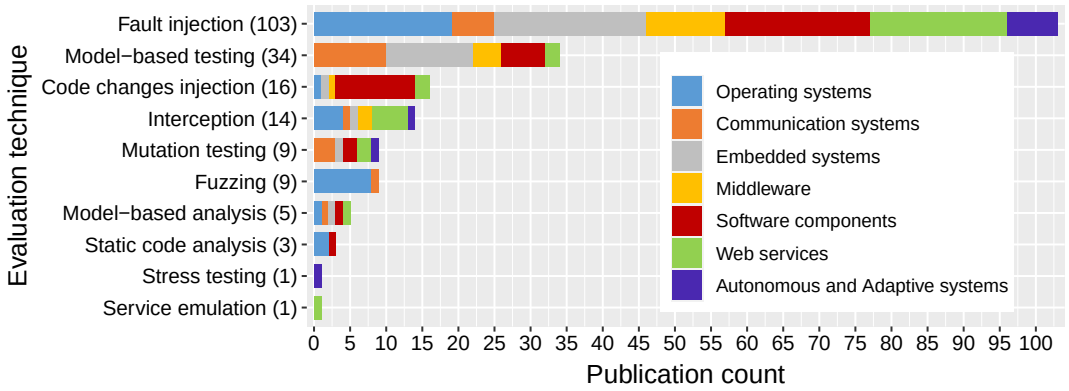


Figure 3.11: Distribution of evaluation techniques per system type.

Considering both the techniques and the system type, it is clear that fault injection dominates the plot and touches all types of systems. We also observe that model-based testing tends to have a stronger relative association with communication systems and also embedded systems. Also worthwhile mentioning is the stronger association of code changes injection with software components, and fuzzing with operating systems. Obviously, there are also cases of techniques not being applied to certain system types (e.g., code changes injection in autonomous and adaptive systems), which may be an indicator of a research opportunity.

Regarding the third question **RQ-3 Which are the targets used by software robustness evaluation approaches?**, we analyzed the prevalence of the evaluation targets (e.g., messages, API calls, function return values) used in the different approaches and their distribution throughout time. Figure 3.12 shows the outcome of this analysis.

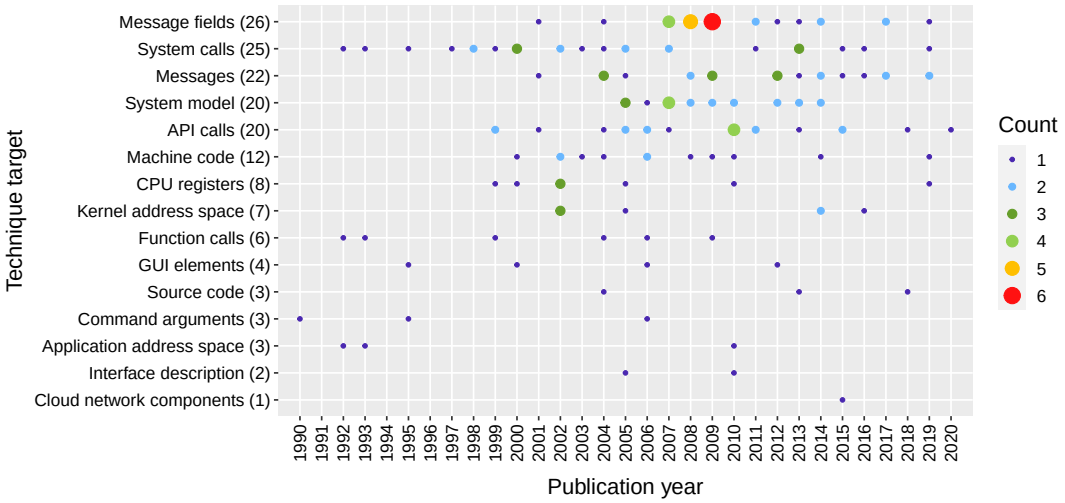


Figure 3.12: Distribution of publications per technique target over the years.

As we can see in Figure 3.12, five different targets (i.e., messages, message fields, system calls, API calls and system model) account for the vast majority of the represented cases. Regarding system and API calls, their use is frequent and we can see their application from early years until recently. The same occurs for messages and message fields, although in this latter case there are few additional gaps in time. If we consider the target code (i.e., source code and machine code), we end up with numbers close to the most popular techniques, although in a lesser extent. Some other technique targets are clearly less frequent (e.g., address spaces) and reflect some specialization of a certain technique that is applied to a specific type of system (e.g., cloud network components).

Figure 3.13 shows the different targets of evaluation, in perspective with the type of system being evaluated.

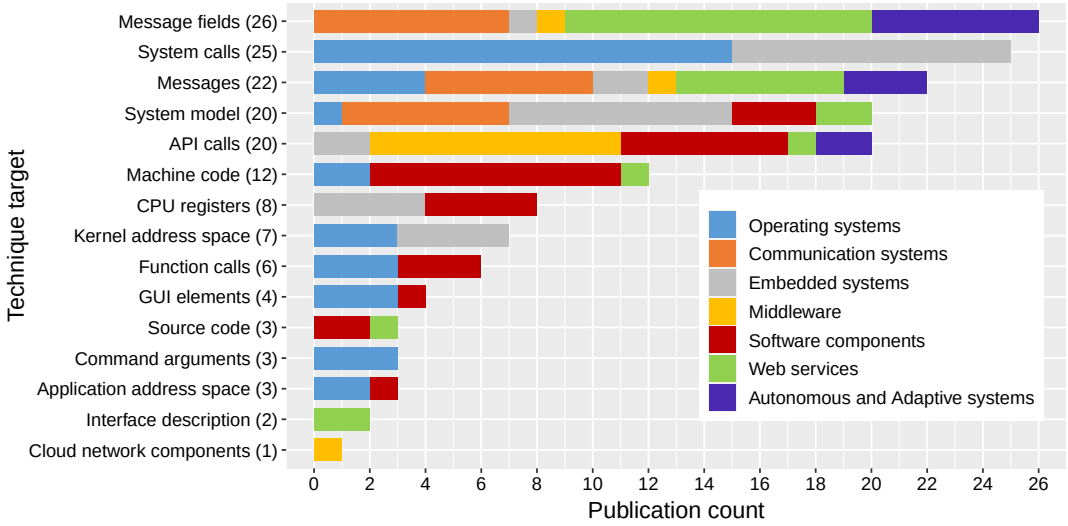


Figure 3.13: Distribution of evaluation targets per system type.

There are a few obvious cases of association which are visible in Figure 3.13, such as *system calls* or *kernel address space* being coupled with operating systems or embedded systems. Message fields appear strongly associated with web services, due to the large number of works based on fault injection applied to the fields of SOAP messages. The association of message fields with communication systems and autonomous and adaptive systems is also quite clear. In the former case, it is an obvious way for evaluating robustness of network-based systems. In the latter case, message fields carrying faulty inputs have been mostly directed to a part of the system, as a way of evaluating robustness of the whole system. The *messages* target is largely associated with communication systems and also web services, which is an expected exploration of the decoupling between clients and servers in this type of systems. A large portion of the target *system model* is associated with communication systems and embedded systems, which is related with the frequent need to verify certain highly critical properties in these systems. Finally, we must mention the case of *API calls*, which tend to be used along with middleware and software components, which is not surprising if we consider that these types of software offer APIs as a main entry point.

The fourth research question **RQ-4 Which types of faults are being used in software robustness evaluation?** aims at characterizing the types of faults used in robustness assessment research. Figure 3.14 shows the prevalence and distribution of the types of faults identified in the literature throughout the time.

As we can see in Figure 3.14, invalid inputs dominate the distribution, being used in about

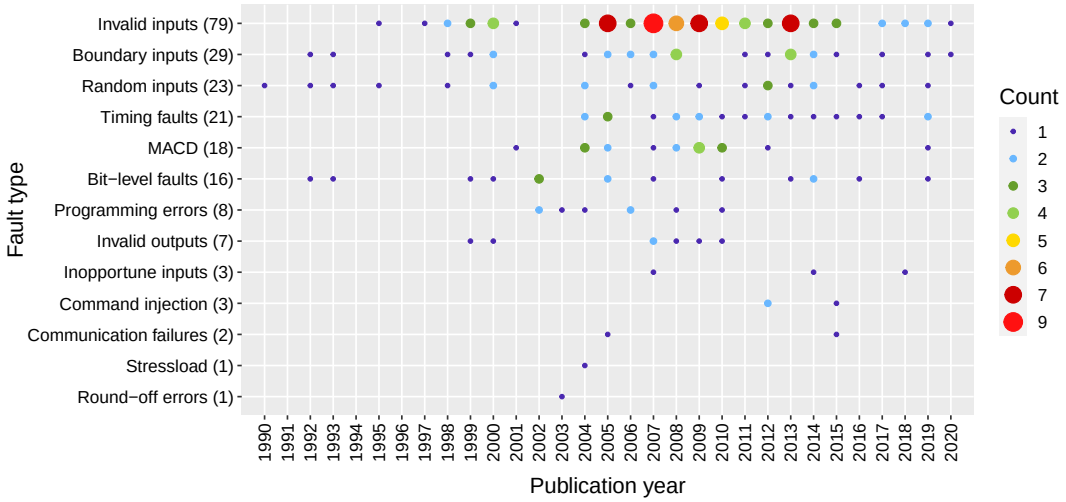


Figure 3.14: Distribution of publications per fault type over the years.

two thirds of the works. At a smaller scale, but still popular, we find boundary inputs, random inputs, timing faults, MACD operations (i.e., Move, Add, Change, Delete), and bit-level faults. Regarding the distribution over the years, we can observe a higher number of works using invalid inputs in the late 2000's (also MACD operations), with the same happening with works using timing faults. Boundary and random inputs have been used regularly throughout the time and also bit-level faults, although at a smaller scale.

Figure 3.15 shows the distribution of types of faults in perspective with the system types in which they are used.

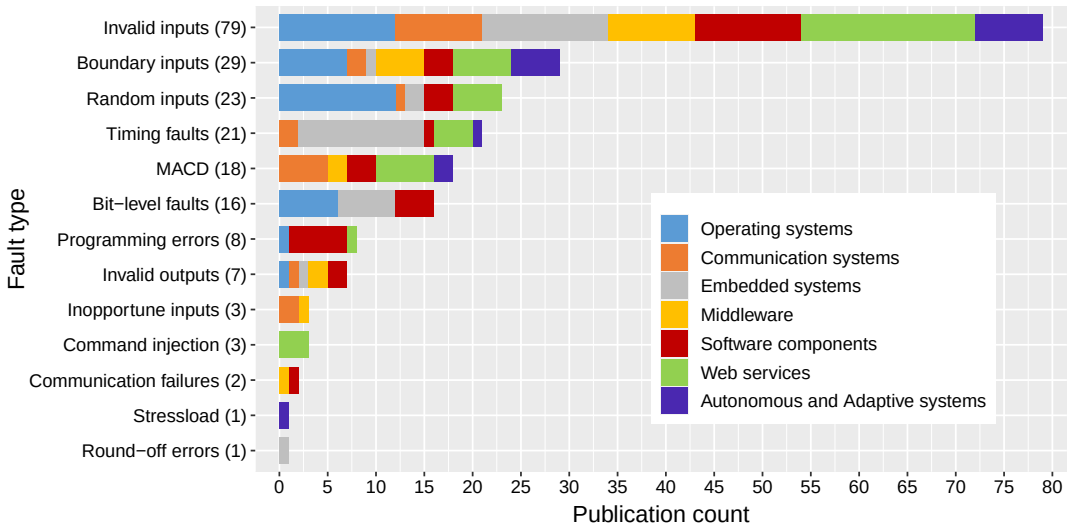


Figure 3.15: Distribution of fault types per system type.

As we can see in Figure 3.15, the two most frequent fault types (i.e., invalid inputs and boundary inputs) have been used in works that span across all identified system types (albeit with different prevalence), which is an indicator of their usefulness and applicability in robustness evaluation. About half of the works using random inputs are evaluating operating systems, and timing faults are also prevalent when the work evaluates embedded systems, which is expected given that many of these systems have to fulfill strong timing

requirements. We found MACD faults frequently being used with communication systems and Bit-level faults being applied only to works involving operating systems, embedded systems and software components.

Regarding the last research question **RQ-5 Which are the methods used to characterize robustness?**, we were aiming at understanding the typical ways of characterizing robustness which is an essential part of the process of robustness evaluation. We observed that most of the works (i.e., about two thirds) simply distinguish correct behavior from incorrect (i.e., non-robust). The classification scheme used by the remaining third is essentially a failure mode scale, in different configurations, that allows characterizing the behavior of the systems under evaluation. Table 3.10 conveys this information, and shows the ways of characterizing robustness, identified during the analysis of the state of the art, including the structure of each scale, the terms used to classify observations, and the references to each work using a certain scale. For layout reasons, we divided Table 3.10 in two parts, separated by double horizontal lines. The top part holds the classification schemes used by a small number of works, and the bottom part refers to classification schemes adopted by several works.

As we can see in Table 3.10, we observed the use of many different structures and sets of classification terms, with most of the works merely resorting to a binary classification (i.e., correct or failure). The remaining use more complex structures, with the CRASH scale [3], and their adaptations like the ones used in [6, 129, 188], dominating the distribution (i.e., CRASH variants account for about half of the works that use a non-binary classification). Some classifications have a more complex or a higher dimensional structure, which allows for a finer classification, but at the same time may add difficulties to the classification process itself making it more error-prone.

We can also see that, regardless of the schema used (and with the exception of the cases using a binary classification) there is a general concern with classifying the severity of the failure. This is also true if we consider the classification schemes that have categories and subcategories, although, in general, in these cases, the categories are related with a certain higher level aspect of the system (i.e., part of the system that was affected, mission completed or not), and not with the severity of the failure itself. This is in line with the discussion by Mukherjee and Siewiorek [199] that precisely shows the importance of categorizing test results so that the severity of an identified failure is registered. Also, the use of an established, adaptable and, ideally, portable classification scheme allows for robustness benchmarks to take place. As a summary, **we observed quite different structures for classifying robustness, with most concerned with classifying severity of failures and with CRASH [3] being the prevalent classification scheme used** (among the non-binary schemes).

In the following section, we highlight the gaps observed in our analysis of the state of the art, and identify open challenges for future research on robustness evaluation.

3.4 Highlights and research challenges

Robustness evaluation is an established area of research that we found applied to numerous types of systems. The origin can be traced to the work on operating systems by Miller [68] and, most notably, by the work carried out within the context of the Ballista project [80]. As time passed by, techniques started to move out of the kernel and expand to libraries or utilities and then to different kinds of systems.

Table 3.10: Classification schemes used in the state of the art

Structure	Classification terms		Works
4 categories, 12 subcategories	Interface	No error message returned, Correct error message, Incorrect error message	Chu et al. [23]
	System	Nothing observed, Module crash, BIP engine crash, OS crash	
	Application (safety)	Safety respected, A safety condition is violated	
	Application (missions)	Mission fulfilled, Mission failure	
9 categories	Correct, Incorrect, Crashing, Invariant preserving, Invariant breaking, Invalid input resistant, Invalid input crashing, Broken invariant resistant, Broken invariant crashing		Lei et al. [152, 153]
8 categories	Effectless, Incorrect output results, Real-time dysfunction, System crash, Memory access dysfunction, Exception trigger, Application hang		Nicolescu et al. [120]
3 categories, 7 subcategories	Application level	Application failure, Application hang	Arlat et al. [108]
	Interface level	Error status, Exception, Wrapper	
	Kernel level	Kernel hang, Kernel debugger	
7 categories	Pass, Pass with exception, RTI internal error, Unknown exception, Abort, Restart, Catastrophic		Fernsler and Koopman [133]
6 categories	Catastrophic failure, Restart failure, Abort failure, Raise unknown exception, Silent failure, Hindering failure		Pan et al. [132]
	System call returns correct status code (0), System call exits with unexpected error code (1), System call succeeds with invalid parameters (2), RTOS terminates benchmark (3), Test causes application restart and RTOS reload (4), Test causes cold system restart (5)		Dingman et al. [22]
3 categories, 5 subcategories	Physical	Server is down, Network connection to the server is broken	Ilieva et al. [31]
	Interaction	Timeout exception, Response error	
	Development	Named parameter incompatibility	
5 categories	Error code returned (SEr), Exception raised (SXp), Panic state (SPc), Hard reboot required (SHh), No-signaling state (SNS)		Kanoun et al. [78]
	Web server crash, Web server unresponsive, Resource use penalty, Wrong results, No impact		Mendes et al. [28]
	OS Crash, Application hang, Abnormal application termination, No impact, Wrong results		Madeira et al. [118]
4 categories	Kernel failure, Workload failure, File system corruption, No impact		Cotroneo et al. [117]
	No problems detected (FM1), System or applications hang (FM2), System crashes and reboots (FM3), Same as FM3 but there are corrupted files (FM4)		Mendonça and Neves [91]
	Detected failure, Silent failure, Hang failure, Crash failure		Zhou et al. [125]
	Operating System exception, Timeout, Correct result, Silent data corruption		Ahmad et al. [137]
	No failure (class NF), Deviation from golden run (class 1), Specification is violated (class 2), OS is unresponsive due to crash or hang (class 3)		Johansson et al., [90]
	Correct, Timeout, Error, Erratic		Durães and Madeira [147, 148, 149, 150], Moraes et al., [154]
	Correct output, Wrong result, System hang, Exception		Ruiz et al., [122]
3 categories	Restart, Abort, Pass		Xiang et al. [87]
	Correct, Abort, Hang		Costa and Madeira [156]
	Abnormal, Hang, Normal		Costa et al. [157]
2 categories	Critical, Not critical		Calori et al. [179]
5 categories	Catastrophic, Restart, Abort, Silent, Hindering	Azevedo et al. [26], Cardoso and Martins [129], Carrozza et al. [164], Cotroneo et al. [116], Cámara et al. [10, 187, 188, 189], Koopman and DeVale [19, 79], Koopman et al. [3], Kropp et al. [80], Laranjeiro et al. [6, 166, 167], Maia et al. [119], Napolitano et al. [136], Shahpasand et al. [124], Shahrokni and Feldt [143], Vieira et al. [177, 178]	
Binary	Robustness issue, No issue	Acharya et al., [76], Ait-Ameur et al. [105], Albinet et al. [77], Alnawasreh et al. [107], Batista et al. [5], Bauersfeld and Vos [158], Belli et al. [138], Bennani and Menascé [186], Cardoso et al. [27], Cavalli et al. [95], Chauvel et al. [130], Cong et al. [70], Cotroneo et al. [131], Csallner and Smaragdakis [146], Durães and Madeira [88], Feng and Shin [72], Fernandez et al. [151], Fetzer and Xiao [139], Forrester and Miller [89], Fouchal et al. [110, 111], Fu and Koné [21], Fu et al. [8], Ghosh and Schmid [7], Ghosh et al. [92], Giuffrida et al. [140], Hanna and Munro [165], Heckeler et al. [159], Hutchison et al. [128], Jin et al. [141], Jing et al. [102], Johansson et al. [103], Kaksonen et al [20], Katz et al. [185], Kuk and Kim [29], Kövi and Micskei [134], Looker et al. [168, 169, 170], Maji et al. [73], Martin et al. [171, 172], Mattiello-Francisco et al., [24, 112], Micskei et al. [25, 135], Miller et al. [68, 81, 82], Montrucchio et al. [83, 84], Naceur et al. [96], Oláh and István, [155], Pattabiraman and Zorn [180], Popovic and Kovacevic [97], Powell et al. [127], Quing-He et al. [142], Rabhi [173], Rodríguez et al. [121], Rollet and Saad-Khorchef, [113], Rollet and Salva [98, 109], Rychlý and Žouželka [174], Saad-Khorchef et al. [4], Salas et al. [175], Salva and Rabhi [30, 176], Sasnauskas and Regehr [74], Schmid et al. [93], Shahpasand et al. [123], Shelton et al. [69], Siblini and Mansour [9], Siewiorek et al., [85], Suh et al., [86], Tarhini et al. [114], Ufuktepe and Tuglular [144], Vasan and Memon [99], Velasco et al. [71], Winter et al. [115], Xiang et al. [100], Xu et al. [101], Yang et al. [106], Ye et al. [75], Zamli et al. [145]	

The main techniques used for assessing robustness are fault injection and model-based testing, which account for about three quarters of the research on robustness evaluation. Other less frequent techniques include mutation testing, fuzzing and also model-based analysis. About two thirds of the techniques aim at function calls (API, function, or system calls) and also messages and their fields. Invalid inputs clearly dominate the types of faults, followed by boundary, random, timing faults and then faults that operate at the message level. We also found that most works merely distinguish correct behavior from incorrect, although works using more complex schemes tend to adopt some variation of the CRASH scale [3].

There are clear challenges associated with the evaluation of robustness of new types of systems, among which we identify the selection of the technique (e.g., model-based, experimental), the target of the evaluation (e.g., an API, a message, message fields) the selection of the faults (e.g., timing faults, boundary values), and finally how to classify

behavior (i.e., the selection/adaption of a failure mode scale and how to retrieve the necessary information from the system to allow classification). It became clear that, at the time of writing, **there are types of systems for which robustness evaluation techniques are not known**. We primarily refer to research on *REST services* robustness, for which we did not identify any academic contribution. Due to their less rigid nature (e.g., there is no official standard service interface description [15]), the way these services are developed (e.g., inputs can be provided in numerous formats, inputs can be placed in the request payload or in a Uniform Resource Locator (URL) as part of a certain resource identifier), and the additional middleware involved, we believe REST services should be interesting candidates for robustness evaluation and, to the best of our knowledge, **there is no current academic contribution which supports testing REST APIs for robustness**.

We also did not identify work on *blockchain systems*, which carry several challenges, namely system complexity, strong integrity concerns, or timing requirements [200]. Another example is the case of *Cyber-physical systems*, characterized by strong interactions between their physical and computational parts [201]. These interactions will pose specific challenges (e.g., which fault injection locations to select, where should behavior observation points be set) related with the uncertainty of the environment and the nature of the overall system, which is many times a large-scale System-Of-Systems [202].

Some of the works analyzed evaluate systems that operate in safety-critical conditions (e.g., [5, 22, 105]). However, we found that there is **little focus on research that considers the interplay between robustness and safety**. As the world progresses towards autonomous driving cars [191] and flying drones [203], which hold strong safety concerns and execute under highly dynamic and uncertain environments, it is expectable that research advances robustness evaluation techniques to allow characterizing robustness in perspective with the safety requirements of such systems.

There has been recent advances in autonomous computing and increasing presence of artificial intelligence in the systems supporting our daily lives. Although we found a few works on autonomous systems, **to the best of our knowledge, there is currently no research on robustness evaluation of Artificial Intelligence based systems**. These systems may be placed under highly variable, limit, or unpredictable conditions and present non-deterministic behavior, being a known case of difficult verification [204], for which new robustness testing techniques may help in providing assurances that the system is able to resist to invalid or stressful conditions.

We have also observed that authors tend to use either a binary classification of the observed behavior (e.g., correct, failure) or a variation of the CRASH scale that properly fits the system under evaluation. Future **research on robustness evaluation would benefit from a standard method for classifying the robustness of systems**. It is likely that a single scale does not fit the whole diversity of existing systems, however, having a small set of classification schemes that could apply to a large variety of contexts or systems would simplify one of the researchers tasks, which is the selection (if a good candidate exists) or the definition of the different ways in which the system can fail. This would involve the definition of proper guidelines for researchers to make informed decisions. As mentioned, this would also foster comparability of robustness evaluation results [199].

As previously highlighted, the robustness of REST services has been largely neglected by researchers. However, other quality attributes (i.e., apart from robustness) have been the focus of researchers and practitioners in recent years. In the following chapter, we explore related academic and industry works on REST services testing.

This page is intentionally left blank.

Chapter 4

Related work on REST service testing

In this chapter, we analyze related work in the context of Representational State Transfer (REST) Application Programming Interface (API) testing. We begin by exploring, in Section 4.1, academic research contributions which propose approaches for testing different quality attributes of REST APIs. Then, in Section 4.2, we present an analysis of a considerable number of industry tools for REST API testing. We conclude this chapter with Section 4.3, where we discuss our main observations regarding the analyzed material and highlight a few limitations of the current state of the art and state of practice on REST API testing.

4.1 Studies on REST service testing

In this section, we analyze related studies on REST API testing, where we cover approaches which rely on multiple techniques for testing different aspects of these systems. This includes functional testing, security testing, and a more formal approach such as model-based testing, just to name a few. The technique targets, test inputs and classification schemes used on results, are also categorized for each of the identified approaches, which we have compiled in Table 4.1. We conclude this section with a few paragraphs summarizing the main patterns we identified among the analyzed approaches.

Figure 4.1 presents an example of approach for testing REST APIs using the fuzzing testing method (i.e., in essence, generate random or invalid inputs and use them in calls to the service, an approach that resembles robustness testing), which is inspired by the approach presented in [11] (and is further detailed later in this section).

As we can see in Figure 4.1, this example approach starts with the testing tool (i.e., essentially, a REST client application) parsing the service specification for the target REST API, from which it generates a set of valid requests (i.e., containing valid input, expected by the service as per its specification). These are passed to a request fuzzer component, which replaces the valid values in the requests with random or invalid values (i.e., ideally those that fall outside the boundaries defined in the specification). The resulting set of fuzzed requests is delivered to the main component of the testing tool, the tester, which iteratively sends each fuzzed request (now suitably formatted as an Hypertext Transfer Protocol (HTTP) request) to the target REST service. The service replies with an HTTP response, which the tool will receive and store for the user to manually inspect later.

Table 4.1: Techniques for testing RESTful web services

Systems	RESTful web services	Arcuri [205, 206, 207], Atlidakis et al. [11, 208], Chakrabarti and Kumar [209], Chakrabarti and Rodriquez [210], Ed-douibi et al. [211], Fertig and Braun [212], Godefroid et al. [213], Karlsson et al. [214], Liu and Chen [215], Pinheiro et al. [216], Segura et al. [217], Seijas et al. [218], Zhang et al. [219]
Techniques	Connectedness testing	Chakrabarti and Rodriquez [210]
	Differential regression testing	Godefroid et al. [213]
	Functional testing	Chakrabarti and Kumar [209], Fertig and Braun [212], Liu and Chen [215]
	Fuzzing	Atlidakis et al. [11, 208], Godefroid et al. [213]
	Integration testing	Arcuri [205, 206, 207], Zhang et al. [219]
	Metamorphic testing	Segura et al. [217]
	Model-based testing	Ed-douibi et al. [211], Fertig and Braun [212], Pinheiro et al. [216]
	Mutation testing	Liu and Chen [215]
	Property-based testing	Karlsson et al. [214], Seijas et al. [218]
	Security testing	Atlidakis et al. [208], Fertig and Braun [212]
Targets	API calls	Arcuri [205, 206, 207], Atlidakis et al. [11, 208], Chakrabarti and Kumar [209], Chakrabarti and Rodriquez [210], Ed-douibi et al. [211], Fertig and Braun [212], Godefroid et al. [213], Karlsson et al. [214], Liu and Chen [215], Pinheiro et al. [216], Segura et al. [217], Seijas et al. [218], Zhang et al. [219]
Inputs	Invalid inputs	Arcuri [205, 206, 207], Atlidakis et al. [11, 208], Ed-douibi et al. [211], Fertig and Braun [212], Godefroid et al. [213], Karlsson et al. [214], Liu and Chen [215], Zhang et al. [219]
	Random inputs	Karlsson et al. [214], Seijas et al. [218]
	Valid inputs	Arcuri [205, 206, 207], Atlidakis et al. [11, 208], Chakrabarti and Kumar [209], Chakrabarti and Rodriquez [210], Ed-douibi et al. [211], Fertig and Braun [212], Godefroid et al. [213], Karlsson et al. [214], Liu and Chen [215], Pinheiro et al. [216], Segura et al. [217], Zhang et al. [219]
Classification	Binary	Arcuri [205, 206, 207], Atlidakis et al. [11, 208], Chakrabarti and Kumar [209], Chakrabarti and Rodriquez [210], Ed-douibi et al. [211], Fertig and Braun [212], Godefroid et al. [213], Karlsson et al. [214], Liu and Chen [215], Pinheiro et al. [216], Segura et al. [217], Seijas et al. [218], Zhang et al. [219]

Ideally, the response should contain enough information for the user to discern whether an issue was triggered in the service as a consequence of the use of invalid input, or if the service aptly detected the illegal input and stopped it from causing any harm.

Chakrabarti and Kumar proposed Test-the-REST (TTR), an HTTP test suite execution and management framework designed to test functional attributes of RESTful web services [209]. TTR operates by first receiving a test case specification, which is provided by a user and written in an eXtensible Markup Language (XML)-based language, and is used as input to the test case validation module. Once validated, a test case is executed against the REST API under test. Once a response is obtained from the target API, it is used for verifying pass or fail conditions defined in the corresponding test case (e.g., asserting that the HTTP response holds a specific status code, or that the returned payload is represented as a specific media type). This process is repeated for all test cases a tester provides to the tool. TTR does not hold automatic test case generation capabilities and thus all tests must be generated manually by the user. The authors implemented TTR in the C# programming language and evaluated it on an in-house RESTful service, with results showing that TTR was able to disclose a considerable amount of software bugs.

In [210], Chakrabarti and Rodriquez detail a method for testing the connectedness of RESTful web services, a property that characterizes whether every resource exposed by the service is reachable from the base resource through consecutive HTTP *GET* requests. The proposed approach relies on the specification of a web service resource graph, which

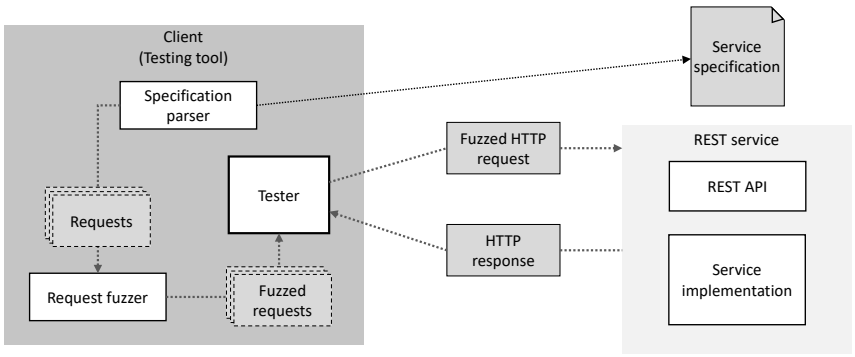


Figure 4.1: Fuzzing approach for testing REST APIs, inspired by [11].

contains the set of resources exposed by the web service, structured in such a way that allows to understand the hierarchies involved. This specification is described in WADL++, an extended version of the Web Application Description Language (WADL), and can only be provided by the web service developers. Then, a depth-first search is executed on the observable resources of the web service, with no previous knowledge of its structure. If the resulting resource graph differs from that in the WADL++ description, then the web service is not fully connected. The authors evaluated the approach on a REST service used internally by a company, and were able to disclose multiple functional and connectedness defects in the implementation. This approach does not, however, take into account security-related requirements of RESTful web services, such as authorization, without which it is expected that access to some resources be blocked.

A study by Seijas et al. [218] details a property-based testing tool, implemented in Erlang, for RESTful web services. It is capable of executing requests with the HTTP verbs that map to the CRUD operations (i.e., POST, GET, PUT and DELETE), and also provides an additional operation, *list*, which uses the GET method to list an entire set of resources. The actual test execution relies on the QuickCheck tool [220], which executes test cases implemented in Erlang and assesses the returned HTTP responses, using user-chosen criteria (from a set of predefined options) to output a test verdict. The authors showed an example application of the approach to the Storage Room and Google Tasks RESTful services.

Pinheiro et al. describe a model-based testing approach for validating the behavior of RESTful web services [216]. The service under test is modelled as a Unified Modeling Language (UML) protocol state machine, which emphasizes the transitions between service states, and is used to generate test cases. Test cases are tuned to maximize coverage of existing states and state transitions. Essentially, the proposed approach explores the states of the service under test in order to identify transitions that may lead to inconsistent or invalid states. The authors implemented the approach in the form of a Java-based tool, and validated it in a case study of a simple RESTful service with a small set of states.

Fertig and Braun introduce a model-based software development and testing technique for functional and security testing of RESTful APIs [212]. Based only on an abstract model of the target RESTful service describing the resources involved (i.e., which directly dictate the layout of its endpoints), the states of the service, and the respective state transitions, the technique automatically generates the source code for the API along with a large set of functional and security test cases. Functional test cases essentially verify the correctness of the HTTP responses returned by the service under test (e.g., valid HTTP headers, valid Uniform Resource Identifier (URI)s to other resources in the service), and security test cases focus on the authorization-related aspects of the service (e.g., access to some service

resources require authorization through username-password combination or an API key). The authors resorted to the Xtext and Xtend domain-specific languages to construct a test case templating framework, which may be extended for use with different kinds of testing types (e.g., performance, behavior). The proposed approach was evaluated on a simple RESTful API model composed only of four resources, where a total of 20000 different test cases were successfully generated.

Liu and Chen introduce a technique to automate test data generation for REST APIs based on a mutation testing approach [215]. Firstly, a WADL specification of the RESTful web service is required, in order to obtain information regarding parameter data types. Parameter constraints are then specified for each data type (e.g., maximum and minimum integer value), which allow to distinguish valid from invalid values, and are used for generating test data through techniques such as equivalence partitioning and boundary value analysis. This test data is then mutated through one of a set of seven mutation operators (e.g., randomly replace values), the best of which are selected by a genetic algorithm. Finally, the fittest test cases are executed in a Hadoop environment, which the authors claim increases the performance of the testing process. The approach was evaluated on a simple RESTful online shopping service system, with no particularly relevant observations in the results.

Arcuri presented a fully automated, white-box testing method which automatically generates integration test cases for REST APIs through evolutionary algorithms [205]. The approach requires full access to the source code of the RESTful service under test, and thus focuses solely on the developer's perspective. Using an evolutionary algorithm, the approach iteratively improves upon randomly generated test cases that aim to maximize code coverage and the amount of error status code responses from the service under test. The evolutionary algorithm uses mutation operators to perform small modifications on each generated HTTP request (e.g., changing the value of a given parameter to another value of the same data type), and continuously optimize its performance according to the code coverage and error responses metrics. The author implemented the approach in the form of a tool called EvoMaster, which has been described more in depth in another paper [206]. EvoMaster was evaluated on two open-source RESTful services and on a large industrial service, and a total of 38 real software bugs were successfully disclosed. In a later paper, Arcuri proposes a few extensions to the EvoMaster approach, including a larger experimental evaluation where a total of 80 real software bugs were disclosed in a set of five open-source RESTful services [207]. The most recent version of EvoMaster, proposed by Zhang et al. [219], improves the test case generation and optimization process by considering the semantics of HTTP methods used in RESTful services, resulting in a significant increase (i.e., about 40%) in code coverage and error response finding compared to previous iterations of the approach.

Ed-douibi et. al [211] propose an approach for automated specification-based test case generation for REST APIs. It first requires the generation of a meta-model of the OpenAPI specification of the REST API under test, which contains constraints and rules the API should comply with. The next step in the approach is using the meta-model of the OpenAPI specification to generate a *TestSuite* model. The concept of *TestSuite* model, introduced by the authors, is essentially a template for defining test cases. From this template, actual test cases are created according to a given set of rules. The generated test suites attempt to validate whether the service under test behaves according to its corresponding OpenAPI specification, and include both positive and negative test cases, with the latter relying solely on invalid input values. The approach was evaluated on a set of 91 OpenAPI documents of different RESTful service APIs, and the generated test cases held an average API coverage of about 76%.

Segura et. al propose an approach for disclosing faults in REST APIs through metamorphic testing [217]. Metamorphic testing helps solve the oracle problem in software testing by exploiting relations (i.e., metamorphic relations) among the outputs of multiple calls to the system under test. It works by introducing small variations in the testing environment while maintaining the same inputs to system calls (e.g., search for a given string with different numbers of results per page), and checking whether the outputs of such calls verify certain conditions, namely metamorphic relation output patterns (e.g., varying the paging size for a given query implies that the total result size of successive calls should hold an *equality* relation, as this value should not be affected by the amount of results per page). The authors introduce multiple output relation patterns, which work in the form of set operations (i.e., as in the mathematical concept of set), namely the *equivalence*, *equality*, *subset*, *disjoint*, *complete* and *difference* operations. The approach was evaluated on the Spotify and Youtube REST APIs, and a total of 11 issues were detected in the services, 10 of which were confirmed by the developers.

Atlidakis et al. present RESTler, a stateful REST API fuzzer [11]. RESTler analyzes the API specification of the target RESTful web service, and generates sequences of requests that automatically test the service through its API. The request sequences are generated in two ways: i) by inferring producer-consumer dependencies among request types declared in the specification (e.g., inferring that a request B should be executed after request A, because B takes as input a resource that is produced by A); and ii) by analyzing dynamic feedback from the responses observed during prior test executions in order to generate new tests (e.g., learning that a request C, performed after the request sequence A-B, is refused by the service, and such combination should be avoided in the future). The authors evaluated both test generation methods on the open-source Git service GitLab, and on several Microsoft Azure and Office 365 cloud services. A considerable number of bugs were found in all the tested platforms, which the respective service owners confirmed and fixed.

Atlidakis et al. introduce a method for testing the security of RESTful web services by exploiting vulnerabilities in their REST APIs [208]. The authors start by defining four security rule that are desirable in a REST API, namely: i) use-after-free - a deleted resource must no longer be accessible; ii) resource leak - a resource that was not created successfully must not be accessible; iii) resource-hierarchy - a child resource must only be accessible from its respective parent resource, and none other; and iv) resource-namespace - a resource created in a user namespace must not be accessible through another user namespace. Violations of such rules might allow an attacker to hijack service resources, bypass quotas, steal information from other users, or to corrupt the backend service state so that it no longer operates properly. The authors extended RESTler, a stateful fuzzer for REST APIs introduced in [11], to support the necessary capabilities to validate these four properties on a given RESTful web service, and evaluated the approach on a set of Azure and Office 365 cloud services. Results show that the approach managed to successfully disclose previously undocumented security bugs in some of the tested services.

Godefroid et al. introduce differential regression testing, a method for performing regression testing on RESTful APIs by comparing the behavior of different versions of a system against each other, using the same inputs [213]. The approach considers regressions in the API specification of the RESTful service, as well as in the actual software components that make up the service. To find regression bugs in both of these elements of the RESTful API, the approach is applied to pairs of different versions. In order to generate test cases, composed of sequences of HTTP requests, the authors use RESTler, a stateful fuzzer for REST APIs [11]. Based on the HTTP responses obtained during testing, the approach is able to automatically detect deviations and highlight possible regression bugs. The authors evaluated the approach across 17 different versions of the Azure networking APIs within a

3-year period, wherein five regression issues were detected in the official API specifications and nine in the software components of the service.

Karlsson et al. present a method to explore and evaluate the behavior of RESTful APIs using an automatic property-based testing method [214]. Test cases are generated by analyzing the OpenAPI documents describing the REST API under test. Inputs sent to the targeted endpoints are randomly generated, and can either be in line with the specification of the API under test (i.e., valid) or not (i.e., invalid). Additionally, test oracles, which are used to provide a verdict on the conformance of response data, are automatically generated from the target OpenAPI document. The proposed approach has been evaluated in an industrial case study at a company, and on the open-source software platform GitLab. Results showed that the approach is able to automatically find faults and gain insights of the API under test given a well-formed OpenAPI document.

In summary, we have primarily observed a wide diversity of testing techniques in the state of the art, including functional testing [209], regression testing [213], integration testing [205, 219] or security testing [208, 212], just to name the most common techniques. Some approaches resort to less conventional testing techniques such as connectedness testing (i.e., verifies whether all resources in the API are connected) [210], metamorphic testing (i.e., searches for relationships between outputs of API calls) [217], or property-based testing (i.e., essentially, asserts that the API under test verifies a given property) [214], just to name a few. The work by Atlidakis et al. [11], in particular, uses fuzzing as a testing technique, and the approach greatly resembles robustness testing.

Regarding test targets, the analyzed state of the art focuses solely on API calls, and the inputs used are either invalid (i.e., not in line with the API specification) [206, 215], randomly generated [214, 218] or valid (i.e., expected by the API) [209, 217]. Regarding classification of results, the analyzed state of the art does not resort to multi-level classification schemes for distinguishing between different test outcomes, but rather use only binary scales, which signal the existence of a problem (e.g., a security vulnerability, the violation of an expected property, failure to properly execute a functionality as per the design of the service), or the absence of any issue.

4.2 Tools for REST service testing

This section presents an analysis of a considerable number of industry tools for REST API testing, with only the exception of payware tools, for which we had no access. We experimented each of the tools described in the following paragraphs, and we provide here a detailed analysis of our experience in using them, as well as some limitations we occasionally observed. We categorized all tools according to a few dimensions, namely the type of tool, the API specifications it supports (if any), the test case generation mode (e.g., predefined options, scripting language), and the elements of HTTP responses which can be verified (i.e., test case assertion targets), which we summarize in Table 4.2. We conclude this section with a few paragraphs outlining our main observations regarding the analyzed tools.

Figure 4.2 presents the operation of a browser extension tool for REST API testing, based on the approach taken by Postman [12] (further detailed later in this section).

As per Figure 4.2, this tool is essentially a small application which operates on the environment of a web browser (i.e., a browser extension). The tool accepts as input JavaScript files describing the steps to take for testing the target REST API, including the target

Table 4.2: Industry tools for testing RESTful web services

Type	Application	Fiddler [221], HttpMaster [222], Jmeter [223], Katalon Studio [224], Pyresttest [225], SOAtest [226], SoapUI [227], Tavern [228], WebInject [229], Zerocode [230]
	Browser extension	Postman [12], RestBird [231], Restlet Client [232], vREST [233]
	Library	Airborne [234], Chakram [235], Karate [236], REST Assured [237]
	Online platform	API Fortress [238], APImetrics [239], Assertible [240], Ping-API [241], Runscope [242], vREST [233]
Specifications	API Blueprint	API Fortress [238]
	APIMatic	APImetrics [239]
	BPEL	SOAtest [226]
	I/O Docs	API Fortress [238]
	OpenAPI	API Fortress [238], APImetrics [239], Assertible [240], HttpMaster [222], Katalon Studio [224], RestBird [231], Runscope [242], SOAtest [226], SoapUI [227], vREST [233]
	RAML	API Fortress [238], SOAtest [226]
	WADL	SOAtest [226]
	WSDL	API Fortress [238], SOAtest [226]
	None	Airborne [234], Chakram [235], Fiddler [221], Jmeter [223], Karate [236], Ping-API [241], Postman [12], Pyresttest [225], REST Assured [237], Restlet Client [232], Tavern [228], WebInject [229], Zerocode [230]
Test cases	CoffeeScript	Ping-API [241]
	FiddlerScript	Fiddler [221]
	Golang	RestBird [231]
	Groovy	Katalon Studio [224], SoapUI [227]
	Java	REST Assured [237]
	JavaScript	Chakram [235], Ping-API [241], Postman [12], SoapUI [227]
	Karate DSL	Karate [236]
	Node.js	RestBird [231]
	Python	RestBird [231]
	Predefined	API Fortress [238], APImetrics [239], Airborne [234], Assertible [240], Fiddler [221], HttpMaster [222], Jmeter [223], Pyresttest [225], Restlet Client [232], Runscope [242], SOAtest [226], SoapUI [227], Tavern [228], vREST [233], WebInject [229], Zerocode [230]
Assertions	Status code	API Fortress [238], APImetrics [239], Airborne [234], Assertible [240], Chakram [235], Fiddler [221], HttpMaster [222], Jmeter [223], Karate [236], Katalon Studio [224], Ping-API [241], Postman [12], Pyresttest [225], REST Assured [237], RestBird [231], Restlet Client [232], Runscope [242], SOAtest [226], SoapUI [227], Tavern [228], vREST [233], WebInject [229], Zerocode [230]
	Headers	API Fortress [238], APImetrics [239], Airborne [234], Assertible [240], Chakram [235], Fiddler [221], Jmeter [223], Karate [236], Katalon Studio [224], Ping-API [241], Postman [12], Pyresttest [225], REST Assured [237], RestBird [231], Restlet Client [232], Runscope [242], SOAtest [226], SoapUI [227], Tavern [228], vREST [233], Zerocode [230]
	Payload	API Fortress [238], APImetrics [239], Airborne [234], Assertible [240], Chakram [235], Fiddler [221], Jmeter [223], Karate [236], Katalon Studio [224], Ping-API [241], Postman [12], Pyresttest [225], REST Assured [237], RestBird [231], Restlet Client [232], Runscope [242], SOAtest [226], SoapUI [227], Tavern [228], vREST [233], WebInject [229], Zerocode [230]

Uniform Resource Locator (URL) and HTTP method to use, the parameters and payload (if required), and the set of assertions to verify on the response (i.e., in this case, verify

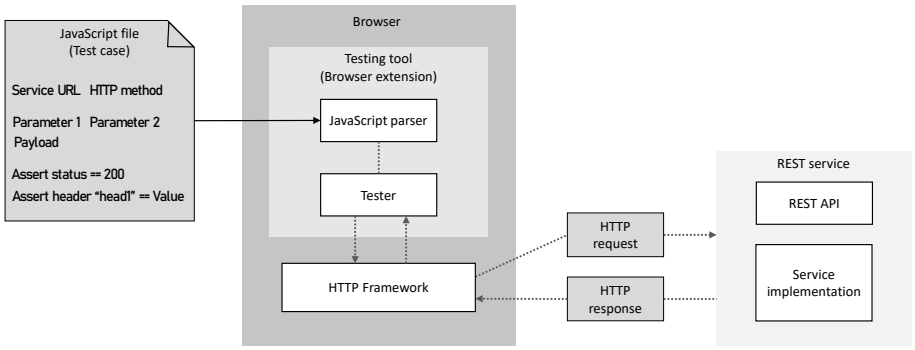


Figure 4.2: Browser extension tool for REST API testing, inspired by Postman [12].

that the response status code is 200, and that the response header named `head1` contains a specific `value`). The tool's JavaScript parser handles the test case script provided by the user, and generates a test case in the form of an HTTP request. For sending the HTTP request to the target API, the tool takes advantage of the HTTP framework existent in the browser and asks it to deliver the request. The request reaches the target REST API, which eventually replies with an HTTP response. The returned response is then caught by the browser's HTTP framework, and sent back to the tool, which will validate the response according to the assertions defined in the test case.

Apache JMeter [223] is a Java tool for testing web services using a wide variety of communication protocols (e.g., File Transfer Protocol (FTP), Simple Object Access Protocol (SOAP), Simple Mail Transfer Protocol (SMTP)) including HTTP, which means it is suitable for testing REST APIs. It is not a dedicated REST API testing tool, however, and therefore it does not support automatic API specification (e.g., OpenAPI) parsing and test case generation, and all test cases must be manually added by the user. An alternative path is for users to resort to a browser input/navigation recorder that is capable of outputting a JMeter-recognized file format, such as Badboy. Users can specify test case assertions (i.e., conditions to verify on test outputs) through the user interface of JMeter, i.e., as opposed to programmatically. Defining an assertion, however, is a multi-step process and may become time consuming for large REST APIs.

SOAtest [226] is a user interface-based tool for testing SOAP and RESTful web services, microservices, databases among other types of software systems. SOAtest provides HTTP request recording capabilities to facilitate test case generation, and to achieve this the tool resorts to artificial intelligence algorithms. Specifically, SOAtest provides a Smart Generator which monitors the activity of a user while navigating through a given online platform. The tool uses the collected information to establish meaningful patterns between requests and their respective content. This results in a set of randomly generated, but semantically meaningful, test cases that target the online platform the user chose. SOAtest is able to parse a large number of API specification types, including OpenAPI, RESTful API Modeling Language (RAML), WADL, Web Services Description Language (WSDL) and Business Process Execution Language (BPEL) (the latter two being used only by SOAP web services). Users may specify test case assertions to perform on HTTP response elements through the interface of SOAtest, but only by selecting from a predefined set of options, and no extension capabilities (e.g., through a scripting language) are supported.

Fiddler is .NET application with a graphical user interface for creating and executing API test cases over RESTful web services [221]. Fiddler allows users to take a client-side perspective for testing a REST API (i.e., performing HTTP requests and receiving

HTTP responses), as well as a man-in-the middle perspective, whereby the tool is able to capture network traffic and modify the corresponding data through fault injection, which allows for a less intrusive testing approach, as long as there is access to the network data between the API client and the REST service (i.e., encrypted data may be harder to work with). Assertions on HTTP responses can be defined through a predefined set of test case validators as well as through custom-made validators written in FiddlerScript, a scripting language used by Fiddler which is based on JScript.NET, a .NET version of JavaScript.

WebInject [229] is an HTTP testing tool. It presents a very minimalistic graphical interface for running test cases and monitoring response times, and test cases must be defined externally in XML files. When defining test cases, users must specify a unique test identifier, the target URL for the REST API to test, and the HTTP method to use, of which only GET and POST are supported. A few optional parameters can also be specified, such as `sleep` which waits a number of seconds after the corresponding test case has run before another one is executed. The options offered by WebInject for HTTP response validation (i.e., test assertions) are somewhat limited. For instance, verifying that the payload of an HTTP response contains a particular string `S`, requires that the test case XML file contain a `verifypositive` key with the value `S`. However, this will output true as long as the string is in the payload, independently of the positions or amount of times it appears in the payload, leading to somewhat coarse grained verifications. In order to isolate only a particular occurrence of the target string, the user must resort to regular expressions. Additionally, only a total of eight verifications of this type are possible, namely `verifiypositive(2/3/4)` for asserting that an element is present, and `verifynegative(2/3/4)` for asserting that an element is absent, which imposes some limitations for detailed test cases where many elements of HTTP responses are validated.

SoapUI [227], as the name suggests, is a tool that is primarily intended for testing SOAP web services, but because it supports the HTTP protocol it also provides REST API testing capabilities. It is a standalone Java application with a graphical user interface, and it supports OpenAPI descriptions, from which it extracts the necessary information to generate a set of test cases (i.e., each being a single HTTP request). Users of the tool may still manually setup test cases, and thus have greater control over some aspects (e.g., input data used in the tests). Test case assertions can be chosen in the graphical interface from a predefined set (e.g., message body `contains` or `not contains` an element, invalid status code), as well as defined programmatically through JavaScript or Groovy scripts, which provides more flexibility in performing complex verifications on the HTTP response content.

REST Assured [237] is a well-known Java tool for performing black-box testing over REST API, which takes advantage of the object-oriented capabilities of Java to provide its own Domain-Specific Language (DSL). Its verbose syntax is in the form of `Given` (an initial context), `When` (an event occurs), `Then` (validate outcomes) statements, as used in behavior-driven development where executable tests are structured in such a way that resembles the flow of natural language statements [243]. Test case assertions can be made on the elements of HTTP responses, such as validating response status codes, payload media types, among other response headers. REST Assured also allows for extensive verification of response payloads formatted in different media types, including the widely used JavaScript Object Notation (JSON). This tool does not, however, provide its own testing framework, and thus must be integrated with existing Java frameworks such as JUnit [244], meaning that test cases must be manually written in Java code.

HttpMaster is a dedicated HTTP testing tool, which is handled through a graphical interface with little to no code requirements for testing REST APIs [222]. The tool provides

functionality for parsing OpenAPI specifications, which facilitates test preparation by effectively eliminating the need to manually craft test cases. This is clearly ideal for large APIs (i.e., those with many operations or parameters). Upon parsing an OpenAPI specification, HttpMaster presents on its graphical interface a list of HTTP requests, one per API operation. Each request contains a URL, an HTTP method and a payload (e.g., JSON, XML or text data). Each request includes a `validation` element which allows for users to specify assertions to perform on the returned HTTP responses. However, the standard version of HttpMaster only allows users to verify if the response status code is in the range 1xx to 3xx (i.e., generally perceived as a success result), or the range 4xx to 5xx (i.e., which represent error code ranges). For users of the professional (i.e., payed) version, HttpMaster only allows to specify a single assertion on the elements of the HTTP response, in the form of a logical expression.

Runscope is an online platform for testing REST APIs which is operated through a regular web browser [242]. It provides users with the capability of importing sets of HTTP requests from multiple sources, including tools such as SoapUI, Fiddler and Postman, or from OpenAPI specifications. Alternatively, users can also manually create test cases in the form of HTTP requests using the graphical options provided in the interface. Included in these options are multiple types of assertions, where users can select the HTTP response element an assertion should target, and the verification to perform over it in the form of a condition. All elements in an HTTP response may be validated, such as status codes, response headers and payload, and users can also verify response size and time for performance testing.Additionally, Runscope allows to define pre-request (i.e., setup) as well as post-response (i.e., tear-down) scripts written in JavaScript, as well as passing variables between test cases which is important for test case sequencing (e.g., create a resource in the API, modify it, and then delete it, through separate test cases).

Airborne, developed in the Ruby programming language, is a client-side framework for testing REST APIs [234]. It is a Ruby module rather than a standalone application, and thus does not provide a user interface. Test cases are written in the RSpec syntax, which is a Ruby testing framework with a natural language-like programming syntax, thus making test cases easier to interpret. For instance, API calls in Airborne are performed by typing the name of the HTTP method to use followed by a space and the respective URI to target (e.g., `post "api.foo.com/bar"`). Assertions are defined through a set of `expect_<something>` commands, such as `expect_json`, `expect_json_types` (for JSON payloads) or even `expect_status` (for the response status code), just to name a few. As the method names imply, this framework only expects HTTP responses to contain JSON objects in the payload, which is the most widely supported payload format in REST APIs [15].

API Fortress [238] is a browser-based platform for functional testing of REST APIs. It is able to automatically generate test suites from numerous types of specification files, including OpenAPI, Postman collections, RAML, or API Blueprint, just to name a few, and it can also automatically generate mock APIs by recording user activity within the browser. The API mocking functionality is particularly useful for testing an API that is still in development but for which there is a specification. Testing REST APIs with API Fortress is entirely done through a graphical user interface, which allows for monitoring traffic and creating codeless tests cases. Test assertions are defined by selecting a condition and a target, such as specifying the condition `assert equals` on the target `status code`. API Fortress also provides functionality for automated test case generation, which works by creating validations for all response fields in an API specification. For instance, if a given operation is expected to output one or more values, API Fortress generates one data type validation for each, thus asserting that the values returned by the API are indeed of

the specified data type.

Postman [12] is a browser extension with a graphical user interface for HTTP testing, which is able to test REST APIs. In order to create a test case in Postman, users must specify the complete target URL, the HTTP method, all required parameters and, if necessary, the payload (e.g., a JSON object, a file to upload). Each test case may have one JavaScript file associated, wherein the user can specify a set of assertions to perform on the returned HTTP response. To help users with little coding knowledge, Postman also provides template code snippets for common verifications over HTTP responses (e.g., response contains an error status code).

Pyresttest [225] is an open-source REST API testing tool implemented in Python, which uses JSON and YAML files to specify test cases, thus eliminating the requirement for programming knowledge. Both the JSON and YAML file types use a key-value syntax, and Pyresttest expects test case files to have predefined keys such as `url`, `method` (i.e., referring to the HTTP method), `header` and `validators`, just to name a few. The `validators` keyword describes the assertions to perform on the returned HTTP response, and users can choose from a predefined set of logical operators to apply to a given target in the response, such as the standard equal, greater than, and less than operators (as well as the respective combinations of these), and other special operators such as `contains` (i.e., for collections), or `type` (i.e., for validating data types of response elements). Pyresttest does not provide any form of automated test case generation, and would thus require a lot of manual effort for testing large APIs.

Katalon Studio is a Graphical User Interface (GUI)-based application, implemented in Java and Groovy, for testing SOAP and RESTful web services [224]. It supports parsing OpenAPI specifications, and can automatically generate simple test cases, but users may still manually craft test cases with higher detail, namely in two ways: i) by writing Groovy scripts; or ii) through a GUI menu wherein testers can select from a predefined set of logical operations and control flow structures (e.g., an `if` statement which checks the content of a given value in the HTTP response). Using the latter method, Katalon Studio still generates the corresponding Groovy code in the background, and which can still be modified and enhanced with additional logic. Additionally, this tool supports recording and playback of user activity on a web browser, which may sometimes be the preferable alternative method for generating test cases.

Chakram [235] is a JavaScript framework for testing REST APIs with endpoints described in JSON files. It takes advantage of object-oriented capabilities of JavaScript to provide a behavior-driven development-like syntax, resembling natural language statements [243]. For instance, asserting that the payload of a given HTTP response is encoded in Gzip can be done in Chakram with the statement `expect(response).not.to.be.encoded.with.gzip`. This provides a verbose framework for specifying test cases, which greatly increases code readability. In fact, some connection elements in the syntax, such as `to`, `and`, `which` or `be` have no underlying logical effect on the outcome, and are merely used for readability improvement, as opposed to other elements which affect how an assertion is evaluated (e.g., `not` negates all subsequent assertion elements). The previous statement could effectively be reduced to the functional keywords, resulting in `expect(response).not.encoded.gzip`, which holds the same semantic content while being far less verbose. Chakram has no graphical interface and is not a standalone application, but rather a dedicated API testing framework which users can include in their own development code.

Restlet Client is a browser extension for testing REST APIs [232]. Test cases can only be created manually, by using the provided graphical menus, with no code requirements. Test case assertions are also specified through menus, which imposes some limitations

on the amount and variety of verifications to perform on HTTP responses. Users can target essentially all elements in a response, including status codes, response headers, and payload. Supported payload types include plain-text payloads, XML and JSON objects.

Ping-API is an online platform for testing REST APIs [241]. Although it is a payware platform, it provides a limited-time free trial. First, a project must be created wherein the user selects the base URL of the REST API to test. Afterwards, users can manually create test cases by specifying the HTTP method to use, the URI of the specific resource to target, and optionally the header and call parameters in the form of key-value pairs, as well as a payload. With this data, Ping-API automatically generates a JavaScript or CoffeeScript test case. Users may then freely modify the resulting script to enhance it and define assertions. The process for generating each test case is rather time-consuming, and important parameters such as authorization tokens must be specified before each test case is executed.

Assertible is an online platform for creating and executing tests on REST APIs. It provides OpenAPI specification parsing capabilities, and can also reuse Postman-generated test cases [240]. In both cases, Assertible automatically generates one test case per API operation, and includes a predefined assertion for verifying that the response status code is 200 (i.e., a success response). Additional assertions must be introduced manually in the platform, which can be done through predefined options in the menus of the graphical interface.

Karate is an open-source REST API testing tool which requires minimal programming knowledge for creating test cases [236]. It provides easy integration with any Java project and includes its own DSL. Test cases are described in documents called `feature` files, which contain multiple scenarios, each of which describing a single test scenario in the from of `Given-When-Then` statements, using a syntax which resembles natural language. For instance, referring to a response status code is done with the statement `status <CODE>`, where `<CODE>` is an HTTP status code (e.g., 200, 404), a specific HTTP method is referred to with `method <METHOD>`, where `<METHOD>` is an HTTP method (e.g., GET, PUT), and authorization data is referred to with `header authorization = <AUTH>`, where `<AUTH>` is an authentication method (e.g., basic HTTP authentication). For more experienced users, the Karate DSL also provides the functionality to make direct calls to JavaScript or Java libraries.

The Zerocode API testing framework [230] is an open-source community project built on top of JUnit [244], for testing SOAP and REST web services as well as software using other communication protocols (e.g., Kafka middleware). As its name implies, no programming effort is required for defining test cases, as indeed Zerocode tests are JSON-based, taking advantage of the file format's key-value structure. For example, the `url` key expects the URI of the target resource, the `operation` key expects a specific HTTP method (e.g., PUT), and the `request` key expects either an empty JSON object (i.e., for requests with no payload or header parameters) or an object describing the header (e.g., for authorization tokens) and/or body (i.e., for payload objects) of the HTTP request. Assertions are described under the `assertions` keyword, and can target the body of the HTTP response or the header elements, such as the status code. Zerocode supports the standard logical comparison operators (e.g., greather than, equal to), to be used in defining assertions for particular response elements (e.g., header, payload), and also includes some specialized validation operators, such as for checking if a string contains a given sub-string.

APImetrics is an online testing platform which, like some other payware API testing tools, also provides users with a free time-limited trial [239]. It provides a wide array of authentication protocols, including basic HTTP authentication and OAuth versions 1 and 2,

which are quite common in public REST APIs. APImetrics allows users to parse OpenAPI and APIMatic [245] specifications, as well as import HTTP requests from Postman. Test case assertions are specified through conditions in the form of `if-then` statements, which are constructed in the drop-down menus of the graphical user interface through, and thus only provide users with predefined options. The `if` part of the statement allows users to specify three elements: the part of the response to target (e.g., status code), a comparison operator (e.g., equals, contains, does not exist), and a value to compare to. The `then` part of the statement allows users to input a value to a global variable and to assign a `Pass`, `Warning` or `Error` label on the test case, depending on the result of condition in the `if` statement. Users may select one of response status code, response header, payload, response time and response size when defining an assertion. APImetrics does not allow, however, users to logically `AND` or `OR` multiple conditions together to create more complex verifications on the HTTP responses.

Tavern is a testing framework which does not require its users to have programming knowledge [228]. Tavern is a lightweight API testing tool built in Python, using the pytest testing framework as a base. Test cases are defined in YAML files by using predefined structures (e.g., the `request` key contains a `url` key and a `method` key), and assertions work by defining, inside the `response` key, the values to expected in specific HTTP response elements (e.g., the `status_code: 200` key-value pair, tells Tavern to assert that the HTTP response for the given test case contains the HTTP status code 200). A downside of the test case syntax used by Tavern, is that a user is unable to verify the value of a single element of a JSON object in the response payload, for example, and instead the entire payload must be exactly matched in the test case definition (i.e., otherwise, the test case will fail).

vREST is an online platform and browser extension for testing REST APIs [233]. It offers request recording capabilities, allowing to gather data regarding parameter values used in API operations in order to automatically generate test cases, and thus reduce the amount of work done by the user. This is particularly useful for testing large REST APIs. Users may also manually create test cases through the graphical interface, which offers a range of predefined options including the HTTP method to use (i.e., one of GET, POST, PUT, PATCH, DELETE or OPTIONS), and the header parameters and the request payload. Assertions on returned responses are also defined through predefined options in the interface, and users may target the response status code, response headers, payload, and timing-related measures. Additionally, vREST provides API mocking capabilities, and automatic test case generation from analyzing OpenAPI specifications and Postman test cases.

RestBird is an application that runs on a Docker container, which is accessible to users through a local network endpoint and can be used from a browser [231]. In RestBird, test cases are created through a graphical user interface, where users can identify the target URI of the API under test, the HTTP method to use, the header parameters of the request (e.g., authentication information) and, optionally, the payload. Pre-request scripts can also be created for setting up the required testing environment, and can be written in Python, Node.js or Golang. Validations on the HTTP responses is also defined in scripts, where testers use a single boolean-returning function for performing their desired assertions on the content of the response (i.e., returning `false` from such a function means an assertion failed, and the test did not pass). Recording and playback capabilities are also provided in RestBird, which helps reduce some of effort on behalf of the user by automatically creating test cases based on user activity.

In summary, most of the analyzed tools for REST API testing are built as standalone applications, spanning multiple programming languages such as Java [224, 227], Python

[225, 228], and C# [221, 222], just to name a few. Most of these have graphical user interfaces to facilitate user interaction, leading to a simplified process of test case generation and executing (e.g., by using predefined options in drop-down menus, rather than having to write code [222, 226]). Others are implemented as smaller modules or libraries which are meant to be integrated with other frameworks through code [234, 235, 237], and the focus here is shifted from user interaction through graphical interfaces, to object-oriented APIs with a syntax resembling that of natural language, leading to test cases with human-readable code.

A few tools take advantage of the inherent HTTP support provided by web browsers and are implemented as extensions, where users can organize test cases, using simple graphical interfaces, which are then executed over the browser's own HTTP framework [12, 232]. Alternatively, some tools are located on remote online platforms and are handled through a web browser, relieving the user of any local resource consumption [233, 242].

We have also observed that about half the tools implement support for different REST API specifications, such as the lesser known API Blueprint [238] or APIMatic [239], as well as more the common RAML [226] or, in particular, OpenAPI [222, 224, 242], which is shown here to be widely supported. A few tools even support SOAP-related specifications, such as BPEL [226], WADL [226] and, in particular, WSDL [226, 238]. The applicability of these specifications to the RESTful context is not recommended due to the inherent differences between these types of systems.

Another differing factor among the analyzed tools refers to the amount of options that are provided to users for creating test cases. The vast majority provide only predefined options to select from [223, 229, 240], and this only occurs in tools that have a graphical user interface, where predefined options are presented in drop-down menus, for example, with no possibility of extension. The remaining tools provide support for programming languages such as JavaScript [235], Python [231] or Groovy [224] (just to name a few), where users have more freedom of choice for defining test cases, especially for complex tests (e.g., large conditions for verifications, setup and tear-down tasks which must interact with external systems). Also, a very small set of tools impose limits on the diversity of HTTP response elements that users can validate through assertions, namely HttpMaster [222] which only supports validation of response status codes, and WebInject [229] which only supports validation of response status codes and payload. In both of these tools validation of HTTP response headers is not supported, quite unlike the vast majority of the analyzed tools, which can target essentially any element of an HTTP response [230, 236, 241].

4.3 Discussion

In this section, we present a discussion regarding our observations of the analyzed approaches for testing REST services. We first focus on related academic works, and then highlight a few relevant aspects of the analyzed industry tools.

Regarding **studies on REST API testing**, we have identified a considerable amount of works that were published within the last decade. These approaches encompass an heterogeneous set of testing techniques, while generally holding similar characteristics. Figure 4.3 depicts the distribution of techniques for testing REST APIs throughout the years, and in the following paragraphs we discuss a few observations regarding the analyzed state of the art.

Regarding Figure 4.3, note that the sum of the values between parentheses in the vertical

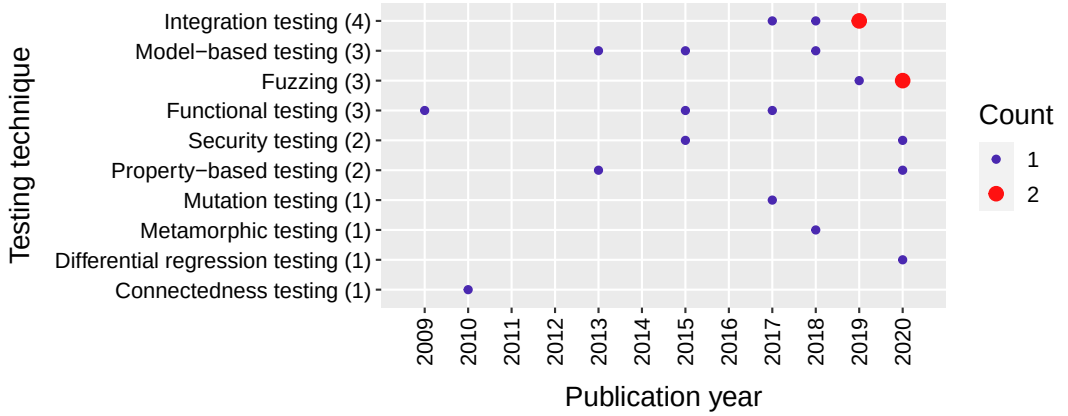


Figure 4.3: Distribution of REST API testing techniques over the years.

axis (i.e., 21) is greater than the total amount of studies analyzed (i.e., 16), and this is a consequence of a few approaches using two or even three testing techniques. Within a time span of 11 years, many different testing techniques were applied experimentally to REST APIs, and they generally hold distinct goals. The overall distribution of publications over the years shows a gradual increase in research interest for exploring novel REST API testing solutions, which is expected given the wide use of this type of system in modern online platforms [15].

About half of the analyzed approaches encompass techniques which were used multiple times over the years, which could hint at the fact that researchers have found a practical use for them in the context of REST API testing. This includes techniques such as integration testing (i.e., guarantees that different modules or systems behave as expected when integrated), model-based testing (i.e., formal approach that models system characteristics, such as behavior), fuzzing (i.e., fault inject-like approach that generally uses random or invalid inputs), and functional testing (i.e., validates whether designed functionality is correctly implemented). The remaining testing techniques, on the other hand, show a pattern of scarce use, and are mostly related with one-off experimental approaches. This includes mutation testing (i.e., modifies system specifications or models to generate invalid or unexpected test data), metamorphic testing (i.e., explores relationships between outputs to API calls), differential regression testing (i.e., compares the behavior of different system versions against each other using the same inputs), and connectedness testing (i.e., ensures that every resource in the API is accessible from every other resource).

We observed little to no diversity regarding the other characteristics that compose the analyzed approaches, such as technique targets (which focus only on API calls), testing inputs (which encompass only three classes), and classification schemes (which rely purely on the binary categorization of the existence or absence of an issue in the system under test). Testing inputs, in particular, vary only between the use of invalid inputs (i.e., those outside of what the service expects), random inputs (i.e., entirely stochastic values) or valid inputs (i.e., using values that the service expects). Note that many studies rely on the use of valid inputs because, otherwise, unwanted perturbations could be introduced into the system and interfere with the testing goal of the approach (e.g., triggering input verification issues at the interface would disrupt metamorphic testing activities). We can thus conclude that, while **there is a wide array of testing techniques that have been experimentally applied to REST APIs**, we have generally observed that **existing approaches hold many similarities regarding the technique targets, the types**

of input used for testing, and the classification performed over test results.

Finally, in the context of **industry tools on REST API testing**, we identified and experimented with a total of 23 different tools. These, however, do not make up all of the state of the practice, and some other tools exist but are payware (i.e., a license must be purchased to use them), and therefore not as easily accessible. The tools generally fit into two categories: i) standalone tools with a graphical user interface; and ii) tools with no user interface, built as libraries or modules to be integrated programmatically with larger frameworks.

Regarding the tools which provide a graphical user interface (i.e., 70%), the focus is generally placed on increasing usability and interaction with the tool, by simplifying the process of setting up the desired testing environment and creating test cases for a given REST API. This is mainly achieved by presenting users with simple graphical elements for defining the data required to communicate with a REST API, including the URI of the API endpoint to test, the HTTP method to use (e.g., POST, DELETE), HTTP request headers (e.g., for specifying authorization data), and a request payload if necessary (e.g., a JSON object describing a resource to create using the POST method). Then, the user must also define the assertions (i.e., validations) to perform on the HTTP response that is to be returned by the API, and here most tools (i.e., about 48%) provide only a predefined set of options (e.g., selected through drop-down menus).

In general, the idea is to select an HTTP response element to target (e.g., an HTTP header with a given name), a boolean condition to test its value with (e.g., greater than or equal to), and another value to compare to. Users are not able to `AND` or `OR` multiple conditions to create complex assertions, and the set of provided boolean operators cannot be extended (e.g., to include a *contains* operator for verifying if a sub-string is present within a larger string). Alternatively, a few tools (i.e., about 26%) provide support for writing test cases in scripting languages, where users have much more freedom of choice and flexibility in their approach (e.g., it is possible to communicate with external systems during testing, which may be useful for populating the RESTful service with the necessary resources before running a test, for example).

The other group of tools (i.e., those built as libraries to be integrated in larger frameworks), represents only 30% of the identified tools, and here no predefined options are given for building test cases. Instead, users must manually write all the code relative to the test cases (e.g., access the desired HTTP response element and apply a logical verification to its value), as well as to import the library itself, and perform setup and tear down tasks, thus making it more time consuming to construct individual test cases. To facilitate this process, however, these few tools take advantage of object-oriented capabilities of programming languages, and provide users with simple APIs that resemble natural language statements, which in turn also makes code more readable and thus test cases easier to interpret.

In essence, we have generally observed that **most tools for REST API testing aim at providing users with increased usability rather than functionality, compromising overall diversity of possible testing approaches in favor of accelerating the creation of multiple, often trivial, test scenarios**. However, we also note that **some of the tools provide support for programmatic test case creation environments, which allow users to manually craft detailed test scenarios, but with an increased time consumption**. Almost half (i.e., about 43%) of the analyzed tools also provide support for REST API specifications, including RAML and the widely used OpenAPI, just to name a few. The specifications are automatically parsed and, in a few cases, one simplified test case per API operation is automatically generated. This shows that **there is a clear effort by industry tool developers for keeping up-to-date**

with REST API quasi-standards such as the OpenAPI specification [15], as well as for providing support to a wide array of alternative specification types.

Throughout the previous decade, researchers as well as practitioners have acknowledged the need for testing REST services, and many different academic approaches and industry tools (supporting various quality attributes) have been proposed. Robustness is an important quality attribute in REST services, especially considering their continuous exposure to the Internet. To the best of our knowledge, and given the comprehensive analysis of the state of the art presented in this dissertation, no approaches have been proposed for evaluating the robustness of REST services. In an attempt to cover this gap, in the following chapter we present an approach and tool for carrying out robustness tests over REST services.

This page is intentionally left blank.

Chapter 5

Approach and robustness testing tool architecture

In this chapter, we describe our approach for testing the robustness of Representational State Transfer (REST) services. We begin by introducing, in Section 5.1, the main concepts that support the approach. Section 5.2 closes this chapter, where we overview the internal architecture of our black-BOX tool for Robustness Testing of rest services (bBOXRT), and explain the different roles and responsibilities of its main components and how they collaborate to support each of the steps of the approach.

5.1 Approach overview

The concept behind our approach is shown in Figure 5.1. In practice, using information regarding the interface of the service under test (i.e., Application Programming Interface (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 its interface description document. This information is gathered to be used in the next steps, and it includes the Uniform Resource Identifier (URI) of available resources and the Hypertext Transfer Protocol (HTTP) methods they implement, input and output datatypes, error codes, and example requests.
- **Step 2: Workload generation and execution**. Valid (i.e., correct according to the specification) requests are generated and sent to the service. This allows to understand the behavior of the service in the 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 JavaScript Object Notation (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.

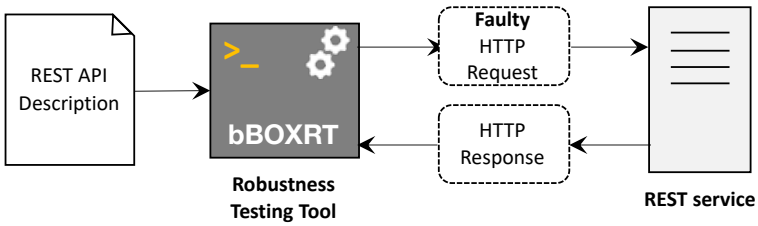


Figure 5.1: Conceptual view of the approach.

The following section explains these steps in further detail, and maps each of them to the different software components that compose our tool, bBOXRT.

5.2 Tool architecture and operation

The architecture of the system is shown in Figure 5.2, which depicts bBOXRT as a container whose scope is delimited by a rectangle. It is comprised of multiple components that interact with each other and/or with external entities, as detailed in the following paragraphs. Note that the interface for bBOXRT works through the command line, and its source code and documentation are available at [32].

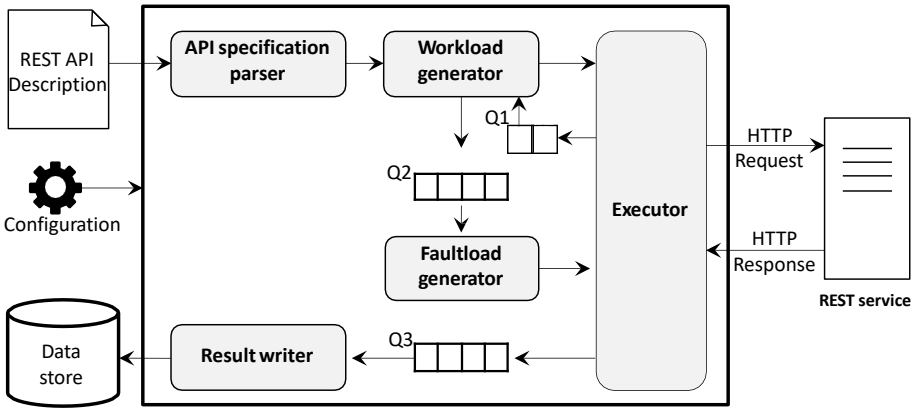


Figure 5.2: bBOXRT architecture.

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 [17]), 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 [15] and this is the reason we chose to support it, by default. However, users may extend the tool to support other API specification languages (e.g., RESTful API Modeling Language (RAML) [37]).

The main idea behind this first step is to identify and extract relevant information for testing the service. An API specification defines the Uniform Resource Locator (URL) of the target server 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 [16], typically POST, GET, PUT and DELETE (PATCH and HEAD may also be found in some APIs). Each set `{endpoint, verb}` is often called 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 execution and generation**, 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. 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 5.2). 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., some operations may be dependent on the execution of others). Requests holding a success status code (e.g., 200 OK) are placed in a queue (*Q2* in Figure 5.2) and will be used in the next step of the approach. The current step ends if the user sets a maximum time limit (i.e., by specifying the `WL_MAX_TIME` parameter), otherwise all generated requests are executed.

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 5.2) that keeps track of the possible injection locations in a request. This component keeps track of which parameters in a given request have been covered with faults (and with which faults) and, thus, it guides the fault injection process by preventing the injection of faults at an already explored location.

Table 5.1 shows the fault model currently used by the *Fault Generator*. The faults are organized by data type and format as defined in the OpenAPI specification [17], where 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. Note that for the string fault types, printable or non-printable refers to the respective portions of the ASCII table, and malicious refers to Structured Query Language (SQL) injection strings (we used a set of 801 SQL injection strings obtained from [246]). Also, regarding the date and date-time formats, note that the faults of the former are also applicable to the latter.

For each parameter of each request, the set of applicable faults (i.e., those that match the parameter's data type) is exhausted 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 the presence of a certain fault. For certain types of faults, it also allows to achieve a greater

Table 5.1: 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

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 exhausted 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 5.2) 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**.

bBOXRT does not currently fully automate the analysis of the service behavior, due to the typical complexity involved in the identification of robustness problems (possibly 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. In the future, we intend to explore the applicability of Machine Learning algorithms to this task, which we plan to integrate in the tool. 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 analyzed and classified with a failure mode scale, such as the CRASH scale [3], in which we perform minor adjustments to fit the particular case of REST services. CRASH distinguishes the following cases of failure:

- **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 between the different cases of severity.

The service behavior should also be characterized in a more detailed manner. In previous work [167], and based on the analysis of the results of robustness testing applied to Simple Object Access Protocol (SOAP) web services, the application of a set of behavior tags was proposed. 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 5.2 presents an adaptation of the set of tags (each with its abbreviation between parenthesis) used in [167], which we now tuned to fit the context of REST services. We performed minor adaptations to the tags as some SOAP elements do not exist in REST (e.g., Web Services Description Language (WSDL) documents). Also, we extended the set with two extra tags, parsing error and allocation error, after analyzing the experimental results. A parsing error occurs when it is clear, from the error message, that an exception

was thrown as a consequence of failure to parse input or improper input handling. An allocation error exists when the error message indicates that an exception was thrown as a consequence of the allocation of memory addresses (e.g., an out-of-memory exception). In the following chapter, we describe our practical experiment for evaluating the tool, and its respective results.

Table 5.2: 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 or development specific information is revealed. This information is, however, generally too vague or too context-specific to allow us 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

This page is intentionally left blank.

Chapter 6

Experimental evaluation

In this chapter, we describe the experimental campaign carried out to show the practical usefulness of our black-BOX tool for Robustness Testing of rest services (bBOXRT). In Section 6.1, we detail the process used to carry out the experiments, and in Section 6.2 we go through the main results. Finally, in Section 6.3, we conclude with the main findings observed during the experiments.

6.1 Experiments description

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. We tested 76% (1351) of the 1775 available operations, because, at the time of writing, the tool does not yet support generation of certain, less usual, payloads (i.e., media types such files or MessagePack messages [247]).

- 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 Representational State Transfer (REST) services was selected from the online database APIs.guru 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 previous works [248, 249, 250]. We performed tests against 823 operations (out of 1012) 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, to be part of this group. We performed tests against 70 of the 106 Docker Engine REST service operations [251]. 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$ and, thus, strong reliability requirements are involved given that any failure may have impact on the business. For this service, we had no access to source code, but had a rather complete interface description and also direct contact and feedback from the company's developers.

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 [252], 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 [253]. All implementations were created in-house, and the Vx0 versions were developed by one Early Stage Researcher and the VxA versions by another one. Vx0 and VxA mostly differ in how the developers built the Structured Query Language (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 [254] of 4.67 (measured by the Statistic plugin [255] of IntelliJ IDEA 2019.2.3 [256]), 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 [257] (with the Find Security Bugs plugin 1.10.1 [258]), 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 user input being directly concatenated with SQL queries. Spotbugs also identified further issues, but they were mostly bad coding practices or false-positives.

The experiments carried out against the 1351 operations used the same values for the configuration parameters of bBOXRT, namely: `WL_REP=10` each request parameter is generated 10 times; `WL_RETRY=true` each unsuccessful request is retried once; and `FL_REP=3` 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 nor 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 399901 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 analyze, 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 68372 responses that we analyzed in detail. For each signalled response, we analyzed its whole content 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 68245 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.

6.2 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 1351 service operations tested (i.e., in about 12% of the operations). Of the 7167 parameters available in these operations, 525 were involved in at least one robustness problem. The tool actually disclosed 24372 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, 23764 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 5.1), 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 6.1 presents the distribution of the 44 types of faults that resulted in the 24372 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 24372. 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, as we were unable to determine what was the reason for the failure.

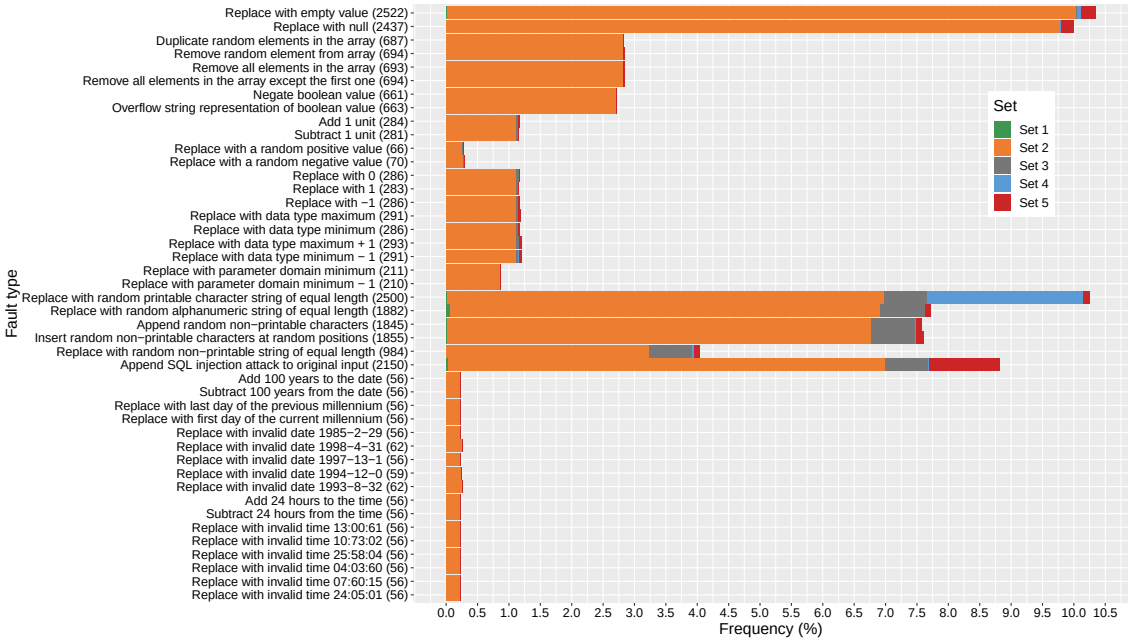


Figure 6.1: Distribution of the successful faults over the service sets.

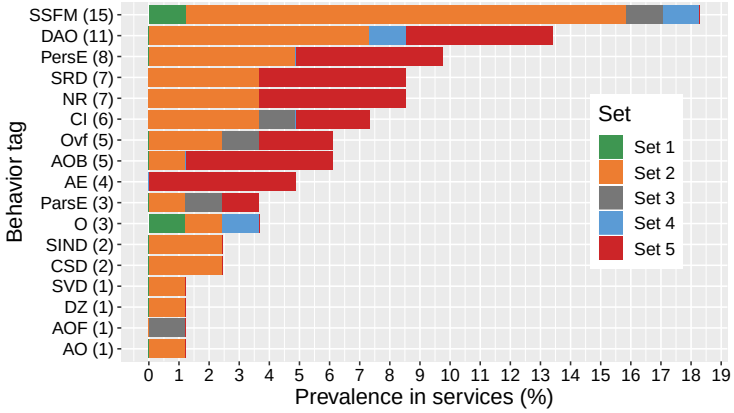


Figure 6.2: Prevalence of the behavior tags in the tested services.

In Figure 6.2 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 5.2 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 6.2 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 [167].

Table 6.1 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 6.1 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 [3]. 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 these are not as relevant as those that represent clear robustness problems.

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 with a `500 internal server error` status code and a JavaScript Object Notation (JSON) payload with `{error: {code: 500, message: null}}`. This was observed in the operations `drive.permissions.list` and `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 6.1) 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*). 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).

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.

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. 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 ser-

Table 6.1: Results of the experimental evaluation for Sets 1 and 2

	API	Operations tested	Status code	Operations with error responses	Parameters		Injected fault types				Behaviour Tags	CRASH
					Total	Affected	Any	Array | Bool | DateTime	Number	String		
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	3 operations	9	3	Any_Empty (2) Any_Null (1)			S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_Malicious (1)	SSFM	A
Set 1	Spotify	36 (97%)	502	3 operations	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	20 (91%)	500	getPresenceOfChannel	6	2	**All** (2)				DAO, SIND	A
Set 2	Ably	20 (91%)	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	14 operations	206	170	**All** (170)	**All Array** (49) All Boolean (47) All Date&Time (4)	N_Add1 (17) N_ReplaceTypeMin (17) N_Sub1 (17) N_ReplaceTypeMax+1 (17) N_Replace1 (17) N_ReplaceTypeMin-1 (17) N_Replace0 (17) N_ReplaceDomainMin (15) N_Replace-1 (17) N_ReplaceDomainMin-1 (15) N_ReplaceTypeMax (17)	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	5 (83%)	404	get search	3	2	Any_Empty (2)				SSFM	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	4 (100%)	500	GET-version-locations-markers-format, GET-version-locations-format	16	2			N_ReplaceNegative (2)		CSD, PersE, DAO, SIND, CI, SVD	A
Set 2	BikeWise	4 (100%)	500	GET-version-incidents-id-format	1	2			N_ReplaceTypeMax+1 (2) N_ReplaceTypeMin-1 (2)		CSD, Ovf, CI, SVD, PersE, DAO, SIND	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 operations	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	6 operations	23	10				S_ReplacePrintable (9) S_AppendNonPrintable (1) S_InsertNonPrintable (1) S_Malicious (9)	SSFM	A
Set 2	Europeana	13 (100%)	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_ReplaceTypeMax (1) N_Sub1 (1) N_ReplaceTypeMin (1) N_Replace0 (1) N_ReplaceTypeMin-1 (1) N_Replace1 (1) N_ReplaceDomainMin (1) N_Replace-1 (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	29 (100%)	400	getMediaByTagId	4	1			N_Replace0 (1)		AO, DZ	A
Set 2	HHS Media Services	29 (100%)	400	getMediaByTagId	4	1			N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)		CI, Ovf, ParsE, NR	A
Set 2	HHS Media Services	29 (100%)	400	getSources, getLanguages, getCampaigns, getMediaByCampaignId, getFeaturedMedia	16	5			N_ReplacePositive (5) N_Replace-1 (5) N_ReplaceNegative (5) N_ReplaceTypeMax (5) N_Add1 (5) N_ReplaceTypeMin (5) N_Sub1 (5) N_ReplaceTypeMax+1 (5) N_Replace0 (5) N_ReplaceTypeMin-1 (5) N_Replace1 (5)		CI, Ovf, ParsE	A
Set 2	HHS Media Services	29 (100%)	400	getMedia	53	1			N_ReplaceTypeMax+1 (1) N_ReplaceTypeMin-1 (1)		CI, Ovf, ParsE	A
Set 2	HHS Media Services	29 (100%)	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	10 operations	9	10	Any_Empty (7)		N_ReplacePositive (3) N_Replace1 (3) N_ReplaceNegative (3) N_Replace-1 (3) N_Add1 (3) N_ReplaceTypeMax (3) N_Sub1 (3) N_ReplaceTypeMin (3) N_Replace0 (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	89 (97%)	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	89 (97%)	500	getTermStatsUsingGET	2	1	Any_Empty (1)			S_ReplacePrintable (1) S_InsertNonPrintable (1) S_ReplaceNonPrintable (1)	SRD, NR	A
Set 2	Rat Genome Database	89 (97%)	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	89 (97%)	500	getAnnotationCountByAccIdAndObjectTypeUsingGET	4	1			N_ReplacePositive (1) N_Replace1 (1) N_ReplaceNegative (1) N_Replace-1 (1) N_Sub1 (1) N_ReplaceTypeMax (1) N_Replace0 (1) N_ReplaceTypeMin (1)		PersE, DAO, SRD, CSD	A
Set 2	Rat Genome Database	89 (97%)	500	getGenesByKeywordUsingGET	2	1				S_Malicious (1)	PersE, DAO, SRD, CSD	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_Replace-1 (1) N_ReplaceNegative (1) N_ReplaceTypeMax (1) N_Add1 (1) N_ReplaceTypeMin (1) N_Sub1 (1) N_ReplaceTypeMax+1 (1) N_Replace0 (1) N_ReplaceTypeMin-1 (1) N_Replace1 (1)		NR	A

vice, 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., as 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., as 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 Hypertext Transfer Protocol (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 services 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 6.1). 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 service 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).

In **Set 3 (Middleware management services)**, all robustness issues observed (during testing of the Docker Engine API) are associated with the `500 internal server error` status code (refer to Table 6.2). 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

Table 6.2: Results of the experimental evaluation for Sets 3-5

	API	Operations tested	Status code	Operations with error responses	Parameters		Injected fault types				Behavior Tags	CRASH
					Total	Affected	Any	Array | Boolean	Number	String		
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, newProducts_Vx0, changePaymentMethod_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)					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)	PersE, DAO	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

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 type 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`).

Regarding **Set 4 (Private services)**, there are a few 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 containing `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 Hypertext Markup Language (HTML) payload. Note that this service 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 escaped the verification activities put in place by the experienced developers in charge of the service.

The tests involving **Set 5 (In-house services)** resulted in the disclosure of several robustness problems, from which we present a few highlights. In ***TPC-App***, calls to the `New products` operation in both Vx0 and VxA implementations, using negative values or the maximum plus one on the `itemLimit` parameter, resulted in an error message stating a call to `setFetchSize` (a function that specifies the number of rows to be fectched from the database) was made using invalid arguments. What is interesting is that this is a function of the query-parsing middleware, and the Oracle JDBC driver used (version 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 [259]). This case highlights 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).

As mentioned before, 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. This was evident at the client side when it received replies such as `ORA-00933: SQL command not properly ended`.

Regarding ***TPC-C***, we observed similar issues to the ones presented earlier (e.g., misrepresentation of datatypes). For instance, in the `Payment` operation of both Vx0 and VxA, a boolean was misrepresented as a 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 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 preceeded 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.

6.3 Main findings

This section highlights the main findings of our experimental evaluation by going through key aspects we identified during the course of this work.

- REST services are being made available online carrying residual bugs that affect the overall robustness of the services.
- Robustness tests proved to be able to disclose bugs that reside at the service implementation level, but also in the middleware that supports the service.
- Robustness tests were able to detect security issues where malicious inputs served to trigger problems 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 known 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 malicious strings (i.e., SQL injection) and faults with random characters 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 Simple Object Access Protocol (SOAP) web services [167], the large number of null/empty value faults that triggered robustness issues did not directly lead to the disclosure of *null reference* problems. Either these triggered other kinds of problems (e.g., *Data Access Operations*), or 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, such as Catastrophic [80], seem to be difficult to trigger.
- Mismatches between the interface description and the actual service implementation were detected during the analysis of the test 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 analyzed 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 test results and creates issues for applications that wish to integrate with these services.
- In almost half of the services tested, we found non-descriptive error messages which, accompanied by a poor specification, do not allow clients to get much insight regarding the real issues.
- Access to server logs was not sufficient for allowing us to understand the exact root cause of the problems detected with the Docker Engine service.
- Robustness testing proved to be 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 Transaction Processing Performance Council (TPC) implementations (although some relate to poor coding 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.

This page is intentionally left blank.

Chapter 7

Threats to validity

In this brief chapter, we present the main threats to the validity of this work and discuss mitigation strategies. We do so in two parts, the first covering the threats related to the identification, analysis and discussion of the systematic review, presented in Chapter 3, and the second referring to the threats to the validity of the approach design and tool implementation presented in Chapter 5, and the design, execution and result analysis of the experimental evaluation carried out in Chapter 6.

We start by covering the **threats to the validity of the systematic review**, where the first case to mention refers to the selection of the works to be included in the review of studies on software robustness testing. In some cases (e.g., a referenced paper) *we had to individually analyze some works and decide about their inclusion* in the review. Although this was mostly carried out by an Early Stage Researcher, all included items were double-checked by an Experienced Researcher, which mostly lead to the elimination of a few out-of-scope works (e.g., research focusing on hardware faults). The Experienced Researcher also verified the steps defined for the review, with particular attention to the identification of any missing paper. This resulted in the inclusion of a few additional works, mostly identified in references or citations to included papers.

One of the main difficulties found during the identification of works is related with the *diversity in the definition of robustness across authors*. This may have lead us to wrongly exclude certain papers (e.g., due to a misleading abstract) as well as to include less relevant papers. Again, we resorted to double-checking by an Experienced Researcher, which allowed us to obtain a better final set of papers. A related aspect is that *some works use different terms to refer to robustness* (e.g., reliability as in [68]). This was detected in the oldest references (i.e., published before the Ballista project [3]) and is very difficult to identify. To mitigate this threat we opted to keep a reduced number of works that are referenced by other authors as being robustness evaluation research (e.g., [68] is referenced by [85]).

As *the whole review process relies on the richness of the data present in the online databases used*, the final set of papers may be incomplete (i.e., if the research is not present or properly indexed by the online databases). To mitigate the effects of this threat, we resorted to a total of six online databases, which are well known and frequently used in this type of process [48, 49, 50]. Also, *the search string used to search the online databases may be weak* leading to either too many results (including irrelevant ones) or too few, adding difficulties to the process. To mitigate this issue, we included different words based on common terms used by authors, which we found in a preliminary analysis of the state of the art.

We must also refer that, due to the complexity of the process, *we opted to not assess the quality of the identified studies* and, as such, the results may be biased towards weak research contributions. We mitigated this threat by indicating important references during the presentation of the research found per type of system. A related aspect is that *the definition of seven groups of systems may lead to works being discussed in an inadequate section* (e.g., because they are at the intersection of two groups). Also, *the human intervention required to place a work in a certain group (or to classify a certain aspect of the work, like the techniques used) may result in errors*. We mitigated these threats by having an Experienced Researcher checking and iteratively identifying inconsistencies and errors, and afterwards by having a second Experienced Researcher reviewing the classification of individual items.

Regarding the **threats to the validity of the approach design, implementation and evaluation**, we start by mentioning the fact that *we manually analyzed 399001 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 from 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 Verification and Validation (V&V) tasks throughout the development of the tool, and we placed it in contact with diverse services (i.e., with different number of operations, parameters and payload requirements) so that any issue would become evident. In the end, when analyzing 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*, which included the number of faults to generate for a certain parameter (some faults involve random value generation) and 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 actually 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 the developers with useful information for robustness assessment, which can be used by providers to improve their services.

This page is intentionally left blank.

Chapter 8

Conclusions and future work

In the modern world, software systems are present in virtually all types of businesses, including business-critical domains such as banking, or safety-critical domains such as aerospace and healthcare, just to name a few. In these particular domains, software is expected to deliver continuous service in a robust manner when facing unexpected conditions, as a single failure may strongly affect the users or the software provider. RESTful web services (i.e., those that implement the Representational State Transfer (REST) architectural style), in particular, are nowadays used extensively to facilitate machine-to-machine communication in the most varied scenarios. Companies like Google, Instagram, Spotify, or Slack use RESTful services to support their main business infrastructure. This type of system is especially exposed to robustness issues (e.g., lack of or weak input verification) given its design with relaxed constraints and lack of standards.

In this dissertation, we present a systematic literature review on software robustness testing, and propose a novel approach and a tool for carrying out robustness tests on RESTful web services. We began by thoroughly analyzing the extensive state of the art of software robustness testing, and identified many gaps in the existing research, such as the absence of support for some types of systems, including blockchain systems and, with particular emphasis, REST services. We also explored the state of the art on REST Application Programming Interface (API) testing, and experimented with industry tools for REST API testing, which led us to confirm the overall lack of support for robustness testing of REST services both from the academic and the industry perspectives.

Based on the analyzed state of the art, we designed an approach for testing the robustness of RESTful web services, and implemented it in the form of a tool named black-BOX tool for Robustness Testing of rest services (bBOXRT). We introduced the concepts behind our approach and described the architecture of bBOXRT, mapping its many components to the different phases of a typical robustness testing campaign. We then showcased the capabilities of bBOXRT in the form of a practical experiment by performing thousands of tests over 52 REST services grouped in 5 different sets. Results showed the ability of the tool to test different kinds of services and to disclose robustness as well as security issues across nearly half of the tested services.

As future work, we intend to improve upon the extensibility of bBOXRT, so as to facilitate the implementation of new functionality. Additionally, we aim to add support for a graphical user interface, which would greatly improve the usability of bBOXRT, as well as simplify the configuration process for new REST APIs to test. Finally, we will also consider studying the applicability of Machine Learning algorithms to our approach, which would be quite useful in automating the process of test result classification.

This page is intentionally left blank.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008.

[2] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[3] Philip Koopman, John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. Comparing Operating Systems Using Robustness Benchmarks. In *SRDS*, 1997.

[4] Fares Saad-Khorchef, Antoine Rollet, and Richard Castanet. A Framework and a Tool for Robustness Testing of Communicating Software. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC, pages 1461–1466, New York, NY, USA, 2007. ACM.

[5] Carlos Leandro Gomes Batista, Anderson Coelho Weller, Eliane Martins, and Fátima Mattiello-Francisco. Towards increasing nanosatellite subsystem robustness. *Acta Astronautica*, 156:187 – 196, 2019.

[6] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. Experimental Robustness Evaluation of JMS Middleware. In *IEEE International Conference on Services Computing*, volume 1, pages 119–126, July 2008.

[7] Anup K. Ghosh and Matthew Schmid. An approach to testing COTS software for robustness to operating system exceptions and errors. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, pages 166–174, Nov 1999.

[8] Chen Fu Ana Milanova, Barbara G. Ryder, and David Wonnacott. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, 31, 2004.

[9] Reda Siblini and Nashat Mansour. Testing Web services. In *Book of Abstracts. ACS/IEEE International Conference on Computer Systems and Applications*, page 135, 02 2005.

[10] Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness Evaluation of Controllers in Self-Adaptive Software Systems. In *Sixth Latin-American Symposium on Dependable Computing*, pages 1–10, April 2013.

[11] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019.

[12] Postdot Technologies. Postman. `https://www.getpostman.com/`, 2014. Accessed 26 March 2019.

[13] Capers Jones. *The Technical and Social History of Software Engineering*. Addison-Wesley, 2013.

[14] Edward E. Ogheneovo. Software Dysfunction: Why Do Software Fail? *Journal of Computational Chemistry*, 2014:25–35, 2014.

[15] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[16] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[17] SmartBear Software. OpenAPI Specification Version 3.0.3. `https://swagger.io/specification/`, 2020. Accessed 20 February 2020.

[18] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical Report IEEE Std 610.12-1990, IEEE, 1990.

[19] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS, pages 30–, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Rauli Kaksonen, Marko Laakso, and Ari Takanen. *Software Security Assessment through Specification Mutations and Fault Injection*, pages 173–183. Springer US, Boston, MA, 2001.

[21] Yulong Fu and Ousmane Koné. Security and Robustness by Protocol Testing. *IEEE Systems Journal*, 8(3):699–707, Sep. 2014.

[22] Christopher P. Dingman, Joe Marshall, and Daniel P. Siewiorek. Measuring robustness of a fault tolerant aerospace system. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 522–527, June 1995.

[23] Hoang-Nam Chu, Jean Arlat, Marc-Olivier Killijian, Benjamin Lussier, and David Powell. Robustness Testing of Robot Controller Software. In Hélène Waeselynck, editor, *12th European Workshop on Dependable Computing, EWDC*, page 2 pages, Toulouse, France, May 2009.

[24] Fátima Mattiello-Francisco, Eliane Martins, Ana Rosa Cavalli, and Edgar Toshiro Yano. InRob: An approach for testing interoperability and robustness of real-time embedded software. *Journal of Systems and Software*, 85:3–15, 01 2012.

[25] Zoltán Micskei, István Majzik, and Francis Tam. Robustness testing techniques for high availability middleware solutions. In *Proceedings of International Workshop on Engineering of Fault Tolerant Systems (EFTS)*, 2006.

[26] Denise Azevedo, Ana Ambrosio, and Marco Vieira. HLA Middleware Robustness and Scalability Evaluation in the Context of Satellite Simulators. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 312–317. IEEE, IEEE Press, 12 2013.

[27] Wallace Cardoso, Eliane Martins, Nuno Laranjeiro, and Nuno Antunes. Combining State and Interface - Based Robustness Testing for OpenStack Components. In *9th Latin-American Symposium on Dependable Computing (LADC)*, pages 1–10. IEEE Press, November 2019.

[28] Naaliel Mendes, João Durães, and Henrique Madeira. Evaluating and Comparing the Impact of Software Faults on Web Servers. In *European Dependable Computing Conference*, pages 33–42, April 2010.

[29] Seung Hak Kuk and Hyeon Soo Kim. Robustness testing framework for Web services composition. In *IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 319–324, Dec 2009.

[30] Sébastien Salva and Issam Rabhi. Stateful Web Service Robustness. In *Fifth International Conference on Internet and Web Applications and Services*, pages 167–173, May 2010.

[31] Sylvia Ilieva, Denitsa Manova, Ilina Manova, Cesare Bartolini, Antonia Bertolino, and Francesca Lonetti. An automated approach to robustness testing of BPEL orchestrations. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 193–203, Dec 2011.

[32] João Agnelo and Nuno Laranjeiro and Jorge Bernardino. black-BOX tool for Robustness Testing of rest services (bBOXRT). https://git.dei.uc.pt/jagnelo/bBOXRT, 2019.

[33] Leonard Richardson. *RESTful web services*. O'Reilly, Farnham, 2007.

[34] Leonard Richardson. *RESTful Web APIs*. O'Reilly, Sebastopol, Calif, 2013.

[35] OASIS OData Technical Committee. Open Data Protocol specification version 4.01. `https://www.odata.org/documentation/`, 2020. Accessed 20 February 2020.

[36] Jonathan Robie, Rob Cavicchio, Rémon Sinnema, and Erik Wilde. RESTful Service Description Language (RSDL): Describing RESTful Services Without Tight Coupling. In *Balisage: The Markup Conference 2013*, volume 10, Aug 2013.

[37] RAML Workgroup. RESTful API Modeling Language specification version 1.0. `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/`. Accessed 20 February 2020.

[38] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999. `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[39] William C. Hetzel. *The Complete Guide to Software Testing*. QED Inforíation Sciences 1988., 1988.

[40] Shivani Anil Acharya and Vidhi Pandya. Bridge between Black Box and White Box - Gray Box Testing Technique. In *International Journal of Electronics and Computer Science Engineering*, 1956.

[41] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. Robustness testing techniques and tools. In *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer, 2012.

[42] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[43] Barbara Kitchenham. Procedures for Performing Systematic Reviews. *Keele, UK, Keele Univ.*, 33, 08 2004.

[44] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571 – 583, 2007. Software Performance.

[45] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE, pages 1–10, London, England, United Kingdom, 2014. Association for Computing Machinery.

[46] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1 – 17, 2013.

[47] Syed Muhammad Ali Shah, Daniel Sundmark, Birgitta Lindström, and Sten F. Andler. Robustness Testing of Embedded Software Systems: An Industrial Interview Study. *IEEE Access*, 4:1859–1871, 2016.

[48] Aurora Ramírez, José Raúl Romero, and Christopher L. Simons. A Systematic Review of Interaction in Search-Based Software Engineering. *IEEE Transactions on Software Engineering*, 45(8):760–781, 2019.

[49] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A Systematic Review on Cloud Testing. *ACM Comput. Surv.*, 52(5), September 2019.

[50] Jennifer Brings, Marian Daun, Kevin Keller, Patricia Aluko Obe, and Thorsten Weyer. A systematic map on verification and validation of emergent behavior in software engineering research. *Future Generation Computer Systems*, 112:1010 – 1037, 2020.

[51] Google. Google Scholar. https://scholar.google.com/, 2020.

[52] DBLP Team. DBLP. https://dblp.uni-trier.de/, 2020.

[53] IEEE. IEEE Xplore. https://ieeexplore.ieee.org/Xplore/home.jsp, 2020.

[54] Association for Computing Machinery. ACM Digital Library. https://dl.acm.org/, 2020.

[55] Elsevier BV. Scopus. https://www.scopus.com/home.uri, 2020.

[56] Springer Nature Switzerland AG. Springer Link. https://link.springer.com/, 2020.

[57] Mathieu Lavallée, Pierre-N. Robillard, and Reza Mirsalari. Performing Systematic Literature Reviews With Novices: An Iterative Approach. *IEEE Transactions on Education*, 57(3):175–181, 2014.

[58] Miroslav Popovic. *Communication Protocol Engineering.* CRC Press, Inc., USA, 2006.

[59] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools.* O'Reilly Media, Inc., 2 edition, 2006.

[60] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach.* The MIT Press, 2nd edition, 2016.

[61] Steve Vinoski. Where is middleware. *IEEE Internet Computing*, 6(2):83–85, 2002.

[62] N Walters. Systems Architecture and COTS Integration. In *Procs of the SEI/MCC Symposium on the Use of COTS in Systems Integration, SEI Special Report CMU/SEI-95-SR-007*, 1995.

[63] Mark Vigder, W Morven Gentleman, and John C Dean. *COTS Software Integration: State of the art*, volume 39190. National Research Council Canada, Institute for Information Technology, 1996.

[64] Aaron E. Walsh. *UDDI, SOAP, and WSDL: The Web Services Specification Reference Book*. Prentice Hall Professional Technical Reference, 2002.

[65] Douglas K. Barry. *Web Services, Service-Oriented Architectures, and Cloud Computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[66] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), May 2009.

[67] Harvey M. Deitel, Paul J. Deitel, and David R. Choffnes. *Operating Systems (3rd Edition)*. Prentice-Hall, Inc., USA, 2003.

[68] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[69] Charles P. Shelton, Philip Koopman, and Kobey DeVale. Robustness Testing of the Microsoft Win32 API. In *Proceeding International Conference on Dependable Systems and Networks*, pages 261 – 270, 02 2000.

[70] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. Automatic Fault Injection for Driver Robustness Testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA, pages 361–372, New York, NY, USA, 2015. ACM.

[71] Alejandro David Velasco, Bartolomeo Montrucchio, and Maurizio Rebaudengo. KITO tool: A fault injection environment in Linux kernel data structures. *Microelectronics Reliability*, 60:153 – 162, 2016.

[72] Huan Feng and Kang G. Shin. Bindercracker: Assessing the robustness of android system services. *Computing Research Repository*, abs/1604.06964, 2016.

[73] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of Inter-component Communication in Android. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2012.

[74] Raimondas Sasnauskas and John Regehr. Intent Fuzzer: Crafting Intents of Death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA, page 1–5, New York, NY, USA, 2014. Association for Computing Machinery.

[75] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, page 68–74, New York, NY, USA, 2013. Association for Computing Machinery.

[76] Mithun Acharya, Tao Xie, and Jun Xu. Mining Interface Specifications for Generating Checkable Robustness Properties. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, page 311–320, USA, 2006. IEEE Computer Society.

[77] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *International Conference on Dependable Systems and Networks*, pages 867–876, United States, June 2004. IEEE Press.

[78] Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina, and Philippe Rumeau. Benchmarking the dependability of Windows and Linux using PostMark/spl trade/ workloads. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 10 pp.–20, Nov 2005.

[79] Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, Sep. 2000.

[80] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, FTCS, pages 230–, Washington, DC, USA, 1998. IEEE Computer Society.

[81] Barton P. Miller, David Koski, Cjin Pheow, Lee Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, Department of Computer Sciences, 1995.

[82] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, page 46–54, New York, NY, USA, 2006. Association for Computing Machinery.

[83] Bartolomeo Montrucchio and Maurizio Rebaudengo and Alejandrand David Velasco. Software-implemented fault injection in operating system kernel mutex data structure. In *IEEE 5th Latin American Symposium on Circuits and Systems*, pages 1–6, 2014.

[84] Bartolomeo Montrucchio and Maurizio Rebaudengo and Alejandrand David Velasco. Fault injection in the process descriptor of a Unix-based operating system. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 281–286, 2014.

[85] Daniel P. Siewiorek, John J. Hudak, Byung-Hoon Suh, and Zary Segall. Development of a benchmark to measure system robustness. *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 88–97, 1993.

[86] Byung-Hoon Suh, John Hudak, Daniel Siewiorek, and Zary Segall. Development of a benchmark to measure system robustness: experiences and lessons learned. In *Proceedings Third International Symposium on Software Reliability Engineering*, pages 237–245, 1992.

[87] Lin Xiang, Zhan Zhang, Decheng Zuo, and Xiaozong Yang. Multi-Layered System Robustness Testing Strategy Based on Abnormal Parameter. *Journal of Computers*, 8, 2013.

[88] João Durães and Henrique Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *Pacific Rim International Symposium on Dependable Computing*, pages 201–209, Dec 2002.

[89] Justin E. Forrester and Barton P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[90] Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the Impact of Injection Triggers for OS Robustness Evaluation. In *The 18th IEEE International Symposium on Software Reliability (ISSRE)*, pages 127–126, 2007.

[91] Manuel Mendonça and Nuno Ferreira Neves. Robustness Testing of the Windows DDK. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 554–564, 2007.

[92] Anup K. Ghosh, Matt Schmid, and Viren Shah. Testing the robustness of Windows NT software. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 231–235, Nov 1998.

[93] Matthew Schmid, Anup K. Ghosh, and Frank Hill. Techniques for evaluating the robustness of Windows NT software. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX*, volume 2, pages 347–360 vol.2, Jan 2000.

[94] Theodore John Socolofsky and Claudia Jeanne Kale. TCP/IP tutorial. RFC 1180, RFC Editor, January 1991. `http://www.rfc-editor.org/rfc/rfc1180.txt`.

[95] Ana Cavalli, Eliane Martins, and Anderson Morais. Use of invariant properties to evaluate the results of fault-injection-based robustness testing of protocol implementations. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 21–30. IEEE Press, April 2008.

[96] Maha Naceur, Lilia Sfaxi, and Riadh Robbana. Robustness testing of secure Wireless Sensor Networks. In *Proceedings of the International Conference on Automation, Control, Engineering and Computer Science, ACECS*, 2014.

[97] Miroslav Popovic and Jelena Kovacevic. A Statistical Approach to Model-Based Robustness Testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 485–494, March 2007.

[98] Antoine Rollet and Salva. Testing robustness of communicating systems using ioco-based approach. In *IEEE Symposium on Computers and Communications*, pages 67–72, July 2009.

[99] Arunchandar Vasan and A.M. Memon. ASPIRE: automated systematic protocol implementation robustness evaluation. In *Australian Software Engineering Conference. Proceedings.*, pages 241–250, April 2004.

[100] Yang Xiang, Zhiliang Wang, and Xia Yin. SIP Robustness Testing Based on TTCN-3. In *International Conference on Advanced Information Networking and Applications Workshops*, pages 122–128, May 2009.

[101] Luo Xu, Ji Wu, and Chao Liu. TTCN-3 Based Robustness Test Generation and Automation. In *International Conference on Information Technology and Computer Science*, volume 2, pages 120–125, July 2009.

[102] Chuanming Jing, Zhiliang Wang, Xia Yin, and Jianping Wu. A Formal Approach to Robustness Testing of Network Protocol. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, pages 24–37, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[103] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, and Vincenzo Gulisano. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332, March 2014.

[104] János Zoltán Szabó and Tibor Csöndes. TITAN, TTCN-3 test execution environment. *Infocommunications Journal*, 62(1):27–31, 2007.

[105] Yamine Aït-Ameur, Gamze Bel, Frédéric Boniol, S. Pairault, and Virginie Wiels. Robustness Analysis of Avionics Embedded Systems. *SIGPLAN Not.*, 38(7):123–132, June 2003.

[106] Shunkun Yang, Bin Liu, Shihai Wang, and Minyan Lu. Model-based robustness testing for avionics-embedded software. *Chinese Journal of Aeronautics*, 26(3):730 – 740, 2013.

[107] Khaled Alnawasreh, Patrizio Pelliccione, Zhenxiao Hao, Mårten Rånge, and Antonia Bertolino. Online Robustness Testing of Distributed Embedded Systems: An Industrial Approach. In *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 133–142. IEEE Press, May 2017.

[108] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frederic Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, 2002.

[109] Antoine Rollet and Sebastien Salva. Two complementary approaches to test robustness of reactive systems. In *IEEE International Conference on Automation, Quality and Testing, Robotics*, volume 1, pages 47–53, May 2008.

[110] Hacène Fouchal, Antoine Rollet, and Abbas Tarhini. Robustness of Composed Timed Systems. In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science*, pages 157–166, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[111] Hacène Fouchal, Antoine Rollet, and Abbas Tarhini. Robustness Testing of Composed Real-Time Systems. *Journal of Computational Methods in Sciences and Engineering*, 10(1-2S2):135–148, September 2010.

[112] Fatima Mattiello-Francisco, Eliane Martins, Andre Corsetti, Ana Rosa Cavalli, and Edgar Yano. Extended interoperability models for timed system robustness testing. In *IEEE Latin-American Conference on Communications*, pages 1–6, 2009.

[113] Antoine Rollet and Fares Saad-Khorchef. A Formal Approach to Test the Robustness of Embedded Systems using Behaviour Analysis. In *5th ACIS International Conference on Software Engineering Research, Management Applications (SERA)*, pages 667–674, 2007.

[114] Abbas Tarhini, Antoine Rollet, and Hacène Fouchal. A pragmatic approach for testing robustness on real-time component based systems. In *The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005.*, pages 143–150, Jan 2005.

[115] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. GRINDER: On Reusability of Fault Injection Tools. In *IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 75–79, May 2015.

[116] Domenico Cotroneo, Domenico Di Leo, Roberto Natella, and Roberto Pietrantuono. A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security*, pages 213–227, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[117] Domenico Cotroneo, Domenico Di Leo, Francesco Fucci, and Roberto Natella. SABRINE: State-based robustness testing of operating systems. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 125–135. IEEE, Nov 2013.

[118] Henrique Madeira, Raphael R. Some, Francisco Moreira, Diamantino Costa, and David A. Rennels. Experimental evaluation of a COTS system for space applications. In *Proceedings International Conference on Dependable Systems and Networks*, pages 325–330, 2002.

[119] Ricardo Maia, Luis Henriques, Ricardo Barbosa, Diamantino Costa, and Henrique Madeira. Xception fault injection and robustness testing framework: a case-study of testing RTEMS. In *VI Test and Fault Tolerance Workshop (jointly organized with the 23rd Brazilian Symposium on Computer Networks (SBRC))*, 2005.

[120] Bogdan Nicolescu, N. Ignat, Yvon Savaria, and Gabriela Nicolescu. Sensitivity of Real-Time Operating Systems to Transient Faults: A case study for MicroC kernel. In *8th European Conference on Radiation and Its Effects on Components and Systems*, Sep. 2005.

[121] Manuel Rodríguez, Arnaud Albinet, and Jean Arlat. MAFALDA-RT: a tool for dependability assessment of real-time systems. In *Proceedings International Conference on Dependable Systems and Networks*, pages 267–272, June 2002.

[122] Juan Carlos Ruiz, José Carlos Campelo, Pedro Gil-Vicente, and Juan Pardo Albiach. On-chip debugging-based fault emulation for robustness evaluation of embedded software components. In *11th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 8 pp.–, 2005.

[123] Raheleh Shahpasand, Yasser Sedaghat, and Samad Paydar. Improving the stateful robustness testing of embedded real-time operating systems. In *6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 159–164, Oct 2016.

[124] Raheleh Shahpasand, Samad Paydar, Yasser Sedaghat, and Reza Ramezani. A State-aware Approach for Robustness Testing of Embedded Real-Time Operating Systems. *Journal of Computer and Knowledge Engineering*, 2(1), 2019.

[125] Zhengmao Zhou, Yun Zhou, Ming Cai, and Lei Sun. A Workload Model Based Approach to Evaluate the Robustness of Real-time Operating System. In *IEEE 10th International Conference on High Performance Computing and Communications*, pages 2027–2033, Nov 2013.

[126] João Carreira, Henrique Madeira, and João G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb 1998.

[127] David Powell, Jean Arlat, Hoang Nam Chu, Felix Ingrand, and Marc-Olivier Killijian. Testing the Input Timing Robustness of Real-Time Control Software for Autonomous Systems. In *Ninth European Dependable Computing Conference*, pages 73–83, May 2012.

[128] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. Robustness Testing of Autonomy Software. In *IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 276–285, May 2017.

[129] Wallace F. F. Cardoso and Eliane Martins. Using a Search and Model Based Framework to Improve Robustness Tests in Cloud Platforms. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, SAST '18, page 67–76, New York, NY, USA, 2018. Association for Computing Machinery.

[130] Franck Chauvel, Hui Song, Nicolas Ferry, and Franck Fleurey. Evaluating robustness of cloud-based systems. *Journal of Cloud Computing*, 4(1):18, 2015.

[131] Domenico Cotroneo, Flavio Frattini, Roberto Pietrantuono, and Stefano Russo. State-based robustness testing of IaaS cloud platforms. In *CloudDP '15: Proceedings of the 5th International Workshop on Cloud Data and Platforms*, 2015.

[132] Jiantao Pan, Philip Koopman, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness testing and hardening of CORBA ORB implementations. In *International Conference on Dependable Systems and Networks*, pages 141–150, July 2001.

[133] Kimberly Fernsler and Philip John Koopman. Robustness testing of a distributed simulation backplane. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, pages 189–198, Nov 1999.

[134] Andras Kövi and Zoltan Micskei. Robustness Testing of Standard Specifications-Based HA Middleware. In *IEEE 30th International Conference on Distributed Computing Systems Workshops*, pages 302–306, June 2010.

[135] Zoltán Micskei, István Majzik, and Francis Tam. Comparing Robustness of AIS-Based Middleware Implementations. In Miroslaw Malek, Manfred Reitenspieß, and Aad van Moorsel, editors, *Service Availability*, pages 20–30, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[136] Aniello Napolitano, Gabriella Carrozza, Antonio Bovenzi, and Christian Esposito. Automatic Robustness Assessment of DDS-Compliant Middleware. In *IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 302–307, Dec 2011.

[137] Hussein Al-haj Ahmad, Yasser Sedaghat, and Mahin Moradiyan. LDSFI: a Lightweight Dynamic Software-based Fault Injection. In *9th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 207–213. IEEE Press, Oct 2019.

[138] Fevzi Belli, Axel Hollmann, and Weicheneric Wong. Towards Scalable Robustness Testing. In *Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 208–216. IEEE Press, June 2010.

[139] Christof Fetzer and Zhen Xiao. Automatic Testing for Robustness Violations. *Testing Commercial-off-the-Shelf Components and Systems*, 01 2005.

[140] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 31–40, Dec 2013.

[141] Ang Jin, Jianhui Jiang, Jiawei Hu, and Jungang Lou. A PIN-Based Dynamic Software Fault Injection System. In *The 9th International Conference for Young Computer Scientists*, pages 2160–2167, Nov 2008.

[142] Pan Qing-He, Hong Rong, and Pan Shu. A Software-Implemented Fault Injector on Windows NT Platform. *Information Technology Journal*, 9, 03 2010.

[143] Ali Shahrokni and Robert Feldt. RobusTest: A Framework for Automated Testing of Software Robustness. In *18th Asia-Pacific Software Engineering Conference*, pages 171–178, Dec 2011.

[144] Ekincan Ufuktepe and Tugkan Tuglular. Estimating software robustness in relation to input validation vulnerabilities using Bayesian networks. *Software Quality Journal*, 26(2):455–489, 2018.

[145] Kamal Z. Zamli, Mohd Daud Alang Hassan, Nor Ashidi Mat Isa, and Siti Norbaya Azizan. An automated software fault injection tool for robustness assessment of java COTs. In *International Conference on Computing Informatics*, pages 1–6, June 2006.

[146] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software Practice and Experience*, 34:1025–1050, 09 2004.

[147] João Durães and Henrique Madeira. Emulation of software faults by educated mutations at machine-code level. In *13th International Symposium on Software Reliability Engineering*, pages 329–340. IEEE Press, 2002.

[148] João Durães and Henrique Madeira. Definition of software fault emulation operators: a field data study. In *International Conference on Dependable Systems and Networks*, pages 105–114, June 2003.

[149] João Durães and Henrique Madeira. Generic faultloads based on software faults for dependability benchmarking. In *International Conference on Dependable Systems and Networks*, pages 285–294, June 2004.

[150] João Durães and Henrique Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, Nov 2006.

[151] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A Model-Based Approach for Robustness Testing. In Ferhat Khendek and Rachida Dssouli, editors, *Testing of Communicating Systems*, pages 333–348, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[152] Bin Lei, Xuandong Li, Zhiming Liu, Charles Morisset, and Volker Stolz. Robustness testing for software components. *Science of Computer Programming*, 75(10):879 – 897, 2010.

[153] Bin Lei, Zhiming Liu, Charles Morisset, and Xuandong Li. State Based Robustness Testing for Components. *Electron. Notes Theor. Comput. Sci.*, 260:173–188, 2010.

[154] Regina Moraes, Ricardo Barbosa, João Durães, Naaliel Mendes, Eliane Martins, and Henrique Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? In *Sixth European Dependable Computing Conference*, pages 53–64, 2006.

[155] János Oláh and István Majzik. A Model Based Framework for Specifying and Executing Fault Injection Experiments. In *Proceedings of the 2009 Fourth International Conference on Dependability of Computer Systems*, DEPCOS-RELCOMEX '09, page 107–114, USA, 2009. IEEE Computer Society.

[156] Diamantino Costa and Henrique Madeira. Experimental Assessment of COTS DBMS Robustness under Transient Faults. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, PRDC '99, page 201, USA, 1999. IEEE Computer Society.

[157] Diamantino Costa, Tiago Rilho, and Henrique Madeira. Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (Formerly FTCS-30 and DCCA-8)*, DSN '00, page 251–260, USA, 2000. IEEE Computer Society.

[158] Sebastian Bauersfeld and Tanja E. J. Vos. GUITest: a Java library for fully automated GUI robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 330–333. IEEE Press, Sep. 2012.

[159] Patrick Heckeler, Bastian Schlich, and Thomas Kropf. Accelerated Robustness Testing of State-Based Components Using Reverse Execution. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1188–1195, New York, NY, USA, 2013. Association for Computing Machinery.

[160] Eliane Martins, Cecília M. F. Rubira, and Nelson G. M. Leme. Jaca: a reflective fault injection tool based on patterns. In *Proceedings International Conference on Dependable Systems and Networks*, pages 483–487, 2002.

[161] Leon Shklar and Rich Rosen. *Web Application Architecture: Principles, Protocols and Practices*. Wiley, Chichester, UK, 2 edition, 2009.

[162] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[163] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007.

[164] Gabriella Carrozza, Aniello Napolitano, Nuno Laranjeiro, and Marco Vieira. WS-RTesting: Hands-On Solution to Improve Web Services Robustness Testing. *Dependable Computing Workshops, Latin-American Symposium on*, 0:41–46, 04 2011.

[165] Samer Hanna and Malcolm Munro. *An Approach for WSDL-Based Automated Robustness Testing of Web Services*, pages 1093–1104. Springer US, Boston, MA, 2009.

[166] Nuno Laranjeiro, Salvador Canelas, and Marco Vieira. Wsrbench: An On-Line Tool for Robustness Benchmarking. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, SCC '08, page 187–194, USA, 2008. IEEE Computer Society.

[167] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. A robustness testing approach for SOAP Web services. *Journal of Internet Services and Applications*, 3(2):215–232, Sep 2012.

[168] Nicholas Looker, Malcolm Munro, and Jiudong xu. Simulating errors in web services. *J. of SIMULATION*, 5:1473–8031, 12 2004.

[169] Nik Looker, Malcolm Munro, and Jie Xu. WS-FIT: a tool for dependability analysis of Web services. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 120–123 vol.2, 2004.

[170] Nik Looker, Malcolm Munro, and Jie Xu. A comparison of network level fault injection with code insertion. In *29th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 479–484 Vol. 2, July 2005.

[171] Evan Martin, Suranjana Basu, and Tao Xie. Automated Testing and Response Analysis of Web Services. In *IEEE International Conference on Web Services (ICWS)*, pages 647–654, July 2007.

[172] Evan Martin, Suranjana Basu, and Tao Xie. WebSob: A Tool for Robustness Testing of Web Services. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION, pages 65–66, Washington, DC, USA, 2007. IEEE Computer Society.

[173] Issam Rabhi. Robustness Testing of Web Services Composition. In *IEEE 14th International Conference on High Performance Computing and Communication*, pages 631–638, June 2012.

[174] Marek Rychlý and Martin Žouželka. Fault Injection for Web-services. In *Proceedings of the 14th International Conference on Enterprise Information Systems - Volume 2: NTMIST, (ICEIS)*, pages 377–383. INSTICC, SciTePress, 2012.

[175] Marcelo Invert Palma Salas, Paulo Lício De Geus, and Eliane Martins. Security Testing Methodology for Evaluation of Web Services Robustness - Case: XML Injection. In *IEEE World Congress on Services*, pages 303–310, June 2015.

[176] Sébastien Salva and Issam Rabhi. Automatic web service robustness testing from WSDL descriptions. *In 12th European Workshop on Dependable Computing, Toulouse, France*, 05 2009.

[177] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Assessing Robustness of Web-Services Infrastructures. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 131–136, June 2007.

[178] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Benchmarking the Robustness of Web Services. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 322–329, 2007.

[179] Ilaria Canova Calori, Tor Stålhane, and Sven Ziemer. Robustness analysis using fmea and bbn: Case study for a web-based application. *Webist 2007 - 3rd International Conference on Web Information Systems and Technologies, Proceedings*, 01 2007.

[180] Karthik Pattabiraman and Benjamin Zorn. DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing. In *IEEE 21st International Symposium on Software Reliability Engineering*, pages 191 – 200, 12 2010.

[181] Jie Xu Nik Looker. Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection. In *The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 163–163, Oct 2003.

[182] Apache Software Foundation. Apache Axis. https://axis.apache.org/, 2002. Accessed 28 June 2020.

[183] Apache Software Foundation. Apache Tomcat. https://tomcat.apache.org/, 1999. Accessed 28 June 2020.

[184] IBM. An Architectural Blueprint for Autonomic Computing. Technical report, IBM, June 2005.

[185] Deborah S. Katz, Casidhe Hutchison, Milda Zizyte, and Claire Le Goues. Detecting Execution Anomalies As an Oracle for Autonomy Software Robustness. *International Conference on Robotics and Automation (ICRA 2020)*, page to appear, 2020.

[186] Mohamed N. Bennani and Daniel A. Menascé. Assessing the robustness of self-managing computer systems under highly variable workloads. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 62–69. IEEE Press, May 2004.

[187] Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Testing the robustness of controllers for self-adaptive systems. *Journal of the Brazilian Computer Society*, 20, 12 2014.

[188] Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness Evaluation of the Rainbow Framework for Self-Adaptation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 376–383, New York, NY, USA, 2014. Association for Computing Machinery.

[189] Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness-Driven Resilience Evaluation of Self-Adaptive Software Systems. *IEEE Transactions on Dependable and Secure Computing*, 14(1):50–64, 2017.

[190] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, and Christian Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[191] Keshav Bimbraw. Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology. *ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings*, 1:191–198, 01 2015.

[192] Fattoh Alqershi, Muhammad AL-Qurishi, Sk Md Mizanur Rahman, and Atif Alamri. Android vs. iOS: The security battle. In *World Congress on Computer Applications and Information Systems (WCCAIS)*, pages 1–8, 2014.

[193] Ahmet Hayran, Muratcan Igdeli, Atif Yilmaz, and Cemal Gemci. Security Evaluation of IOS and Android. *International Journal of Applied Mathematics, Electronics and Computers*, pages 258–261, 2016.

[194] Sijun Chu and Hao Wu. Research on offense and defense technology for iOS kernel security mechanism. *AIP Conference Proceedings*, 1955(1):040132, 2018.

[195] Apache Software Foundation. Apache Kafka. https://kafka.apache.org/, 2011. Accessed 16 July 2020.

[196] Amazon Web Services. Amazon Kinesis. https://aws.amazon.com/pt/kinesis/, 2013. Accessed 16 July 2020.

[197] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[198] Yang Lu. The blockchain: State-of-the-art and research challenges. *Journal of Industrial Information Integration*, 15:80 – 90, 2019.

[199] Arup Mukherjee and Daniel P. Siewiorek. Measuring Software Dependability by Robustness Benchmarking. *IEEE Transactions on Software Engineering*, 23(6):366–378, June 1997.

[200] Karl Wüst and Arthur Gervais. Do you Need a Blockchain? In *Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54, 2018.

[201] Nuno Laranjeiro, Camilo Gomez, Enrico Schiavone, Leonardo Montecchi, Manoel J. M. Carvalho, Paolo Lollini, and Zoltán Micskei. Addressing Verification and Validation Challenges in Future Cyber-Physical Systems. In *9th Latin-American Symposium on Dependable Computing (LADC)*, pages 1–2, 2019.

[202] Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson, and Ghaith Rabadi. System of Systems Engineering. *Engineering Management Journal*, 15(3):36–45, 2003.

[203] Bruno Areias, Nuno Humberto, Lucas Guardalben, JosÃ© Maria Fernandes, and Susana Sargento. Towards an Automated Flying Drones Platform. In *Proceedings of the 4th International Conference on Vehicle Technology and Intelligent Transport Systems - Volume 1: VEHITS,*, pages 529–536. INSTICC, SciTePress, 2018.

[204] Tim Menzies and Charles Pecheur. Verification and Validation and Artificial Intelligence. *Advances in Computers*, 65:153–201, 12 2005.

[205] Andrea Arcuri. RESTful API Automated Test Case Generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20, 2017.

[206] Andrea Arcuri. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397, 04 2018.

[207] Andrea Arcuri. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1):3:1–3:37, January 2019.

[208] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397, 2020.

[209] Sujit Kumar Chakrabarti and Prashant Kumar. Test-the-REST: An Approach to Testing RESTful Web-Services. *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 302–308, 2009.

[210] Sujit Kumar Chakrabarti and Reswin Rodriquez. Connectedness Testing of RESTful Web-services. In *Proceedings of the 3rd India Software Engineering Conference*, ISEC, pages 143–152, New York, NY, USA, 2010. ACM.

[211] Hamza Ed-douibi, Javier Canovas Izquierdo, and Jordi Cabot. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–190, 10 2018.

[212] Tobias Fertig and Peter Braun. Model-Driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 1497–1502, New York, NY, USA, 2015. Association for Computing Machinery.

[213] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential Regression Testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 312–323, New York, NY, USA, 2020. Association for Computing Machinery.

[214] Stefan Karlsson, A. Causevic, and Daniel Sundmark. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, 2020.

[215] Jing Liu and Wenjie Chen. Optimized Test Data Generation for RESTful Web Service. In *24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 683–688, Dec 2017.

[216] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. Model-based testing of RESTful web services using UML protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*, pages 1–10. Citeseer, 2013.

[217] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic Testing of RESTful Web APIs. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE, pages 882–882, New York, NY, USA, 2018. ACM.

[218] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. Towards Property-Based Testing of RESTful Web Services. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, page 77–78, New York, NY, USA, 2013. Association for Computing Machinery.

[219] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. Resource-Based Test Case Generation for RESTful Web Services. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1426–1434, New York, NY, USA, 2019. Association for Computing Machinery.

[220] John Hughes. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL, pages 1–32, Berlin, Heidelberg, 2007. Springer-Verlag.

[221] Progress Software Corporation. Fiddler. `https://www.telerik.com/fiddler`, 2003. Accessed 26 March 2019.

[222] Borvid. HttpMaster. `https://www.httpmaster.net/`, 2013. Accessed 26 March 2019.

[223] Apache Software Foundation. Apache JMeter. `https://jmeter.apache.org/`, 1998. Accessed 26 March 2019.

[224] Katalon LLC. Katalon Studio. `https://www.katalon.com/`, 2015. Accessed 26 March 2019.

[225] Sam Van Oort. PyRestTest. `https://github.com/svanoort/pyresttest`, 2014. Accessed 26 March 2019.

[226] Parasoft Corporation. SOAtest. `https://www.parasoft.com/products/soatest`, 2002. Accessed 26 March 2019.

[227] SmartBear Software. SoapUI. `https://www.soapui.org/`, 2006. Accessed 26 March 2019.

[228] taverntesting. Tavern. `https://taverntesting.github.io/`, 2017. Accessed 26 March 2019.

[229] Corey Goldberg. WebInject. `http://www.webinject.org/`, 2004. Accessed 26 March 2019.

[230] authorjapps. Zerocode. `https://authorjapps.github.io/zerocode/`, 2016. Accessed 26 March 2019.

[231] RestBird. RestBird. `https://restbird.org/`, 2018. Accessed 26 March 2019.

[232] Restlet. Restlet Client. `https://restlet.com/modules/client/`, 2015. Accessed 26 March 2019.

[233] Optimizory Technologies. vREST. `https://vrest.io/`, 2015. Accessed 26 March 2019.

[234] Alex Friedman. Airborne. `https://github.com/brooklynDev/airborne`, 2014. Accessed 26 March 2019.

[235] Daniel Reid. Chakram. `http://dareid.github.io/chakram/`, 2015. Accessed 26 March 2019.

[236] Intuit. Karate. `https://github.com/intuit/karate`, 2016. Accessed 26 March 2019.

[237] Haleby Johan. REST Assured. `http://rest-assured.io/`, 2010. Accessed 26 March 2019.

[238] Patrick Poulin. API Fortress. `https://apifortress.com/`, 2014. Accessed 26 March 2019.

[239] APImetrics. APImetrics. `https://apimetrics.io/`, 2016. Accessed 26 March 2019.

[240] Assertible. Assertible. `https://assertible.com/`, 2016. Accessed 26 March 2019.

[241] Ping-API. Ping-API. `https://ping-api.com/`, 2016. Accessed 26 March 2019.

[242] Runscope. Runscope. `https://www.runscope.com/`, 2013. Accessed 26 March 2019.

[243] John Ferguson Smart. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Manning Publications, 2014.

[244] JUnit Team. JUnit 5. `https://junit.org/junit5/`, 2020. Accessed 20 February 2020.

[245] APIMATIC. APIMatic specification. https://docs.apimatic.io/, 2020. Accessed 20 February 2020.

[246] İsmail Taşdelen. SQL Injection Payload List. https://github.com/payloadbox/sql-injection-payload-list, October 2019.

[247] Sadayuki Furuhashi. MessagePack. https://msgpack.org/, 2008.

[248] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Jim A. Laredo, Julian Dolby, Christopher C. Young, and Aleksander A. Slominski. Opportunities in Software Engineering Research for Web API Consumption. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, pages 7–10, 2017.

[249] Beatriz A. Sanchez, Konstantinos Barmpis, Patrick Neubauer, Richard F. Paige, and Dimitrios S. Kolovos. RestMule: Enabling Resilient Clients for Remote APIs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 537–541, 2018.

[250] Yun Wan Kim, Mariano P. Consens, and Olaf Hartig. An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries. In *AMW*, 2019.

[251] Docker Inc. Docker Engine API Reference Version 1.40. https://docs.docker.com/engine/api/v1.40/, 2019.

[252] Transaction Processing Performance Council (TPC). TPC BENCHMARK™ App Specification Version 1.3. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-app_v1.3.0.pdf, February 2008.

[253] Transaction Processing Performance Council (TPC). TPC BENCHMARK™ C Specification Version 5.11. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, February 2010.

[254] Thomas J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[255] Tomas Topinka. Statistic plugin. https://plugins.jetbrains.com/plugin/4509-statistic, April 2020.

[256] JetBrains. IntelliJ IDEA. https://www.jetbrains.com/idea/, September 2019.

[257] Kengo Toda. SpotBugs. https://spotbugs.github.io/, April 2020.

[258] Philippe Arteau. Find Security Bugs plugin. https://find-sec-bugs.github.io/, October 2019.

[259] Thomas Becker. Oracle JDBC Driver bug report CORE-2130. https://liquibase.jira.com/browse/CORE-2130, November 2014.

This page is intentionally left blank.

Appendices

This page is intentionally left blank.

Appendix A

This chapter details the work plan which was initially laid out for this dissertation, in Section 8, where we list the set of expected tasks and highlight their current completion status. We also detail the outcomes derived from some of the tasks. In Section 8, we present a brief discussion on the actual performance of the execution of this dissertation.

Dissertation work plan and outcomes

This dissertation began in February 2019 and was expected to terminate in January 2020. The work plan initially laid out was divided into the two semesters that compose one academic year, and it encompassed the following tasks:

1. **First semester** (2nd semester of 2018/2019)
   - 1.1 Gather and analyze the state of the art on software robustness testing in the form of a systematic review.
   - 1.2 Gather and analyze a comprehensive set of studies on Representational State Transfer (REST) Application Programming Interface (API) testing.
   - 1.3 Gather and analyze a comprehensive set of industry tools for REST API testing.
   - 1.4 Define an approach for testing the robustness of REST APIs.
   - 1.5 Define the main quality attributes for the architecture of a tool for REST API robustness testing.
   - 1.6 Design the architecture of a tool for REST API robustness testing, based on the approach defined in task *1.4* and on the quality attributes defined in task *1.5*.
2. **Second semester** (1st semester of 2019/2020)
   - 2.1 Integrate the comments provided by the jury during the intermediate defense.
   - 2.2 Implement a tool for REST API robustness testing based on the architecture designed in task *1.6*.
   - 2.3 Evaluate the tool for REST API robustness testing implemented in task *2.2* on a small set of REST APIs.
   - 2.4 Use the tool for REST API robustness testing implemented in task *2.2* to perform a large-scale evaluation on online public REST APIs.
   - 2.5 Study the applicability of Machine Learning algorithms for the automated categorization of test results from the tool for REST API robustness testing implemented in task *2.2*.

Figure A.1 shows a Gantt chart which summarizes, for each task in the work plan, the expected start and end dates, the actual end date, and the activity and inactivity periods. We also show important dates such as the intermediate defense, the expected final defense, and the actual final defense.

As we can see in Figure A.1, by the intermediate defense, in July 2019, the completion status of the work plan (i.e., tasks *1.1* through *1.6*) was the following: task *1.1* was only partially completed, because the discussion of the systematic review was left unfinished; tasks *1.2* through *1.4* were completed, albeit with a one to two week delay; and the last

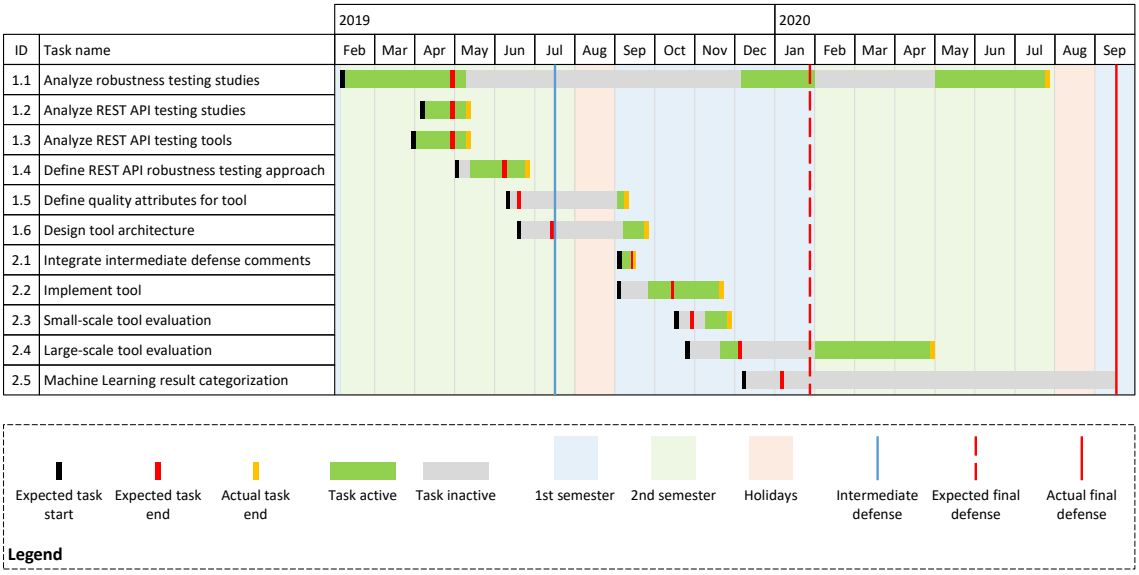


Figure A.1: Gantt chart showing the expected versus the actual work plan.

two tasks of the work plan (i.e., tasks *1.5* and *1.6*) had not been carried out, and their completion was thus postponed to the start of the following semester, in September 2019.

At this time we carried out and finished the delayed tasks *1.5* and *1.6*, and in parallel we proceeded to accomplish task *2.1*, where we integrated the comments provided by the jury in the intermediate defense. Given that task *2.2* depended on the completion of the delayed tasks *1.5* and *1.6*, the former was started with an accumulated delay of a few weeks, which itself resulted in both tasks *2.3* and *2.4* also being delayed, as the small-scale and large-scale (respectively) evaluations could only be performed once the tool (i.e., implemented in task *2.2*) reached a stable development state.

Because both tool evaluation tasks were delayed, around December 2019 the pending completion of task *1.1* had to be rushed, as the expected final defense (i.e., end of January 2020) was approaching. However, it soon became clear that this would greatly decrease the overall quality of the work, and it was instead decided to postpone the final defense to September 2020. This allowed us to invest far more time in the survey (task *1.1*) and in the large-scale evaluation (task *2.4*), by expanding both works and increasing their quality.

By the end of April 2020, we had expanded the large-scale evaluation from a testing campaign that covered 25 REST APIs, to a larger one covering a total of 52 REST APIs, and we thus concluded task *2.4*. The outcome of this task was a novel tool, named black-BOX tool for Robustness Testing of rest services (bBOXRT), for testing the robustness of REST APIs, and the paper *A Black Box Tool for Robustness Testing of REST Services* which we submitted to the Journal of Systems and Software (JSS). We then focused on completing task *1.1*, which had seen intermittent activity periods throughout the previous year. By July 2020, we had immensely expanded and improved upon the survey on software robustness testing, having more than doubled the initial set of 61 studies to a total of 132, as well as having included an extensive and thorough discussion with important highlights and challenges for future researchers. This allowed us to finally conclude task *1.1*, and its outcome was the paper *A Systematic Review on Software Robustness Assessment* which we submitted to the ACM journal on Computing Surveys (ACM CSUR).

At this point, not enough time was left to put effort into the remaining task *2.5*, and it was thus not carried out entirely. Curiously, we should note that during the intermediate

defense the jury did warn us about being overly ambitious in the inclusion of this task, and that even without it, this work plan already appeared challenging enough. In the following section, we present a brief discussion where we provide some insight on the positive and negative aspects regarding the work plan and the performance of the student.

Performance assessment

The execution of this dissertation suffered a 9-month delay compared to what was initially planned, resulting in the expected terminus of January 2020 being postponed to September 2020. We attribute this to a few factors. First, this is clearly the result of mismanagement, by the student, of the available time budget, especially during the first semester of this dissertation. As we observed in the previous section, the delay introduced during the execution of the first task (i.e., the survey on software robustness testing), itself resulted in a delay in a few other tasks. Given the dependency between some tasks (e.g., the tool architecture may only be designed when the quality attributes are selected and the approach defined), these small delays started accumulating into bigger delays, and this process repeated itself in a domino-like effect. This resulted in two tasks from the first semester of the work plan being left untouched, which in turn forced all planned tasks for the second semester to be postponed by a few weeks. The risks of this issue had already been pointed out by the jury during the intermediate defense, and here the student is responsible for a large part of the delays introduced into this work.

As a second factor, the student wishes to point out the large effort required to carry out the first task (i.e., the survey on software robustness testing), which was initially estimated to last three months but proved to be far more demanding. In particular, we conducted this survey in the form of a systematic literature review, a formal and highly methodical process which led us to manually analyze, summarize and classify more than 130 research contributions spanning three decades. The total accumulated time spent on this task was almost eight months, nearly three times the initially estimated amount.

Finally, the student acknowledges that there was a lack of risk management effort put into this work, and had the student initially accounted for the possibility of delays being introduced into this work, some mitigation strategies could have been defined. In hindsight, the tasks in the work plan could have been numerically prioritized, so as to distinguish the essential tasks from the minor ones and, given a large enough delay in the work (e.g., half a month), the lowest ranking minor task(s) would be excluded.

Still, despite the recurring delays, the student believes this work was successful in regards to the outcomes it produced. Specifically, we produced two high quality papers which have been submitted to international journals and are currently under review, and we developed a tool for testing the robustness of REST services (available at [32]) which, to the best of our knowledge and given the extensive analysis carried out over the state of the art, is the only one in existence at the time of writing.