isec Engenharia

DEPARTAMENTO DE INFORMÁTICA E SISTEMAS

Injecting Software Faults in Python Applications

Relatório de Trabalho de Projeto para a obtenção do grau de Mestre em Informática e Sistemas

Especialização em Desenvolvimento de Software

Autor

Henrique Manuel Domingues Marques

Orientadores

Prof. Jorge Bernardino

Prof. Nuno Laranjeiro

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, janeiro de 2022

This page is intentionally left blank.

Abstract

Software fault injection techniques have been largely used as means for evaluating the dependability of systems in presence of certain types of faults. Despite the large diversity of tools offering the possibility of emulating the presence of software faults, there is little practical support for emulating the presence of software faults in Python applications, which are increasingly being used to support business critical cloud services. In this thesis, we present a tool (named FIT4Python) for injecting software faults in Python code and then use it to analyse the effectiveness of OpenStack's test suite against new probable software faults. We begin by analysing the types of faults affecting Nova Compute, the core component of OpenStack. We use our tool to emulate the presence of new faults in Nova Compute API to understand how well OpenStack's battery of unit, functional, and integration tests cover these new, but probable, situations. Results show clear limitations in the effectiveness of OpenStack developers' test suites, with many cases of injected faults passing undetected through all three types of tests and that most of the analysed problems could be detected with trivial changes or additions to the unit tests.

Keywords

Software Faults, Bug Reports, Orthogonal Defect Classification, Fault Injection, Mutation Testing, Dependability Evaluation

Resumo

As técnicas de injeção de falhas de software têm sido amplamente utilizadas como meio para avaliar a confiabilidade de sistemas na presença de certos tipos de falhas. Apesar da grande diversidade de ferramentas que oferecem a possibilidade de emular a presença de falhas de software, há pouco suporte prático para emular a presença de falhas de software em aplicações Python, que cada vez mais são usados para suportar serviços *cloud* críticos para negócios. Nesta tese, apresentamos uma ferramenta (de nome Fit4Python) para injetar falhas de software em código Python e, de seguida, usamo-la para analisar a eficácia da bateria de testes do OpenStack contra estas novas, prováveis, falhas de software. Começamos por analisar os tipos de falhas que afetam o Nova Compute, um componente central do OpenStack. Usamos a nossa ferramenta para emular a presença de novas falhas na API Nova Compute de forma a entender como a bateria de testes unitários, funcionais e de integração do OpenStack cobre essas novas, mas prováveis, situações. Os resultados mostram limitações claras na eficácia da bateria de testes dos programadores do OpenStack, com muitos casos de falhas injetadas a passarem sem serem detectadas por todos os três tipos de testes. Para além disto, observamos que que a maioria dos problemas analisados poderia ser detectada com mudanças ou acréscimos triviais aos testes unitários.

Palavras-Chave

Falhas de Software, Análise de bugs, Orthogonal Defect Classification, Injeção de falhas, Testes de mutação, Resiliência de software

This page is intentionally left blank.

Acknowledgments

This work has been supported by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020; by project VALU3S - Verification and Validation of Automated Systems' Safety and Security, H2020-EU.2.1.1.7. - ECSEL, Grant agreement ID: 876852; and by project AIDA: Adaptive, Intelligent and Distributed Assurance Platform project, co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalization – COMPETE 2020 (POCI-01-0247-FEDER-045907) and by the Portuguese Foundation for Science and Technology under CMU Portugal Program.

We also would like to acknowledge the National Distributed Computing Infrastructure (INCD) for providing access to computing resources to execute the fault injection test campaign.

I would like to express my sincere gratitude to Prof. Nuno Laranjeiro and Prof. Jorge Bernardino for the invaluable guidance and motivation provided during my MSc thesis, their knowledge and plentiful experience made me grow as an engineer.

I would also like to thank my family and my fiancee for the incredible support during this long endeavor.

This page is intentionally left blank.

Contents

1 Introduction **1**

2 Background and Related Work **4**
- 2.1 Background concepts 4
- 2.2 Fault injection 7
- 2.3 Mutation testing 10
- 2.4 Related Work Gaps 12

3 FIT4Python – A Software Fault Injection Tool for Python **14**
- 3.1 Tool Overview 14
- 3.2 FIT4Python Components and Operation 15

4 Fault Model **18**

5 OpenStack Case study **26**

6 Fault Reduction Analysis **32**

7 Main Findings **36**

8 Threats to Validity **40**

9 Conclusion and Future Work **42**

This page is intentionally left blank.

Acronyms

API application programming interface. 5, 6

DHCP Dynamic Host Configuration Protocol. 5

IaaS Infrastructure as a Service. 5

IP Internet Protocol. 5

ODC Orthogonal Defect Classification. 4, 5, 8, 12

SUT System Under Test. 7, 8

VLAN Virtual Local Area Network. 5

VM virtual machines. 5, 6

This page is intentionally left blank.

List of Figures

2.1 Openstack's component architecture. Adapted from [OpenStack, 2021]. . . . 6
2.2 Software fault injection process, adapted from [Natella et al., 2016b] 7
2.3 Mutation testing process. 10

3.1 Basic operation of the FIT4Python tool. 14
3.2 FIT4Python components and operating mode. 15

6.1 Distribution of R^2 by number of fault types 35

This page is intentionally left blank.

List of Tables

2.1 Fault injection operators proposed by [Durães and Madeira, 2006]. 9

4.1 OpenStack component development metrics. 19
4.2 ODC defect type results 19
4.3 ODC qualifier results 20
4.4 Verification results for Defect Type. 20
4.5 Verification results for Qualifier. 20
4.6 Accuracy and Cohen's Kappa Agreement between researcher1 and researcher2. 21
4.7 Fault types based on the ODC extension in [Durães and Madeira, 2006]. . . 22

5.1 Experimental results overview. 27
5.2 Experimental results grouped by Defect Type. 28
5.3 Experimental results grouped by Qualifier. 28
5.4 Analysed problems and necessary corrections. 30

6.1 Sufficient fault types. 34

Chapter 1

Introduction

Software Fault Injection is a well known technique used for evaluating dependability of systems in presence of faults. Traditionally aiming at reproducing the effects of hardware faults it was later used to reproduce effects of software faults [Hsueh et al., 1997]. Some approaches put particular focus on the emulation of the software fault itself, by usually inserting the faulty code in the target application [Natella et al., 2016a].

The injection of code changes may occur in several different ways depending on factors that mostly relate to the nature of the programming language and associated tools used. If a particular language compiles to machine code, possible techniques include changing the source code or change the machine code directly [Durães and Madeira, 2006]. In case the language is interpreted, the immediate option is to change source code to emulate the presence of software defects [Fonseca et al., 2014]. If intermediate code compilation exists, byte code manipulation may also be a viable option [Sanches et al., 2011]. In many cases, we are able to use abstract forms of the code (e.g., an abstract syntax tree) to inject a particular kind of fault [Cotroneo et al., 2019, Hajdu et al., 2020].

The programming language itself limits the constructs available, rendering certain types of known software faults useless, or, on the other hand, creating space for new types of faults to emerge. For instance, a well-known software fault like missing value assignment [Durães and Madeira, 2006] is caught by current Java compilers, thus it should not be used when injecting faults in Java-based software. In addition, a fault model designed for C, such as the one presented in [Durães and Madeira, 2006], will not include faults related with, for instance, exception handling that may occur if the language used is Python, Java or C#. Certain faults may appear more rarely due to the different context involved, including the maturity of development tools which may warn the developer of certain mistakes and, thus, potentially lower their probability of occurrence. This kind of specificities brings in the need for new specialized approaches and tools for software fault injection [Natella et al., 2016a].

With the growing popularity of Python [Tiobe, 2019], especially in business-critical environments, having a way to emulate software faults in Python code can be extremely helpful for two main general reasons: i) *during software verification activities*, a fault injection tool may be used to create representative faulty versions of a certain system under test, which can be used to assess the effectiveness of an existent test suite (e.g., allowing the test suite to be corrected or extended, thus fostering the system's dependability) [Natella et al., 2016b, Pizzoleto et al., 2019]; and ii) *after other software verification activities*, a fault injection tool can be used to introduce faults in a system component, allowing to evaluate fault tolerance mechanisms and, overall, allowing to understand how dependable the whole system is in presence of a faulty component [Natella et al., 2016b, Kanoun and Spainhower, 2008]. Such tool must include not only the basic ability to inject *representative* faults (i.e., faults that represent typical mistakes performed by developers), but also the associated infrastructure for handling all the process (e.g., reading configuration regarding the faults to inject, manipulating code files). Indeed, recent work has emphasized the fact that research targeting Python software is scarce, especially in what concerns software fault injection [Cotroneo et al., 2019].

The goal of this thesis is two-fold: i) we present a tool for injecting software faults (i.e., software bugs) by applying code changes to Python source code; and ii) we use the tool to analyse the effectiveness of the test suite of OpenStack, a Python-based business-critical cloud platform, against new probable software faults injected by our tool. We begin by analyzing the types of software faults (i.e., software bugs) affecting OpenStack's main component – Nova Compute, and define a fault model for this core component. Based on this fault model, we implement our tool and run it to create faulty versions of the Nova Compute API. We then run OpenStack developers' unit, functional, and integration tests against the faulty versions. Our results, which are available in detail at [Marques et al., 2021], were collected from the execution of more than 245 Million tests (i.e., 21,683 tests present in the test suite ran against 11,309 faulty versions) and show not only the ability of the tool to inject different kinds of faults, but especially show the fault detection limitations of the OpenStack developer's test suite, with several cases of faults passing silently undetected through the tests. Moreover, we show that, for the analyzed issues, it would be trivial to correct or extend most of OpenStack tests, in order to increase their coverage regarding those (probable) faults, which currently pass undetected through the different types of tests that compose OpenStack's test suite.

The main contributions of this work are the following:

- A tool for injecting software faults in Python applications, freely available at [Marques et al., 2021];
- A fault model characterizing software faults that affect OpenStack, a well-known Python-based cloud platform used in business-critical environments (labelled dataset available at [Marques et al., 2021]);
- The practical application of the tool to the OpenStack Nova Compute API in more than 245 Million tests, with the identification of clear limitations in the effectiveness of OpenStack's battery of unit, functional, and integration tests, which showed to not be able to detect the presence of certain types of *probable* bugs.

The outcome of this work is the publication of the following paper:

- Henrique Marques, Nuno Laranjeiro, Jorge Bernardino, "Injecting Software Faults in Python Applications: The OpenStack Case Study", accepted for publication in the Journal of Empirical Software Engineering, doi:10.1007/s10664-021-10047-9.

The remainder of this thesis is organized as follows. Chapter 2 presents background on the characterization of software faults and basic concepts regarding OpenStack and proceeds with related work on two aspects that are central to this thesis: i) fault injection tools and techniques; ii) and mutation testing approaches and tools. Chapter 3 presents a conceptual view of the fault injection tool architecture and operating mode of its components. Chapter 4 describes the characterization of software faults performed for OpenStack, the selected Python case study for this work, whereas Chapter 5 presents the experimental evaluation carried out to illustrate the practical usefulness of the tool. Chapter 6 explores the cost/benefit trade-off associated with injecting larger numbers of faults. Chapter 7 summarizes the main findings we observed during the execution of our work, while Chapter 8 discusses the main threats to the validity of this work. Finally, Chapter 9 concludes this thesis.

Chapter 2

Background and Related Work

This chapter presents background concepts on two main topics. First, we discuss methods for characterizing software faults, which is an important aspect that should be considered in fault injection campaigns. Second, we overview OpenStack - the system we selected to be part of our the experimental evaluation. The chapter then proceeds with two main parts: i) the discussion of related work on *fault injection* techniques; and ii) tools and the related work on *mutation testing* tools and approaches. The chapter concludes with a brief discussion regarding the main differences between the related work and this thesis.

2.1 Background concepts

There are a few relatively well-known fault classification schemas, which, in this context, are useful means for defining fault models, and that are essentially sets of types of faults known to be representative of what is experienced by a certain system during operation. Hewlett-Packard's Defect Origins, Types and Modes (DOTM) [Grady, 1992], IEEE's standard 1044-2009 (IEEE-1044) [IEEE, 2010], and Orthogonal Defect Classification (ODC) [Chillarege, 1996, IBM, 2013] are relatively well-known.

Orthogonal Defect Classification is a set of analytical methods mostly used for the analysis of software development and test processes [Chillarege, 1996, IBM, 2013]. ODC was created with the intention of bridging two methods commonly used for analyzing defects, namely Statistical Defect Modelling [Wood, 1996] and Root Cause Analysis [Wilson et al., 1993]. It allows the defect classification process to be faster (as in the former method) and have better accuracy in categorizing issues (similarly to the latter) [Chillarege, 1996].

The Orthogonal Defect Classification characterizes software defects according to eight orthogonal attributes, grouped into two sections: opener section and closer section [IBM, 2013]. The opener section refers to attributes that can be classified when the defect is found (i.e., it does not consider the aspects related with the correction of the defect) and includes the following attributes:

- Activity: describes the activity being performed at the time the defect was identified (e.g., system testing);
- Trigger: indicates what caused the defect to surface, i.e., the required condition that allowed the defect to manifest;

- Impact: refers to either the impact a user experienced when the defect surfaced, or to the impact a user would have suffered, if the defect had surfaced.

The closer section refers to the attributes that can be classified when a defect has been corrected and the correction information becomes available. Of the following attributes of the closer section, we highlight **defect type** and **qualifier**, as they characterize the type and nature of a certain software fault [IBM, 2013]:

- Target: the object of the correction (e.g., source code); Age: the instant in time in which the defect was introduced (e.g., introduced during the correction of another bug); Source: whether the defect was introduced by an external component, or was something introduced in-house, i.e., by the team developing the product;
- **Defect Type:** refers to the type of change that is performed to correct a certain bug (e.g., changing a checking condition) and can take one of the following seven values: *Assignment/Initialization*, *Algorithm/Method*, Checking, *Function/Class/Object*, *Timing/Serialization*, *Relationship*, and *Interface/O-O Messages*;
- **Qualifier:** Describes the implementation prior to having been corrected. That is, whether it was *missing*, *incorrect*, or *extraneous* (i.e., present but unnecessary).

To be applied correctly, ODC, requires a couple of conditions to be achieved:

- The semantic information of the known defects must be available to be possible to classify the opener section of the concept.
- The code fix for each defect must be available to be possible to classify the closer section of the concept.

OpenStack Nova [Rosado and Bernardino, 2014], an open-source cloud computing platform that provides Infrastructure as a Service, widely used in the industry, fulfills both previous conditions. The semantic information of the defects can be found at the Launchpad platform https://launchpad.net/nova and the code fix for each defect is linked to the issue itself. Nova's code is available at https://github.com/OpenStack/nova and the platform itself is composed by several components. Figure 2.1 presents the conceptual relation between the components of Openstack. We describe the responsibilities of each component in the following paragraphs.

- Compute (Nova): OpenStack compute (codename: Nova) is the component that allows the user to create and manage virtual servers using the machine images. It is considered the brain of the OpenStack and it is responsible for provisioning and managing large networks of virtual machines interacting with hypervisors, such as KVM, Xen, VMware, and Hyper-V. For being considerate the core component of OpenStack it was chosen as the subject for this study.
- Glance: Glance is the OpenStack Image Service. It provides an API to Nova for discovering, retrieving and registering VM images.
- Neutron: Neutron provides network connectivity as a service, it allows for managing DHCP, static IP and VLAN.

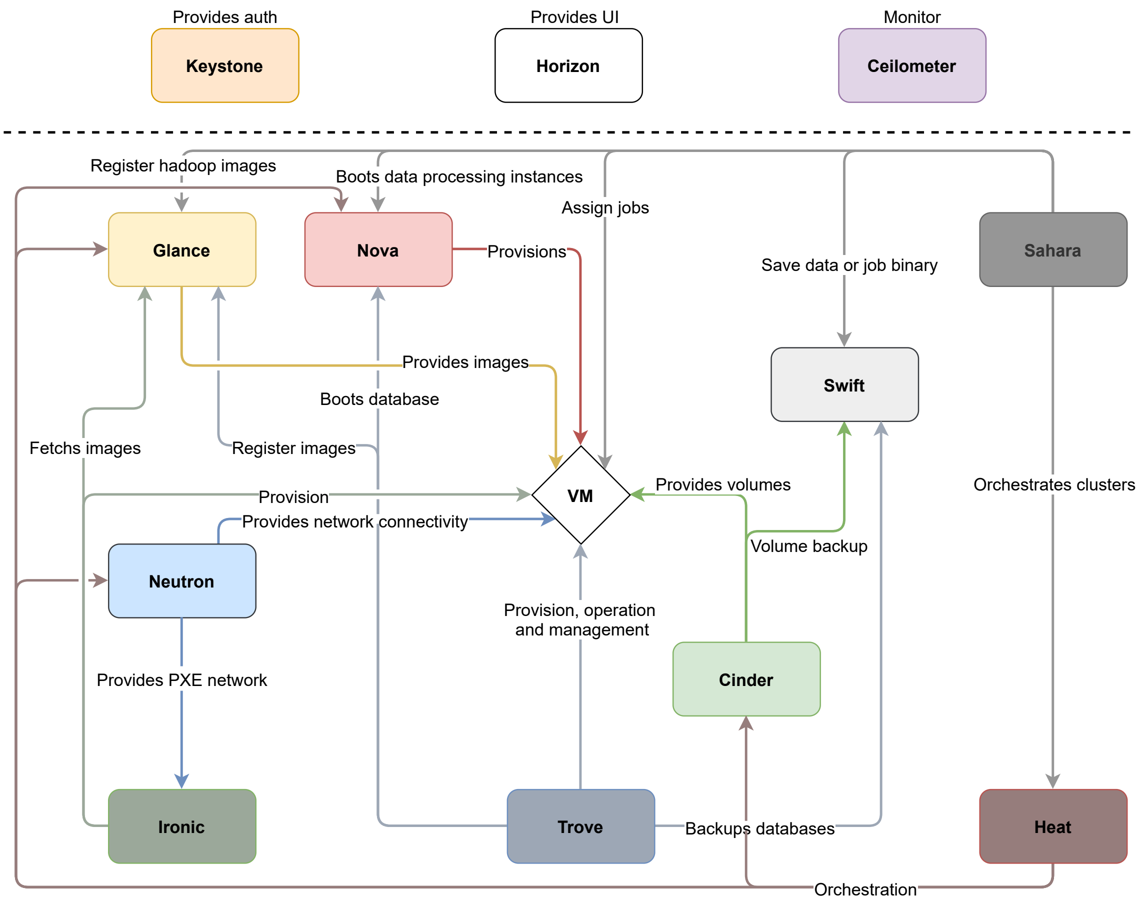

Figure 2.1: Openstack's component architecture. Adapted from [OpenStack, 2021].

- Swift and Cinder: Swift and Cinder provide storage mechanisms for OpenStack with the difference being on the type of storage. Swift is a highly available object/blob store and allows users to store and retrieve files from it. Cinder is a more persistent storage and is used by Nova to store volumes for VM, manipulate those volumes and save snapshots.
- Keystone: Keystone is a single point of integration between all components of OpenStack and provides token and authentication for users and services interaction.
- Horizon: OpenStack has a dashboard that provides a user interface for managing and interacting with all services and APIs, called Horizon.
- Ceilometer: Ceilometer is an optional component that provides billing and metering. With Ceilometer is possible to measure CPU and network costs.
- Ironic: Ironic is an integrated Openstack program responsible for provisioning bare metal machines instead of virtual machines.
- Trove: Trove provides scalable and reliable cloud database as a service.
- Heat: Heat allow the creation of human and machine accessible services for managing the entire lifecycle of infrastructure and applications within Openstack clouds.
- Sahara: Sahara provides means to provision a data-intensive application cluster (Hadoop or Spark) on top of Openstack.

2.2 Fault injection

Software fault injection is a vastly explored field, where research has focused on the injection of the effects of faults in the software or, more precisely, on the emulation of software faults. The main idea involved is the possibility of evaluating the behavior of the target system (e.g., in terms of dependability) in presence of a certain faulty component [Natella et al., 2016b, Kanoun and Spainhower, 2008]. The characteristics of the injected fault should ideally be representative of what is found in real systems [Durães and Madeira, 2006].

Fault injection can be performed mostly using three different techniques [Natella et al., 2016a]: *injection of data errors* [Arlat et al., 2003, Barton et al., 1990], *interface error injection* [Miller et al., 1995, Koopman and DeVale, 2000] and *injection of code changes* [Durães and Madeira, 2006], illustrated in Figure 2.2 and explained in more detail in the next paragraphs.

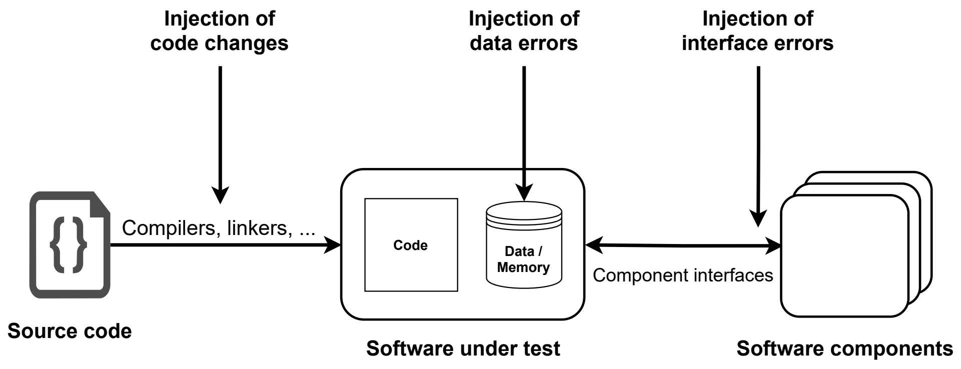

Figure 2.2: Software fault injection process, adapted from [Natella et al., 2016b]

Injection of data errors aims to reproduce the effects of hardware faults by corrupting memory or registers that are being used by the System Under Test (SUT) through software. The *Fault-Injection-based Automated Testing environment* (FIAT) [Barton et al., 1990], was one of the first tools developed for software fault injection, and was capable to emulate both hardware and software faults by injecting data errors on memory. The tool was able to overcome the operating system restrictions that prevented one process to modify the memory under use by another process. The tool consisted on a dynamic library that would be linked to the SUT at compile time. This required the code of the SUT to be publicly available which is not the case for all kinds of programs (e.g., third-party software).

After the proposal of FIAT, *The integrated software fault injection environment* (DOCTOR) [Han et al., 1995] was created. This injector was implemented at the kernel level allowing to introduce a layer in the protocol stack to intercept the communication between the SUT and lower protocol layers to inject data errors. This technique was not designed to emulate software faults in a representative way as software faults can be much different then faults caused by hardware errors such as bit-flips. Moreover, these injected faults might not have an effect on the target system if the injected location is not being actively used by the SUT or if the memory is overwritten before being used.

Interface error injection focuses on corrupting the input and output of components that are part of the software being tested.

The *Fuzz and Ptyjig: Robustness Testing of UNIX Applications Through Interface Error Injection* [Miller et al., 1990] was one of the first studies to be published on interface error injection with the goal to evaluate the robustness of UNIX utilities. Two tools, Fuzz and ptyjig, were developed to submit random stream of data as input to the SUT applications, and the study found that a significant number of utility programs were vulnerable to interface errors. Injecting random data streams as input proved to be a cheap but effective way to discover bugs in commercial applications, fueling further research on how to generate corrupted data in a systematic way.

The *FIG: Robustness Testing of UNIX Applications Against the C Library* [Broadwell et al., 2002] aimed to test the robustness of desktop and server applications by injecting errors from the C library. To accomplish this task, a wrapper library was developed to mimic the C library but returning the most common erroneous return codes from methods such as malloc, read, write, select and open (e.g., failed memory allocation).

Finally, *injection of code changes* tries to emulate bugs by introducing wrong code in the software and analysing the behaviour of the application, as a whole, when that bug is executed. Several works have looked at the problem of injecting faults via changing program code. In [Mahmood et al., 1984], the authors placed faults in source code with the main goal being to understand the effectiveness of code assertions in capturing errors. In [Hudak et al., 1993], a tool was used to inject software faults in source code, with the goal of evaluating different fault tolerance mechanisms. A tool that is able to introduce faults in binary code is presented in [Kao et al., 1993] and was used to evaluate Unix operating systems in presence of faults. The types of faults supported were extended later in [Wei-Lun Kao and Iyer, 1994] and the tool was improved to support execution in a distributed environment.

In [Ng and Chen, 2001], the authors use fault injection to evaluate the Rio file cache, which has the main goal of allowing the use of main memory for persistent storage by surviving operating system crashes. A fault model including different types of typical software faults like initialization, off-by-one or interface error, was designed to be used for the evaluation. Some of the faults are injected in machine code, other by modifying behavior of kernel functions.

The problem of emulating representative software faults for software reliability and dependability assessment, and, in particular, the aspect of injecting software faults in a source-code independent manner was studied in [Durães and Madeira, 2006]. The authors analyse a total of 668 real software defects in C applications and classify them using the Orthogonal Defect Classification. Despite providing a solid foundation for the emulation of faults, the ODC attributes *defect type* and *qualifier* are too broad and do not characterize developers mistakes in a detailed manner, at least not with the sufficient detail necessary for fault injection campaigns (e.g., a fault that is of type *assignment* and qualified as *incorrect* may take numerous forms). Considering this aspect, a finer detailed analysis of the defects is carried out in [Durães and Madeira, 2006] where, for instance, *assignment* faults that are qualified as *incorrect* are further divided in different 13 cases (identified in the authors' dataset). This finer characterization, which can be seen as an extension to ODC, include cases like *wrong arithmetic expression used in assignment* and *wrong miss-by-one value used in variable initialization*, which represent the necessary information to represent a certain software fault. In total the authors identified 62 of these 'fault operators' listed in Table 2.1, that allow carrying out fault injection campaigns and that were defined based on real developer mistakes found in C applications.

A tool for software fault injection in Java applications, named JACA, is presented in [Martins et al., 2002]. JACA uses Reflection to inject faults by corrupting attribute values, methods parameters or return values. In [Sanches et al., 2011] the authors present the Java Software Fault Injection Tool, that does not require the presence of source code as injection is carried out directly in the compiled form of the code.

Table 2.1: Fault injection operators proposed by [Durães and Madeira, 2006].

ODC Defect Type	ODC Qualifier	ID	Description
Algorithm / Method	Extraneous	EIFS	Extraneous Function Call
	Missing	MCA	Missing iteration construct around statements
		MFC	Missing function call
		MIEA	Missing If construct plus else plus statements around statements
		MIEB	Missing if construct plus statements plus else before statements
		MIES	Missing if-else construct plus statements
		MIFS	Missing if construct plus statements
		MLPA	Missing small and localized part of the algorithm
		MLPL	Missing large part of the algorithm
		MCS	Missing case: statement(s) inside a switch construct
		MBC	Missing break in case
		MLPS	Missing sparcely spaced parts of the algorithm
	Wrong	WALD	Wrong Algorithm - small sparce modifications
		WALR	Wrong Algorithm - code was misplaced
		WFCD	Wrong function called with different paramenters
		WBC1	Wrong branch construct - goto instead break
		WSUC	Wrong conditional compilation definitions
		WFCS	Wrong function called with same parameters
Assignment / Initialization	Extraneous	EVAV	Extraneous variable assignment using another variable
		EVAL	Extraneous variable assignment using a variable
	Missing	MVAE	Missing variable assignment using an expression
		MVAV	Missing variable assignment using a value
		MVIE	Missing variable initialization using an expression
		MVIV	Missing variable initialization using a value
		MVAI	Missing variable auto-increment
		MVAD	Missing variable auto-decrement
		MLOA	Missing OR sub-expr in larger expression in assignment
		MLAA	Missing AND sub-expr in larger expression in assignment
	Wrong	WIDS	Wrong string in initial data
		WIDSL	Wrong string in initial data - missing one char
		WSUT	Wrong data types or conversion used
		WVAE	Wrong arithmetic expression used in assignment
		WVAL	Wrong logical expression used in assignment
		WVAM	Miss by one value assigned to variable
		WVAV	Wrong value assigned to variable
		WVIV	Wrong value used in variable initialization
		WPLA	Wrong parenthesis in logical expression used in assignment
		WVIM	Wrong miss-by-one value used in variable initialization
		WIDI	Wrong constant in initial data
		WIDIM	Wrong miss-by-one constant in initial data
		WIDM	Wrong initial data - array has value in worng order
Checking	Extraneous	ELOC	Extraneous "OR EXPR" in expression used as branch condition
	Missing	MIA	Missing if construct around statements
		MLAC	Missing "AND EXPR" in expression used as branch condition
		MLOC	Missing "OR EXPR" in expression used as branch condition
	Wrong	WLEC	Wrong logical expression used as branch condition
		WPLC	Wrong parenthesis in logical expression used as branch condition
		WAEC	Wrong arithmetic expression in branch condition
Function / Class / Object	Missing	MFCT	Missing functionality
	Wrong	WALL	Wrong algorithm - large modifications
Interface /O-O Messages	Missing	MPFC	Missing paremeter in function call
		MLOP	Missing OR sub-expression in parameters of function call
		MLAP	Missing AND sub-expr in parameters of function call
		MRS	Missing return statement
	Wrong	WPFL	Wrong value used in parameter of function call
		WPFO	Wrong paramenter order in function call
		WPLP	Wrong parenthesis in logical expression in parameters of function call
		WLEP	Wrong logical expression in parameter of function call
		WAEP	Wrong arithmetic expression in parameter of function call
		WPFML	Miss by one value in parameter of function call
		WPFV	Wrong variable used in parameter of funtion call
		WRV	Wrong return call

The SAFE tool is used in [Natella, 2011], which is able to inject software faults in C and C++ applications. The tool implements 13 fault operators, with injection being performed at the source code level. In [Cotroneo et al., 2019] the authors empirically analysed a total of 179 OpenStack bug reports (i.e., about 8% of the total number of bugs analysed in this work and about one third of the bugs for which we have performed a detailed analysis). The authors use a tool which generates an Abstract Syntax Tree and inject faults in the tree, converting the result back to source code and place their focus on understanding failures. The authors have recently further developed their work in [Cotroneo et al., 2020] to propose a fault injection tool offered as a service, which supports the user through the definition of some of the key steps of a fault injection campaign (e.g., workload and faultload definition, failure data analysis).

One well known problem of fault injection approaches is the cost to execute tests. As the number of potential faults and locations to inject them is generally very high, in practice the tester may be faced with numerous cases that must be executed, which is a time-consuming activity. This is a very well-known problem in the mutation testing community [Woodward, 1993].

2.3 Mutation testing

Fault injection is mostly used to understand the effect of the activation of faults in a certain system, thus the target is set on the behavior of the system (e.g., if the system can tolerate the activation of a certain fault). On the other hand, **mutation testing** has the main goal of improving the tests effectiveness, by firstly evaluating their ability to defect software faults (which allows improving the tests themselves) [Natella et al., 2016b] like illustrated in Figure 2.3. *Mutants* (i.e., faulty versions of a certain program being evaluated) are created by fault injection (fault types are named *mutation operators*), which results in the problem of generating too many different mutants [Pizzoleto et al., 2019].

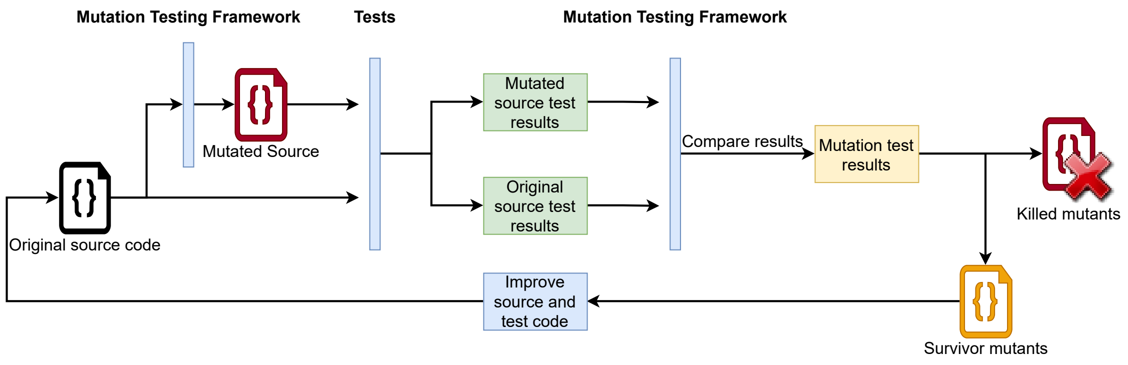
Mutation Testing Framework
Tests
Mutation Testing Framework
Mutated Source
Original source code
Mutated source test results
Original source test results
Compare results
Mutation test results
Killed mutants
Survivor mutants
Improve source and test code

Figure 2.3: Mutation testing process.

A huge number of different solutions for the above mentioned problem have been proposed over the last decades and generally target the following goals, as discussed in detail in [Pizzoleto et al., 2019]: i) reducing the total number of mutants; ii) detecting equivalent mutants (i.e., mutants that are equal to the original program); iii) reducing the execution time; iv) reducing the number of test cases; v) generating specific mutants only; vi) generating test cases.

Within the whole set of specific techniques that aim at reducing the overall cost of mutation testing, and that range from parallel execution [Li et al., 2015, Wang et al., 2017], to random mutation techniques [Kurtz et al., 2016, Petrović and Ivanković, 2018], compiler optimization [Denisov and Pankevich, 2018, Kintis et al., 2018], minimization and prioritization of test sets [Derezińska, 2013, Zhang et al., 2013], evolutionary algorithms [Lima and Vergilio, 2018, Fraser and Arcuri, 2015], among several others, we find classic cases that particularly focus on reducing the application of certain *mutation operators*, namely *selective mutation* [Zhang et al., 2013, Praphamontripong and Offutt, 2017] and *sufficient operators* [Namin et al., 2008].

Parallel execution [Byoungju and Mathur, 1993] is a technique, where mutants are executed against tests at the same time, by utilising the multiple cores of the processor. In [Li et al., 2015], parallel execution of mutation testing was performed utilising several Amazon EC3 virtual machines with different characteristics and a performance and cost analysis was done. The results showed that the testing tool was capable of taking advantage of the numbers of cores and more powerful machines executed the tests in less time.

Selective mutation proposes to avoid *operators* that are responsible by large number of *mutants* or that generate *mutants* that are killed by tests (i.e., do not pass the tests) that also kill *mutants* of other operators. Selective mutation techniques have been proposed, for instance, in [Praphamontripong and Offutt, 2017], where the authors evaluate the degree of redundancy among mutation operators for Web applications by analyzing the types of mutants that can be killed by tests that have been created with the purpose of killing other types of mutants. Another example can be found in [Delgado-Pérez et al., 2017], where authors evaluate operators for C++ classes using two rankings. One of the rankings sorts the operators based on their redundancy, the other one sorts operators based on the quality of the tests they allow to generate.

Sufficient operator techniques, which can also be seen as a particular case of selective mutation, focus on discovering a subset of *operators* that still accurately measures the effectiveness of the tests for the system under test. As examples, in [Just and Schweiggert, 2015], the authors propose non-redundant versions of a few operators (e.g., conditional replacement, unary insertion, relational replacement), which allowed for a decrease of over 20% in the run time of the experiments. In [Namin et al., 2008], the authors apply statistical methods, namely linear regression analysis, to programs written in C to identify a set of sufficient operators. Previous research has discussed the fact that the mutation scores used may be inflated, due to the fact of some mutants being redundant with respect to each other, which centers the problem on finding more realistic mutation scores [Kurtz et al., 2016].

Mutation testing is an established technique that is supported by numerous tools that allow generating mutants and collecting results from the execution of the tests. There are a few tools built for mutation testing of Python code, such as Mutmut [Hovmöller, 2021], Cosmicray [AS, 2021], Mutatest [Kepner, 2021], or MutPy [Hałas, 2021]. In general, these tools use code as input, produce an abstract form of it (e.g., an abstract syntax tree), generate mutants, based on predefined operators, and run certain types of tests (e.g., typically unit tests).

Mutatest [Kepner, 2021] uses a list of 12 relatively low-level operators (e.g., bitwise comparison and shift, arithmetic operation mutations) that mostly fit the *Assignment/Initialization* and also *Checking* ODC defect type attributes. MutPy [Hałas, 2021] is based on a limited list of 20 operators that tend to represent higher level operations like exception handler deletion, break/continue replacement and also touches the Interface/O-O messages and Relationship ODC defect type attributes. Mutmut [Hovmöller, 2021] relies on the following types of mutations: i) operator mutations (e.g., replacing '+' with '-', replacing 'or' with 'and'); ii) keyword mutations (e.g., replacing 'in' by 'not in'); iii) numeric mutations (e.g, adding one to a number); iv) name mutations (e.g., replacing 'copy' with 'deepcopy'); v) argument mutations (e.g., changing the name of a key in a key-value data structure); and vi) string mutations (i.e., appending a constant to strings). In the case of Mutmut, despite the large variety of operators, the focus is set on cases that mostly tend to fit *Assignment/Initialization* and also *Checking* defect types. As we will see, in Chapter 4, our study identifies representative faults with a level of detail that is not present in any of these tools.

2.4 Related Work Gaps

As a summary, current works on fault injection tend to resort to generic fault models (e.g., such as the one proposed in [Durães and Madeira, 2006]), built based on the analysis of C applications, which may not be a good fit for newer types of systems, written in different programming languages and different contexts. In the state of the art, research targeting Python code is quite scarce (e.g., an example can be found in [Cotroneo et al., 2019], but based on the analysis of a relatively small number of bug reports). The fault injection applied in this thesis differs from mutation testing, with more operators being applied and with more relevance based on the analysis of known faults using ODC.

In this work, we aim towards the definition of a configurable and extensible tool (in terms of implemented faults) that is particularly tailored for the evaluation of multiple target systems in a *distributed* manner and in *parallel* (i.e., for reducing the overall execution time). By applying our tool, we aim at understanding, the overall effectiveness of the battery of tests of a business-critical cloud platform (OpenStack, in our case), pinpointing cases of tests that need to be modified, missing test cases, and summarizing general recommendations for developers and testers, based in our findings. We also apply a typical fault reduction technique [Namin et al., 2008], with the goal of understanding the cost/benefit involved with running the battery of tests against all faulty OpenStack versions, or running them against just a subset.

This page is intentionally left blank.

Chapter 3

FIT4Python – A Software Fault Injection Tool for Python

In this chapter, we provide an overview of the main fault injection concept supported by our tool, which we named FIT4Python (Fault Injection Tool for Python). We then go through a basic view of its architecture, identifying the main components involved and their role in the operation of the tool.

3.1 Tool Overview

The main idea behind FIT4Python is to allow emulating the presence of a certain type of software fault in Python code. In practice it is able to modify the code, so that the bug is inserted at an adequate code location, as illustrated in Figure 3.1.

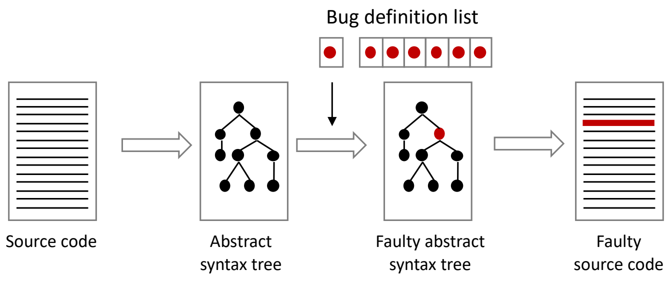

Figure 3.1: Basic operation of the FIT4Python tool.

Based on a certain bug definition (e.g., missing assignment, incorrect check), our tool is able to: i) identify candidate locations at the source code of a certain system being tested; and ii) insert the bug at one selected location. Starting from a source code file, the tool first generates the corresponding Abstract Syntax Tree (AST). Then, based on the definition of a single software fault (i.e., a software bug of a certain type, like a missing assignment or invalid check) it is able to identify candidate locations in the AST and change one single location, so that the AST now carries the bug. It then converts the AST back to source code, which results in the generation of a faulty code version (i.e., carrying one known bug of a certain type) that is then ready to be used. This is the main concept supported by the tool, which we describe in further detail in the next section.

3.2 FIT4Python Components and Operation

Figure 3.2 illustrates the four main components of our tool (in dark blue in Figure 3.2) and how they collaborate at runtime, so that it is possible to generate and test a faulty version of a given Python application.

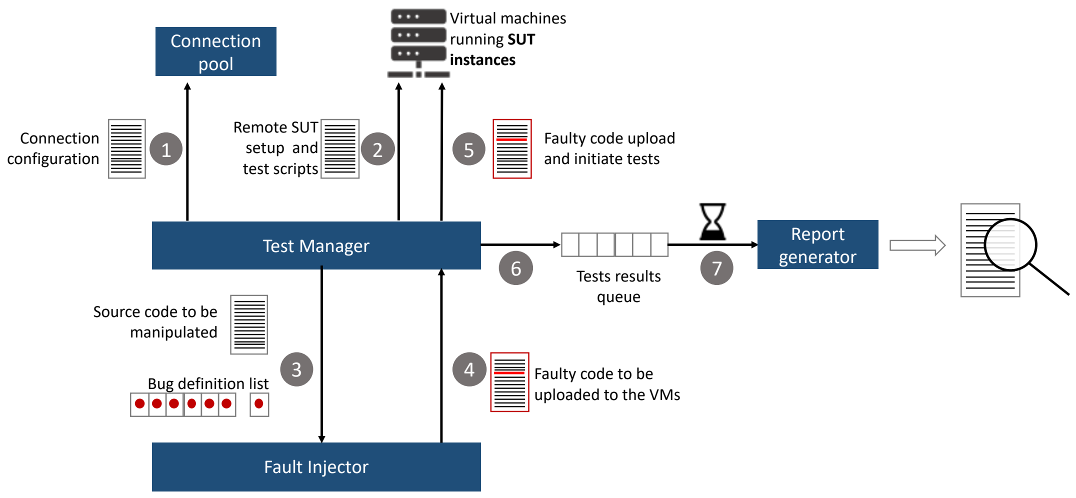
Connection pool
Virtual machines running SUT instances
Connection configuration
1
Remote SUT setup and test scripts
2
5
Faulty code upload and initiate tests
Test Manager
6
Tests results queue
7
Report generator
Source code to be manipulated
3
Bug definition list
4
Faulty code to be uploaded to the VMs
Fault Injector

Figure 3.2: FIT4Python components and operating mode.

Our tool begins by reading the configuration, namely the connection configuration to remote machines (e.g., virtual machines (VM) that will be accessed by SSH and SFTP) responsible for running tests. Besides performing fault injection, FIT4Python is also able to trigger a set of predefined tests (provided by the user of the tool) to be run over faulty version(s). It then configures a connection pool (**step 1** in Figure 3.2) to be used later to prepare and execute tests.

In **step 2**, a given number of virtual machines is prepared to run tests, namely a copy of the System Under Test (SUT) without artificially introduced bugs is deployed and a battery of tests (e.g., unit, functional, and integration tests) is uploaded. Thus, all virtual machines are setup with the unmodified SUT instances ready to be executed.

The Test Manager, in **step 3**, orders the Fault Injector to read one file (the first file of a list of target files with size T, selected by the tool user) that will be used as the target of fault injection. At the same time, it also informs the Fault Injector that it should use the first software fault from a list of predefined software faults (with size B and represented by 'Bug definition list' in Figure 3.2) for performing fault injection. Based on the two inputs (i.e., the target file and the fault definition), the fault injector identifies all candidate locations (L) in the AST corresponding to the source code of the provided file, where it is possible to inject the software fault. Certain faults will also have additional configuration (C); for instance, it is possible to emulate a wrong assignment with a boolean, text, or date, which represents three different cases. It then generates faulty versions of the file, with each version corresponding to the application of one fault at one of the identified locations for the provided file and using one particular configuration. The injection of the fault follows the procedure explained earlier in Figure 3.1.

Considering that the user of the tool sets no particular restrictions, for a certain list of T target files, a list of B faults, the corresponding number of candidate locations L (that depends on the file and on the fault to be injected) and particular fault configuration C, we will have the following number N of faulty versions:

$$N = \sum_{i=1}^{T} \sum_{j=1}^{B} \mathbf{L}_j * \mathbf{C}_j \tag{3.1}$$

Note that for certain faults, we may have 0 (zero) candidates if the tool finds there is no suitable place to inject that particular fault. The list of predefined faults and the list of candidate locations can obviously be reduced by the tool user to subsets that comply with her/his requirements (e.g., available resources for executing experiments).

It is relevant to mention at this point that, the list of faults that we are using in this work is described in Chapter 4 and that we opted to build this list based on the analysis of reported bugs for a popular Python-based system (OpenStack). We are essentially using the ODC - Orthogonal Defect Classification [IBM, 2013] along with a customized characterization of faults based on the one defined by Durães and Madeira [Durães and Madeira, 2006]. Please refer to Chapter 4 for the complete details.

The resulting faulty files are returned to the Test Manager (**step 4**), and then distributed (**step 5**) among the VM instances (one faulty version per instance), replacing the original component in the SUT that had been previously uploaded in step 2. Each VM instance will hold a different faulty file and in case there are no sufficient instances for all faulty versions, the remaining faulty files will be placed on a waiting queue for later execution.

In **step 6**, the Test Manager orders the previously configured tests to start and waits for each VM instance to conclude. Whenever an instance concludes, tests results are copied back to the test manager and pushed to a results queue. In case tests block, the tool is able to continue from the last succeeded set of tests. Whenever an instance becomes free, the waiting queue is analyzed for any faulty files and if any files are present in the queue, the next file is uploaded and tests begin against the new faulty SUT. The Test Manager is also responsible for resetting the state of the VMs whenever the user finds appropriate (e.g., after running a set of tests).

Periodically, the Report Generator (**step 7**) checks the results queue, processes the results and generates a summary for easier analysis. Each line of the summary holds basic information that allows to easily identify problematic cases (as signaled by the test battery). This information includes basic aspects necessary for understanding the outcome of tests, like the identifier of the test, the fault type identifier, the original code line, the faulty code line, details the outcome of the test, the ratio of tests failed and a summary of the test execution.

The whole process is centered around a pre-built list of faults, but it is fairly easy to add a new fault. In practice, the user needs to extend `AbstractFault` and implement the following methods:

i) `get_fault_id`, which returns a fault identifier (e.g., WVAV);

ii) `count_all`, which provides a total count of the candidate locations where the fault is applicable;

iii) `transform`, which uses source code and the respective AST (which are created when the new fault class is instantiated) to produce modified code along with metadata for reporting (e.g., the affected code line number(s), the original code line(s), the modified code line(s)).

Thus, we mostly reduced the developer's tasks implementing the concept behind the fault in the `transform` method.

Finally, the new class should be added to the tool's `faults` module in order to be dynamically instantiated by the tool and the fault identifier included in the configurations (in case the intention is to run just part of the existent faults).

Chapter 4

Fault Model

In this chapter, we describe how we built the list of software faults used by our tool. We built it based on the analysis of bug reports for OpenStack, a Python-based system. We selected OpenStack due to its popularity and usage in business-critical scenarios. The alternative would be to resort to a more generic fault model [Durães and Madeira, 2006], however possibly less representative of a Python case and not accounting for software faults that may occur in Python, but not in C (e.g., exception handling related faults). We went through the following steps:

i. Selection of a random set of bug reports from OpenStack's public bug tracking system;

ii. Manual classification of each software fault present in a particular bug report, carried out by one researcher (named *researcher1*) and using the Orthogonal Defect Classification (ODC);

iii. Independent verification (i.e., classification by a different researcher, named *researcher2*) of 1/4 of the software faults classified in the previous step;

iv. Manual classification of 1/4 of software faults using an extension to ODC, based on the extension proposed in [Durães and Madeira, 2006].

We began by **randomly selecting** about one fifth of all OpenStack Nova bug reports, that had been marked with 'fixed and released'. Duplicates were filtered out during search and no particular importance filter was applied to the search (i.e., the bugs touch all importance levels defined by OpenStack developers, from unknown to critical). Also, no particular profile was selected (i.e., it may be any kind of bug, it may be a security vulnerability or not).

The data was extracted from OpenStack Nova bug tracking system available at *launchpad.net/nova.* OpenStack is built around several components, in which the Nova component plays a central role, as it provides a way to provision compute instances (i.e., virtual servers). Table 4.1 identifies OpenStack components, as defined in the system's conceptual architecture [OpenStack, 2021], and presents their associated development metrics.

As we can see in Table 4.1, Nova is the oldest OpenStack component and the second largest in terms of lines of code. Since its inception it has been the most actively developed, also gathering a larger community of developers. It also holds the highest number of reported bugs, which is important for analysis activities, as diversity tends to increase with larger

Table 4.1: OpenStack component development metrics.

Component	Lines of Code	# Commits	# Developers	Reported bugs	Creation date
Ceilometer	36927	7035	336	1355	Nov 2012
Cinder	548311	18984	757	5308	Nov 2012
Glance	247202	7214	391	2127	Aug 2011
Heat	233995	15829	392	26	Dec 2012
Horizon	135535	16012	641	5005	Oct 2011
Ironic	203389	11504	406	15	May 2013
Keystone	162253	14611	456	2648	May 2013
Neutron	349789	25091	748	8031	Sep 2011
Nova	**537880**	**58843**	**1076**	**10152**	**Jul 2010**
Sahara	60989	6467	202	10	Oct 2013
Swift	317604	9472	311	1656	Jul 2010
Trove	94180	4573	232	24	Jun 2013

numbers of software defects. We aimed at analysing precisely 2,048 bugs (which represent nearly 20% of 10,152 'fixed and released' bugs, as of July 2021), so we began randomly extracting bug reports and pre-analysing them to understand if they were valid candidates or not. This resulted in the collection of 2,566 bugs from which we eliminated 518 bugs. The reasons for eliminating these bugs were the following:

- Bug target is documentation. We are only interested in bugs that require a code fix.
- Bug fits in OpenStack's wish list. Suggestions or improvements were discarded.
- Bug refers to an error in unit tests or functional tests. Tests are not part of this system, as they are not part of the release or executable code in production.

By the end of this process, we reached a dataset composed of a total number of 2,048 unique bug reports, from which 182 had been marked with a Common Vulnerabilities and Exposures (CVE) identifier. **Manual classification using ODC** was then performed by an Early Stage Researcher, named *researcher1*, against each of the 2,048 reported software defects. In this work, we are using the 'Defect Type' and 'Qualifier' ODC attributes (the useful attributes for our purpose), although in our detailed results available at [Marques et al., 2021], the reader may find the data for other attributes for which we had sufficient information to perform the classification (i.e., Activity, Trigger, Impact) and also the severity associated with each reported bug classified using the CRASH scale [Kropp et al., 1998].

Table 4.2 shows the distribution found for the ODC 'Defect Type' attribute in our dataset and Table 4.3 shows the distribution found for the ODC 'Qualifier' attribute. We can observe the distinct prevalence of the different values, which is useful to understand which types of bugs are frequent.

Table 4.2: ODC defect type results

Defect type	Percentage
Interface/O-O Messages	17.63
Assignment/Initialization	13.53
Algorithm/Method	45.75
Function/Class/Object	13.13
Checking	7.47
Timing/Serialization	0.83
Relationship	1.66

To *verify the classification* of the software defects and to understand the overall quality of the classification process, we asked an Early Stage Researcher (named *researcher2*) to perform an independent classification of a subset of defects composed of 25% of the software

Table 4.3: ODC qualifier results

Qualifier	Percentage
Incorrect	52.83
Missing	42.73
Extraneous	4.44

defects present in the whole dataset (i.e., 512 bugs). As this is a time consuming activity, we used this number of bugs, essentially due to human resource limitations. However, it should be sufficient to provide a first indicator of the quality of the dataset. Note that this is nearly the size of the dataset found in [Durães and Madeira, 2006], which is composed of 668 bugs.

Table 4.4 shows the detailed outcome of the verification procedure for the Defect Type attribute and Table 4.5 presents the same for the Qualifier attribute. In each of the matrices, each cell holds the total number of defects marked with a particular attribute value, whereas the values that are read in the rows represent the outcome of the verification carried out by *researcher2*. In the diagonal, we mark the true positives (the defects in which both researchers agreed).

Table 4.4: Verification results for Defect Type.

	A/I	C	A/M	F/C/O	T/S	I/OM	R
Assignment/Initialization (A/I)	60	2	15	2	0	9	1
Checking (C)	0	25	2	0	0	0	0
Algorithm/Method (A/M)	2	6	233	25	1	19	0
Function/Class/Object (F/C/O)	1	0	6	39	0	3	1
Timing/Serialization (T/S)	0	0	0	0	0	0	0
Interface/O-O Messages (I/OM)	3	3	5	0	1	37	1
Relationship (R)	1	0	5	0	0	2	2

Table 4.5: Verification results for Qualifier.

	Incorrect	Missing	Extraneous
Incorrect	206	22	3
Missing	29	233	1
Extraneous	3	0	15

Despite the several cases of disagreements between both researchers, the reality is that the main cases of disagreement are just a few. For Defect Type the most frequent disagreements (considering the absolute numbers) are between bugs marked as Assignment/Initialization by *researcher1* but marked as Algorithm/Method by *researcher2* and also between Algorithm/Method marked by *researcher1*, which were marked as F/C/O or I/OM by *researcher2*.

We also analyzed the inter-rater agreement using Cohen's Kappa coefficient (k), which characterizes the agreement between two raters that classify items in mutually exclusive categories [Cohen, 1960]. The definition of k is given by the following Equation 4.1:

$$k = \frac{p_o - p_c}{1 - p_c} \tag{4.1}$$

where p_o is the relative observed agreement between raters (i.e., accuracy) and p_c is the probability of agreement by chance. If both raters fully agree, then $k = 1$, if there is no agreement beyond what is expected by chance, then $k = 0$, and, finally, a negative value reflects the cases where agreement is actually worse than random choice. The following qualitative terms apply for the following values of k: *poor* when $k < 0$, *slight* when $0 <= k <= 0.2$, *fair* when $0.21 <= k <= 0.40$, *moderate* when $0.41 <= k <= 0.60$, *substantial* when $0.61 <= k <= 0.80$, and finally *almost perfect* when $0.81 <= k <= 1.00$ [Landis and Koch, 1977]. Table 4.6, presents the accuracy results (i.e., the number of true positives divided by the total number of defects) for both ODC attributes being considered, and the respective Kappa value.

Table 4.6: Accuracy and Cohen's Kappa Agreement between researcher1 and researcher2.

	Accuracy	Cohen's Kappa	Term
Defect Type	0.77	0.74	Substantial
Qualifier	0.89	0.83	Almost perfect

Table 4.6 shows that we obtained at least substantial agreement values for both attributes, which increases our confidence in the overall quality of the dataset.

Finally, we performed an **extended review** of 25% of the bugs (512 bugs) based on the extension to ODC proposed by Durães and Madeira [Durães and Madeira, 2006]. We actually augmented the authors extension, mostly due to the fact that certain types of bugs were not observed in their work with C applications (e.g., faults related with exception handling, which does not exist in C, or faults related with Object Oriented Programming). Table 4.7 presents our extension to ODC and the resulting classification for these 512 bugs, including their prevalence in absolute numbers and a marker indicating if the fault is new (i.e., with respect to [Durães and Madeira, 2006]).

The data in Table 4.7 indicate, in a detailed manner, which types of faults are more prevalent which is useful for carrying dependability evaluation experiments, namely those based on the definition of fault models - in this case for Nova Compute, the core component of OpenStack. There are two relevant aspects to mention regarding the data in Table 4.7.

- More than one fourth of the faults identified are new, with respect to the ODC extension proposed in [Durães and Madeira, 2006], which reflects the specificity of the context and also the number and type of analysed software defects.
- Of the new faults, more than half are directly related to Python features, in particular those related with exception handling, and object-oriented features like the use of constructors and inheritance.

Listings 1, 2, and 3 illustrate three cases of faults that are specific to the Python context. The listings respectively present particular cases of each of the three possible values for the Qualifier ODC attribute: extraneous, missing, and wrong.

As we can see, Listing 1 represents a very simple case of an *extraneous* defect, where the code is modified to handle a runtime exception that should not be caught at that particular location (i.e., in line 9). Listing 2 calls a super class constructor in line 5 (whose function would be to inject a reference to a database driver), and we can see that the faulty code omits such call. Finally, Listing 3 presents the case where an exception is being handled (ValueError in line 5) and we can see that the corresponding faulty version tries to catch an exception that is not related with the original one (i.e., SystemExit in line 16). Please refer to our replication package at [Marques et al., 2021], which includes one example of the application of each of the fault types to the Nova Compute API.

Overall, the identified types of faults reflect the overall prevalence of defects found in Nova Compute and, in this sense, there may be more faults that relate to the context

Table 4.7: Fault types based on the ODC extension in [Durães and Madeira, 2006].

Defect Type	Qualifier	ID	Description	#	New
Algorithm / Method	Extraneous	EALD	Extraneous Algorithm - Small sparsed modifications	2	✓
		EIFS	Extraneous Function Call	3	
	Missing	MCA	Missing iteration construct around statements	4	
		MFC	Missing function call	25	
		MIEA	Missing If construct plus else plus statements around statements	2	
		MIEB	Missing if construct plus statements plus else before statements	1	
		MIES	Missing if-else construct plus statements	5	
		MIFS	Missing if construct plus statements	42	
		MLPA	Missing small and localized part of the algorithm	28	
		MLPL	Missing large part of the algorithm	5	
		MLPS	Missing sparcely spaced parts of the algorithm	18	
	Wrong	WALD	Wrong Algorithm - small sparce modifications	44	
		WALR	Wrong Algorithm - code was misplaced	9	
		WASL	Wrong Algorithm Small Localised modifications	59	✓
		WFCD	Wrong function called with different paramenters	7	
		WFCS	Wrong function called with same parameters	11	
Assignment / Initialization	Extraneous	EVAV	Extraneous variable assignment using another variable	3	
	Missing	MVAE	Missing variable assignment using an expression	13	
		MVAV	Missing variable assignment using a value	7	
		MVIE	Missing variable initialization using an expression	1	
		MVIV	Missing variable initialization using a value	2	
	Wrong	WIDS	Wrong string in initial data	13	
		WIDSL	Wrong string in initial data - missing one char	2	
		WSUT	Wrong data types or conversion used	1	
		WVAE	Wrong arithmetic expression used in assignment	3	
		WVAL	Wrong logical expression used in assignment	2	
		WVAM	Miss by one value assigned to variable	0	
		WVAV	Wrong value assigned to variable	15	
		WVIV	Wrong value used in variable initialization	5	
Checking	Extraneous	EIA	Extraneous if construct around statements	3	✓
	Missing	MIA	Missing if construct around statements	17	
		MLAC	Missing "AND EXPR" in expression used as branch condition	6	
		MLOC	Missing "OR EXPR" in expression used as branch condition	1	
	Wrong	WLEC	Wrong logical expression used as branch condition	9	
Function / Class / Object	Extraneous	EFCT	Extraneous functionality	3	
	Missing	MFCT	Missing functionality	33	
	Wrong	WALL	Wrong algorithm - large modifications	31	
Interface / O-O Messages	Extraneous	ECEFL	Extraneous caught exceptions for function call	2	✓
		EPFC	Extraneous parameter in function call	4	✓
	Missing	MCEFL	Missing caught exceptions for function call	23	✓
		MPFC	Missing parameter in function call	13	
		MRS	Missing return statement	2	
	Wrong	WCEFL	Wrong caught exceptions for function call	2	✓
		WPFL	Wrong value used in parameter of function call	10	
		WPFO	Wrong paramenter order in function call	1	
		WPFV	Wrong variable used in parameter of funtion call	10	
		WRV	Wrong return call	2	
Relationship	Missing	MICAFC	Missing Object construct around function call	1	✓
		MSC	Missing Super Classe construct call	1	✓
	Wrong	WASC	Wrong arguments in super construct	2	✓
		WCE	Wrong String encoding	1	✓
		WDFSC	Wrong derivation from super class	1	✓
Timing / Serialization	Missing	MAS	Missing Serialization calls	1	✓
	Missing	MSFC	Missing Synchronization calls	1	✓

of the Python programming language. Still, in the analysed project, we found them to not be representative and, as such are not part of this set of faults. Also, it is worthwhile mentioning that current mutation testing tools like Mutatest [Kepner, 2021], Mutpy [Hałas, 2021], or Cosmic Ray [AS, 2021] do not capture some of the identified faults in this work, namely the higher level ones (e.g., missing synchronization calls, missing object construct around functional call). At the same time, our tool does not implement some of the operators provided by current tools, as they were not present in our software defects dataset (e.g., break continue replacement) and, in this sense, were found to not be representative.

Overall, the list in Table 4.7 aims at characterizing real mistakes made by developers, in a fine granularity (e.g., missing if construct plus else plus statements around statements;

```python
### Original code:
if instance.locked and not context.is_admin:
    raise exception.InstanceIsLocked(instance_uuid=instance.uuid)

### Faulty code:
try:
    if instance.locked and not context.is_admin:
        raise exception.InstanceIsLocked(instance_uuid=instance.uuid)
except:
    aux = 'extraneous caught exception'
```

Listing 1: Extraneous caught exceptions for function call (ECEFL).

```python
### Original code:
def __init__(self, rpcapi=None, servicegroup_api=None):
    self.rpcapi = rpcapi or compute_rpcapi.ComputeAPI()
    self.servicegroup_api = servicegroup_api or servicegroup.API()
    super(HostAPI, self).__init__()


### Faulty Code:
def __init__(self, rpcapi=None, servicegroup_api=None):
    self.rpcapi = rpcapi or compute_rpcapi.ComputeAPI()
    self.servicegroup_api = servicegroup_api or servicegroup.API()
```

Listing 2: Missing super class constructor call (MSC).

missing variable initialization using an expression), and within the context of this work. The tool presented in this thesis implements all the faults shown in Table 4.7.

```python
### Original code:
def _get_image_meta_obj(image_meta_dict):
    try:
        image_meta = objects.ImageMeta.from_dict(image_meta_dict)
    except ValueError as e:
        # there must be invalid values in the image meta properties so
        # consider this an invalid request
        msg = _('Invalid image metadata. Error: %s') % six.text_type(e)
        raise exception.InvalidRequest(msg)
    return image_meta

### Faulty code:
def _get_image_meta_obj(image_meta_dict):
    try:
        image_meta = objects.ImageMeta.from_dict(image_meta_dict)
    except SystemExit as e:
        # there must be invalid values in the image meta properties so
        # consider this an invalid request
        msg = _('Invalid image metadata. Error: %s') % six.text_type(e)
        raise exception.InvalidRequest(msg)
    return image_meta
```

Listing 3: Wrong caught exceptions for function call (WCEFL)

This page is intentionally left blank.

Chapter 5

OpenStack Case study

In order to illustrate the capabilities of our tool, we selected a popular Python application to carry out a fault injection campaign. The application selected was the latest stable version (at the time of the experiments) of OpenStack – version *Ussuri* (*openstack.org*), a cloud management platform used nowadays to support business-critical systems. We aimed at its core component Nova (which is bundled in version 21.0.0 in the *Ussuri* version of OpenStack). Nova's key component is Compute, which aims at providing massively scalable, on demand, self-service access to compute resources. The Compute file *api.py* holds the main operations that support the Compute service and that we selected to be the target of fault injection. This file is composed of 3,870 source lines of code resulting in 2,975 logical lines of code, spread across 233 blocks (i.e., classes, functions, methods), and for which McCabe's cyclomatic complexity [McCabe, 1976] reaches 17.0.

Our tool was configured with the scripts to remotely deploy OpenStack Ussuri and run OpenStack developers' full battery of tests, which comprise unit, functional, and integration tests.

Using the resources made available by INCD, we setup 5 virtual machines running Ubuntu Server 16.04 LTS with 8 core CPU, 16GB of RAM, and 50GB of storage with the main intention of decreasing the total time required to run the time-intensive experiments.

We implemented all 54 fault types shown earlier in Table 4.7 and, as a result of the fault injection process, we ended up with a total of 11,309 faulty versions to test. These versions were the result of injecting most of the faults in all possible candidate locations. The exception is the set of faults of type EALD, MLPA, MLPS, WALD, WALR, WASL, EIA for which we injected a single fault per code block (i.e., per method) because these faults can result in an undetermined amount of injection candidates. This is very clear in the particular case of EIA (i.e., *extraneous if construct around statements*), which is a fault of type *checking*, where there is the presence of extraneous code, which can obviously take numerous forms and which we implemented using a basic case as example. In this particular case of EIA, we randomly select an interval of lines of code within the method and add an `if` construct around them, with the condition validating to `false` (i.e., so that it affects the code execution). Notice also that, in Table 5.1, the fault type WCE (wrong string encoding) does not have any injected locations due to the fact that there is no string encoding conversion in the code file under test. The faulty versions were then integrated in the deployment of OpenStack setup in the virtual machines (as illustrated earlier in the description of Figure 3.2).

With the faulty deployment in place, our tool started the `tox` command, which creates virtual environments, populates them with dependencies and runs all of the tests associated with OpenStack's Continuous Integration system. The tool runs 17,344 unit tests, 2,272 functional tests, and 2,197 integration tests, for which we analyse the outcome in the next paragraphs.

We performed more than 245 million test case executions and first identified the cases where the fault injected was not detected by any of the tests ran by the continuous integration system used by OpenStack developers. Table 5.1 presents the results, in particular the number of faults injected by fault type and, from those, how many were undetected by OpenStack Unit tests, Functional tests, and Integration tests.

Table 5.1: Experimental results overview.

Fault Type	Fault	Injected Locations	Undetected by Unit		Undetected by Functional		Undetected by Integration		Undetected by all	
			#	%	#	%	#	%	#	%
Algorithm/Method	EALD	233	0	0%	10	4%	233	100%	0	0%
	EIFS	501	0	0%	261	52%	498	99%	0	0%
	MCA	53	0	0%	11	21%	53	100%	0	0%
	MFC	1310	0	0%	583	45%	1306	100%	0	0%
	MIEA	63	0	0%	24	38%	63	100%	0	0%
	MIEB	63	0	0%	17	27%	63	100%	0	0%
	MIES	63	0	0%	9	14%	61	97%	0	0%
	MIFS	293	0	0%	132	45%	293	100%	0	0%
	MLPA	233	0	0%	77	33%	233	100%	0	0%
	MLPL	129	0	0%	26	20%	129	100%	0	0%
	MLPS	233	0	0%	12	5%	233	100%	0	0%
	WALD	233	0	0%	0	0%	222	95%	0	0%
	WALR	233	0	0%	7	3%	232	100%	0	0%
	WASL	233	0	0%	0	0%	213	91%	0	0%
	WFCD	297	0	0%	90	30%	274	92%	0	0%
	WFCS	382	0	0%	0	0%	367	96%	0	0%
Assignment/Initialization	EVAV	190	0	0%	27	14%	190	100%	0	0%
	MVAE	39	0	0%	0	0%	39	100%	0	0%
	MVAV	7	1	14%	3	43%	7	100%	1	14%
	MVIE	213	35	16%	44	21%	211	99%	19	9%
	MVIV	31	4	13%	9	29%	31	100%	1	3%
	WIDS	9	6	67%	8	89%	9	100%	6	67%
	WIDSL	9	4	44%	4	44%	9	100%	4	44%
	WSUT	13	1	8%	4	31%	13	100%	1	8%
	WVAE	26	8	31%	13	50%	25	96%	5	19%
	WVAL	19	3	16%	6	32%	18	95%	1	5%
	WVAM	2	0	0%	0	100%	2	100%	0	0%
	WVAV	5	0	0%	4	80%	5	100%	0	0%
	WVIV	24	9	38%	11	46%	24	100%	5	21%
Checking	EIA	233	0	0%	35	15%	233	100%	0	0%
	MIA	356	67	19%	121	34%	355	100%	45	13%
	MLAC	109	0	0%	62	57%	109	100%	0	0%
	MLOC	26	0	0%	22	85%	26	100%	0	0%
	WLEC	260	38	15%	93	36%	258	99%	26	10%
F/C/O	EFCT	5	0	0%	0	0%	5	100%	0	0%
	MFCT	233	0	0%	63	27%	204	88%	0	0%
	WALL	5	0	0%	0	0%	4	80%	0	0%
Interface/O-O Messages	ECEFL	233	0	0%	64	27%	232	100%	0	0%
	EPFC	1238	0	0%	285	23%	1150	93%	0	0%
	MCEFL	76	8	11%	34	45%	76	100%	7	9%
	MPFC	969	172	18%	383	40%	966	100%	125	13%
	MRS	187	16	9%	27	14%	186	99%	9	5%
	WCEFL	82	0	0%	66	80%	79	96%	0	0%
	WPFL	49	0	0%	27	55%	49	100%	0	0%
	WPFO	711	0	0%	201	28%	645	91%	0	0%
	WPFV	1132	0	0%	445	39%	1086	96%	0	0%
	WRS	187	0	0%	59	32%	186	99%	0	0%
Relationship	MICAFC	25	0	0%	13	52%	25	100%	0	0%
	MSC	3	1	33%	1	33%	3	100%	1	33%
	WASC	3	0	0%	0	0%	3	100%	0	0%
	WCE	0	-	-	-	-	-	-	-	-
	WDFSC	5	0	0%	3	60%	5	100%	0	0%
T/S	MAS	1	0	0%	1	100%	1	100%	0	0%
	MSFC	42	41	98%	35	83%	42	100%	34	81%

Our experiments revealed a total of 290 cases of faulty versions passing undetected through

OpenStack's battery of tests. Overall, we observe that the unit tests are the most effective in detecting the injected faults (capturing 96.3% of all injected faults), followed by the functional tests which, on their own, detect about two thirds of the faults (i.e., 69.6%). The integration tests show a residual performance, being able to detect 2.9% of the faults. Two thirds of the faults that escape the unit tests also elude detection by the functional tests. Of the 293 faults that reach the integration testing level, only 1% are captured at this level.

Table 5.2 presents the results from the *Defect Type* point of view. As we can see, in

Table 5.2: Experimental results grouped by Defect Type.

Fault Type	Injected	Undetected by Unit		Undetected by Functional		Undetected by Integration		Undetected by all	
	Locations	#	%	#	%	#	%	#	%
Algorithm/Method	4552	0	0%	1259	28%	4473	98%	0	0%
Assignment/Initialization	587	71	12%	133	23%	583	99%	43	7%
Checking	984	105	11%	333	34%	981	100%	71	7%
F/C/O	243	0	0%	63	26%	213	88%	0	0%
Interface/O-O Messages	4864	196	4%	1591	33%	4655	96%	141	3%
Relationship	36	1	3%	17	47%	36	100%	1	3%
T/S	43	41	95%	36	84%	43	100%	34	79%

Table 5.2, the Timing/Serialization faults tend to pass undetected by all tests, which reflects the general difficulty of detecting this kind of faults. Assignment/Initialization and Checking are the next contributors, much at a much lesser extent. Also, it is worthwhile mentioning the fact that the tests were able to capture all Algorithm/Method and all Function/Class/Object faults, which are all detected at the unit test level. The unit tests are also quite effective with Interface/O-O Messages and Relationship faults. Functional tests show slightly better performance with Assignment/Initialization, Function/Class/Object, and Algorithm/Method, but generally perform in a balanced manner (with the exception of Timing/Serialization defects). Integration tests are generally ineffective in detecting the various types of faults.

Table 5.3 presents the results grouped by the *Qualifier* attribute. By analysing the results

Table 5.3: Experimental results grouped by Qualifier.

Qualifier	Injected	Undetected by Unit		Undetected by Functional		Undetected by Integration		Undetected by all	
	Locations	#	%	#	%	#	%	#	%
Extraneous	2633	0	0%	682	26%	2541	97%	0	0%
Incorrect	3919	69	2%	1041	27%	3728	95%	48	1%
Missing	4757	345	7%	1709	36%	4715	99%	242	5%

from the point of view of the *Qualifier* attribute, we mostly see that 'missing' faults contribute the most to the number of undetected faults, while 'extraneous' do not contribute at all. From the testing level perspective, we again see the high effectiveness of the unit tests, in particular capturing the 'extraneous' faults, which are totally detected at this testing level.

We further analysed 25% (72) of the total undetected cases (i.e., a sample that would allow us to obtain insights regarding the quality of the test suite). We performed this analysis focusing on two key aspects: i) understanding if the injected fault could really affect the regular behavior of the application (via code inspection); ii) understanding which action should be taken so that the particular (probable) fault is caught by the tests (e.g., modifying an existent unit test or creating a new one). We did the individual analysis of each undetected case by inspecting calls (made by the different tests) to the affected area, thus identifying the test or set of tests involved, and then analysing them for a potential modification or need for a new test.

Each corrective action was marked with one of three numeric values in Table 5.4, characterizing the associated difficulty (1 for *trivial*, 2 for *moderate*, 3 for *complex*) and described as follows. We consider a correction to be *trivial* if it involves validating if a returned value is the expected, which includes asserting that fields of a returned object hold non-null values, or, in the case the function under test throws exceptions, verifying if an exception raised is the expected. A correction marked with *moderate* also involves validating specific branches in the code (e.g., validating that certain inner function calls are executed and also return expected values), by providing diverse input values to the functions under test. This definition aligns with the term 'simple path' used in the ODC *trigger* attribute [IBM, 2013]. Finally, a correction marked with 'complex' aims at executing specific multiple combinations of code paths (i.e., execute multiple branches under several different conditions). This definition aligns with the term 'complex path' used in the ODC *trigger* attribute [IBM, 2013]. In practice, this type of fix may require substantial refactoring of existing tests or a significant amount of effort to create a new test.

In what concerns corrections, we set the focus at the unit test level, which allows early disclosure of defects, preventing them to reach later testing phases (e.g., integration). We did, however, consider to correct functional or integration tests, in case the difficulty of the correction would be *complex*. Table 5.4 overviews the analysed problems (i.e., the sample of 25% of the total number of undetected faults) and necessary corrections. All detailed results are available online at [Marques et al., 2021].

As we can see in Table 5.4, all problems disclosed could be solved directly at the unit testing level (we did not identify any particularly difficult case that should be handled otherwise). Of the 72 analysed problems, about half of them (i.e., 34) are marked with *trivial*, 38% with *moderate*, and only 15% are considered to be *complex*. More than half (i.e., 57%) can be fixed by adding a new unit test(s). The most common problems detected are related with wrong exceptions being raised and incorrect values being passed as parameter to other functions that might result in incorrect values being returned. In the next paragraphs, we highlight a few illustrative cases including the action required to detect each case.

Injected fault 4 (WIDS) modifies a string assigned to a global variable. This variable is used to raise an exception based on a string passed as parameter. The fault causes an incorrect exception to be raised and, because there are no unit tests for the given function, the fault is not detected. To deal with this issue, a *trivial* correction would consist in creating a new unit test that verifies the raised exception against the value passed as parameter.

Injected fault 179 (MIA) affects a method that receives a set of VM instance attributes that are used in a lookup operation (e.g, system metadata, security groups). In case `None` is provided to the method, the defaults are used, otherwise the method arguments are used. The injected fault removed the `if` condition that controls this decision; thus, the default values are always used making it impossible to fetch specific details about a certain instance, which breaks the intended functionality. This passes undetected by the unit tests because the method is tested precisely with values that are equal to the defaults. To cover this case, a test case using diverse combinations of the different attributes involved should be added (the correction is marked with *moderate*).

In the case of **Injected fault 560 (WLEC)**, a branch condition is modified causing a variable holding an attachment *id* to be set incorrectly, this variable is later passed as parameter to a function that detaches a volume (a block-level storage device that can be attached to an instance). The tests do not detect this because there is no validation of the parameters passed to the function. To cover this case, the tests should be modified to test complex paths and branch execution using data input diversity. This diversity should

Table 5.4: Analysed problems and necessary corrections.

Type	Fault	Id	Problem summary	Correction summary	Difficulty
Assignment/Initialization	MVAV	1953	Incorrect exception log	Modify - Validate exception message	1
	MVIE	1724	Incorrect exception log	Modify - Validate exception message	1
		1779	Incorrect field initialization	Modify - Validate all object's fields	1
		1821, 1834	Incorrect value passed to function call	Add - Test multiple input variations	2
		1852	Incorrect value passed to function call	Modify - validate inputs to mock function	1
	MVIV	1938	Incorrect exception raised	Add - Add test to validate exception raised	1
	WIDS	2, 3, 4	Incorrect exception raised	Add - Add test to validate exception raised	1
		1	Incorrect value passed to function call	Add - Test multiple input variations	2
		6, 7	Incorrect exception log	Modify - Validate exception message	1
	WIDSL	1975, 1976, 1977	Incorrect exception raised	Add - Add test to validate exception raised	1
	WSUT	1981	Incorrect exception raised	Add - Test multiple input variations	2
	WVAE	2004	Exception not thrown	Add - Test multiple input variations	2
		2001	Incorrect value passed to function call	Modify - Validate all object's fields	1
		2013	Incorrect exception log	Modify - Validate exception message	1
	WVAL	2033	Incorrect value passed to function call	Modify - validate inputs to mock function	1
	WVIV	2039, 2042	Incorrect log message	Add - Test multiple input variations	2
		2041	Return an incorrect value	Modify - validate inner function output	2
		2045	Return object with incorrect value	Modify - Validate returned object's values	1
		2065	Return an incorrect value	Modify - Add validation for complex paths and branch execution	3
Checking	MIA	159	Extraneous exception log	Modify - Add testing for complex paths and branch execution	3
		66	Return an incorrect value	Add - Test multiple input variations	2
		64	Extraneous function call	Add- Assert method not called	2
		165, 351	Functionality broken	Add - Add validation for complex paths and branch execution	3
		150	Extraneous function call	Modify - Add validation for complex paths and branch execution	3
		326, 321, 36	Return an incorrect value	Add - Test multiple input variations	2
		179	Functionality broken	Add - Test multiple input variations	2
		188	Incorrect value passed to function call	Modify - validate inputs to mock function	1
		98	Extraneous exception log	Modify - Add validation for complex paths and branch execution	3
		265	Incorrect exception raised	Add - Test with multiple mock outputs	2
	WLEC	377	Incorrect log message	Modify - Validate log output	1
		439	Return an incorrect value	Modify - Add validation for complex paths and branch execution	3
		560	Incorrect value passed to function call	Modify - Add validation for complex paths and branch execution	3
		593	Incorrect exception raised	Add - Test with multiple mock outputs	2
		606	For loop interrupted	Add - Test multiple input variations	2
		614	Cache reset	Add- Assert method not called	2
		546	Extraneous function call	Add- Assert method not called	2
Interface/O-O Messages	MCEFL	630, 645, 683	Incorrect exception raised	Add - Add test to validate exception raised	1
		648	Incorrect exception raised	Add - mock function to raise exception	1
		641, 642	Exception raised	Add - Add test to validate exception raised	1
	MPFC	1015, 1018, 1667	Incorrect exception log	Modify - Validate exception message	1
		1036	Return object with incorrect value	Modify - Validate all object's fields	1
		1447	Function not called	Modify - validate inputs to mock function	1
		1124	Functionality broken	Add - Test multiple input variations	2
		1004, 1511	Incorrect log message	Modify - Validate log output	1
		1530, 1081	Functionality broken	Modify - validate inputs to mock function	1
	MRS	2140, 2142, 2222	Functionality broken	Add - Test with multiple mock outputs	2
		2197, 2198, 2199	Incorrect value passed to function call	Modify - Add validation for complex paths and branch execution	3
		2214	Return an incorrect value	Add - Test with multiple mock outputs	2
T/S	MSFC	718, 719, 720	Functionality broken	Add - Test multiple input variations	2

allow all cases of the `if` condition to be exercised, which combined with the validation of the parameters in the inner function calls, would detect the fault. This correction is marked with *complex* due to its nature, which involves exercising complex paths.

In **Injected fault 645 (MCEFL)** a try-except block is removed, leaving a set of instructions responsible for fetching a volume snapshot (which is a copy of a volume at a specific moment) from the database, without any exception handling. This can result in an exception being raised (not caught by any tests, which are not prepared for this kind of exceptional behavior). Raising an unhandled exception in this context will result in a stack trace logged, instead of the programmed error message. To cover this issue, new tests cases should be added to trigger and assert that the expected exception is raised (i.e., a *trivial* correction).

Injected fault 718 (MSFC) removes a wrapper responsible for checking the VM instance state and confirming it is not in a `locked` mode. The fault breaks the functionality because it is possible to perform actions like shelving (stop instance and take a snapshot), that should not be allowed. This fault is not detected by the unit tests because the instances provided as parameter to the function have the `locked` state set to `false`. To cover this case new tests should be added using input diversity for the instance locked state (i.e., a *trivial* correction).

Injected fault 1511 (MPFC) removes the VM instance parameter on a log function call, resulting in an incomplete log message, which may impair maintainability. The fault is not caught by the unit tests because there is no validation on the log output. To cover this case, the tests should be modified to validate if the log output is the expected (i.e., a trivial correction).

It is worthwhile mentioning that although the use of simple techniques like coverage analysis may be helpful in detecting some of the identified issues (e.g., 560, 179), in many other cases they are not useful (e.g., 614, 641). This emphasizes the need of specialized approaches, such as the one used in this thesis, when reliability is a critical requirement.

Chapter 6

Fault Reduction Analysis

In the previous chapter, we showed how our approach can be used to detect problems in the system under test and especially the limitations in the OpenStack test suite. However, in situations where time or resources are limited, it may not be possible to apply the approach using its full configuration. This is because, depending on the system, the application of our fault model can result in a very large number of faulty code versions that must then be passed through the entire OpenStack's (or some other system being tested) test suite.

During our experimental evaluation, the 54 implemented faults resulted in the generation of 11,309 faulty versions. Running the entire OpenStack test suite over the faulty versions with the tests being distributed over 5 virtual machines (in the conditions mentioned in the previous chapter), takes about 235 days. Depending on the tester's priorities, this can make the approach very expensive to apply, which is a well known problem in the mutation testing community [Pizzoleto et al., 2019, Zhang et al., 2013, Praphamontripong and Offutt, 2017, Byoungju and Mathur, 1993, Namin et al., 2008].

Previous works on mutation testing have targeted the problem of the cost of test execution, by reducing the number of generated faulty versions (i.e., mutants) [Pizzoleto et al., 2019, Zhang et al., 2013, Praphamontripong and Offutt, 2017, Byoungju and Mathur, 1993, Namin et al., 2008]. In the next paragraphs, we present the application of a well established technique that not only allows reducing the total time cost of applying our approach but also allows selecting a set of *sufficient fault types* (i.e., the faults that are most effective in disclosing problems) in an informed manner.

In [Namin et al., 2008], the authors propose an approach to reduce the number of injected faults by selecting a sufficient subset of fault types that can still accurately measure the effectiveness of a test suite (i.e., they are able to disclose a significant number of issues in the test cases). The approach is based on Least Angle Regression (LARS) [Efron et al., 2004], which is an algorithm for fitting linear regression models to high-dimensional data that allows to estimate which variables to select and their coefficients. The algorithm input variables are called *predictors* and will determine the *response* by linear combination. For a given problem with multiple input variables (in our case, each variable would correspond to one fault type) the algorithm will try to select which *predictors* have more influence over the *response*, based on the provided sample as training data. The algorithm's outputs are the coefficients for each of the *predictors*, which indicate the weight that each *predictor* has over the *response*, the higher the absolute value the more correlated the *predictor* and the *response* are.

The authors in [Namin et al., 2008] propose that *test suite effectiveness* can be measured by computing a *mutation adequacy ratio* (AM), as described by Equation 6.1:

$$AM(P, S) = KM(P, S)/NM(P) \tag{6.1}$$

In Equation 6.1, P refers to the program under test and S the test suite. KM is the total number of faulty versions generated by all the fault types that are killed by the test suite S (i.e., versions where the fault is detected by at least one of the tests) and NM the total number of faulty versions generated by all faults. As we can see, AM is a value greater than or equal to 0 and less than or equal to 1, where higher values represent more effective test suites. A value of AM that is equal to 1 means that all injected software faults are detected by the test suite, whereas $AM = 0$ means that no faults are detected by the test suite.

For defining the input data, we randomly selected 128 groups composed of 1,130 faulty versions each (10% of all the injected bugs) from the tests executed and calculated the *mutation adequacy ratio per fault type* (AM_i), according to Equation 6.2 (explained in the next paragraph) and then calculate AM (the mutation adequacy ratio, i.e., the *response*), as described in Equation 6.1, for each of the 128 groups. We used a statistical measure of goodness of fit (R^2), as described later in this chapter, to determine the number of groups (128) and faulty versions (1,130). We ran the LARS algorithm with 64, 128, 256 and 512 groups of faulty versions using 10%, 20% and 50% of the total amount of faulty versions tested (all combinations were tested and repeated 10 times) for each of them and discovered that groups of 128 and 10% of the total bugs yielded the best results with R^2 averaging to 0.825 with the corresponding standard deviation being 0.017 (for the 10 runs). The LARS algorithm will use each of the groups to calculate the relation between the *predictors* (AM_i) and the *response* (AM), as follows:

$$AM_i(P, S) = KM_i(P, S)/NM_i(P) \tag{6.2}$$

In Equation 6.2, KM_i is the number of faulty versions killed by the test suite for a given fault type i, whereas NM_i is the total number of faulty versions tested for the same fault type. In our case, we have a total of 54 input variables (i.e., the 54 different fault types), for which we will determine the corresponding AM_i. The *LARS output*, expressed by Equation 6.3, will allow us to predict the effectiveness of the test suite used in our case:

$$AM \cong k + c_1 AM_1 + c_2 AM_2 + \cdots + c_j AM_j \tag{6.3}$$

Graphically, Equation 6.3 can be represented by a line that represents the relation between the *predicted AM* (x axis) with the subset of fault types and the *true value of AM* (y axis). k (which is named intercept and is also an output of the LARS algorithm) represents the interception of the line with the y axis and $c1, ..., cj$ are the coefficients for each selected fault type. Fault types with coefficients equal to zero have no correlation with the *response*, which means that injected faults of that type do not have impact in the mutation adequacy score (AM) of the solution and, as such, are not part of the sufficient subset necessary to accurately measure the test effectiveness.

In the case of our experiments, the execution of the LARS algorithm identified the 16 fault types that precisely generate issues that the OpenStack test suite does not capture, with the corresponding coefficients for each fault type and *intercept* value. The results are presented in Table 6.1.

Table 6.1: Sufficient fault types.

Fault type	Description	Coefficient	Number of faulty versions
MPFC	Missing parameter in function call	0.0895868	969
MIA	Missing if construct around statements	0.03099973	356
WLEC	Wrong logical expression used as branch condition	0.02339605	260
MVIE	Missing variable initialization using an expression	0.02085302	213
MRS	Missing return statement	0.01415938	187
MCEFL	Missing caught exceptions for function call	0.00721201	76
MVIV	Missing variable initialization using a value	0.00460843	31
MSFC	Missing Synchronization calls	0.00341365	42
MVAV	Missing variable assignment using a value	0.00236844	7
WVIV	Wrong value used in variable initialization	0.00218223	24
WVAL	Wrong logical expression used in assignment	0.00167488	19
WIDS	Wrong string in initial data	0.00154257	9
WVAE	Wrong arithmetic expression used in assignment	0.00147688	26
WIDSL	Wrong string in initial data - missing one char	0.0009434	9
WSUT	Wrong data types or conversion used	0.00080991	13
MSC	Missing Super Classe construct call	0.00048964	3
(Intercept)		0.79422690	

It is worthwhile mentioning that three of the new fault types that are specific to this work, namely MCEFL, MSFC, MSC, are present in this set and actually account for 14.48% (42) of the 290 faulty versions that passed undetected through the tests. We also observe that, at the time of writing, the set of mutation tools discussed in Chapter 2 would be able to generate about half of the fault types identified in Table 6.1. We further analysed the adequacy of this method by calculating the R^2 coefficient [Efron et al., 2004]. As previously mentioned, R^2 is a statistical measure of goodness of fit, representing how close the data are to the fitted regression line and is defined by Equation 6.4 (extracted from [Efron et al., 2004]):

$$R^2 = 1 - \frac{\sum_{i}^{n}(yTrue_i - yPred_i)^2}{\sum_{j}^{n}(yTrue_j - avg\_yTrue))^2} \tag{6.4}$$

In Equation 6.4, the numerator represents the residual sum of squares of the differences between true and predicted values, whereas the denominator holds the total sum of squares of the differences between the true values and the average of the true values. The best possible output value is 1 and a negative value represents an arbitrarily worse model.

The R^2 value obtained for the 16 selected faults (shown in Table 6.1) is 0.879, which is a quite high value for this kind of scenario. The 16 fault type configuration corresponds to 2,244 generated faulty versions, which is nearly 20% of the 11,309 total faulty versions. Although 16 types of faults represents about 29% of the total number of types of faults, this number allows us to execute the test suite in about 19% of the time required to execute the whole set of faults, i.e., around 46 days in our experimental environment.

Using 16 types of faults may also still result in a high cost, depending on the tester's priorities and on the available resources. Thus, we tried to understand the cost of further reducing the selected fault types. Note that this has the immediate consequence of lowering the corresponding test execution time but potentially decreases the chances of detecting problems in the test cases. This further reduction of the selected types of faults supports testers in deciding how many fault types should be considered when performing this kind of fault injection campaigns.

Figure 6.1 presents the results, where we can see the distribution of R^2 and the corresponding cost of execution in days for a certain number of fault types (x axis).

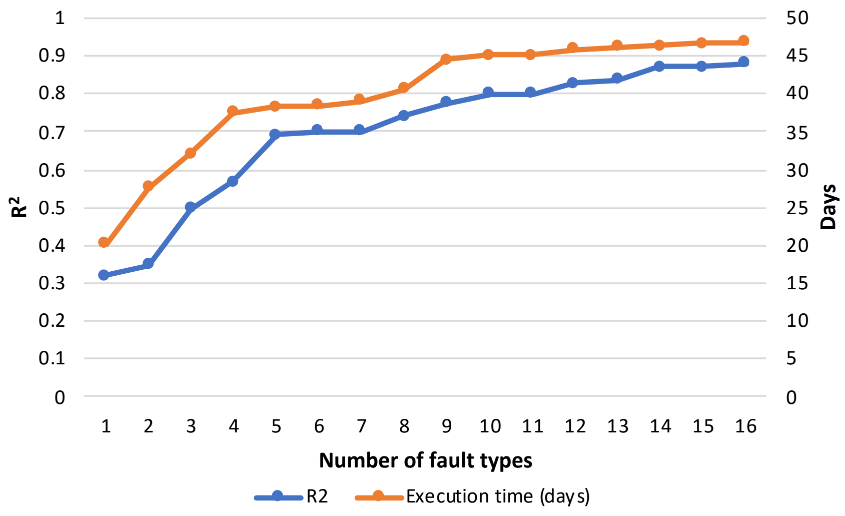

Figure 6.1: Distribution of R^2 by number of fault types

Figure 6.1 shows a progressive increase in cost (time of execution) with the increasing number of fault types, but it can provide useful information for testers. For instance, it shows that the selection of 4 fault types results in a good R^2 value associated with a relatively small execution cost (i.e., when compared with smaller values for the fault types).

Chapter 7

Main Findings

This chapter highlights the main findings of our experimental evaluation. We go through key and high-level aspects of the test suite effectiveness and present general recommendations for the developers of the OpenStack, based on the results of our experiments. It is worthwhile mentioning that we also shared our findings with OpenStack developers, to allow any possible improvements in the test suite.

- The distribution of Orthogonal Defect Classification (ODC) defects found in OpenStack Nova is unique (i.e., when compared with other works using ODC [Agnelo et al., 2020]), which reflects the specificity of the system and the expected need for performing the software faults classification, when the goal is to define a fault model.
- The analysis of the OpenStack Nova software defects lead to the definition of an extension of the fault model proposed in [Durães and Madeira, 2006]. In the model proposed in this thesis, about one fourth (14) of the 54 faults that compose the model are new and account for the specificity of the Python programming language, in the context of the OpenStack platform.
- The majority of defects detected in Nova's source code are of type Algorithm and Interface/O-O Messages, which reflects the component's complex integration in the OpenStack system.
- During our experiments, the whole OpenStack test suite (i.e., unit, functional, and integration tests) showed to be able to detect about 97.4% (i.e., 11,019 out of 11,309 faults) of the total amount of injected faults.
- The unit tests are the most effective in detecting the injected faults, being able, on their own, to detect 96.3% (i.e., 10,895 out of 11,309 faults) of the faults.
- The functional tests, on their own, are able to detect 69.6% of the whole set of injected faults (i.e., 7,876 out of 11,309 faults).
- The integration tests, on their own, are able to detect 2.9% of all injected faults (i.e., 325 out of 11,309 faults).
- The functional tests are able to detect nearly one third of faults that escape unit tests detection. More precisely, they detect 29.2% (i.e., 121 out of 414 injected faults) that are not detected by unit tests, being slightly helpful in improving the overall tests effectiveness, which raises from 96.3% (i.e., the outcome of unit testing) to 97.4%. However, there are still 293 faults undetected by the joint set of unit and functional tests.

- Of the 293 faults not detected by the set of unit and functional tests, the integration tests were able to detect 1% (i.e., 3 faults), just marginally increasing the overall effectiveness from 97.40% to 97.44%.
- From the defect type perspective, Timing/Serialization faults showed to be the most difficult to detect (regardless of testing level), which confirms the common view of the difficulty in detecting this kind of faults. On the other hand, the Algorithm/Method and Function/Class/Object faults did not contribute to the undetected cases.
- The unit tests were able to detect all of the Algorithm/Method and Function/Class/Object injected faults and showed also to be quite effective with Interface/O-O Messages and Relationship faults. Functional tests show a relatively balanced performance across all defect types (with exception of Timing/Serialization faults). Still, it is worth noting that the best results at the functional testing level are obtained with Assignment/Initialization defects, which show the worst values at the unit test level (i.e., excluding T/S faults). This highlights the complementarity of the different test levels in this test suite. Integration tests showed to be generally ineffective in detecting the various types of faults.
- Faults qualified as Missing are the largest contributor to the undetected cases, while Extraneous are fully detected by the test suite. Extraneous faults are actually all eagerly detected by the test suite, at the unit testing level.
- Three of the 14 new fault types that compose the fault model defined in this work are actually part of the *sufficient fault types* set. These are MCEFL, MSFC, and MSC, and account for 14.48% (42) of the 290 faulty versions that passed undetected through all tests.
- Overall, we found it would be *trivial* to correct or extend OpenStack tests in about half of the cases (i.e., 47%) of undetected faults. Corrections of *moderate* difficulty apply to more than one third (i.e., 38%) of the undetected faults. Only 15% of the faults would require *complex* corrections to be caught at the unit testing level. Also, our analysis did not reveal particular cases, where a correction at the functional or integration testing level was required (i.e., due to the complexity of the fix at the unit testing level).
- Our analysis to the code shows that the most common problems resulting from the injected faults are related with incorrect exceptions being raised, incorrect exception messages or incorrect log messages. Wrong values or objects with wrong field values being returned, or passed to function calls are also among the common problems found that can break the expected functionalities.
- In general, developers writing tests that target functions with multiple conditional control instructions, should consider placing more effort on implementing multiple conditions coverage tests, to allow covering as many combinations of sub-conditions, as possible.
- Tests should validate exceptions being raised, at least against predefined sets of exceptions. The same should happen with exception messages, which should at least be validated against expected message patterns, to minimize the generation of incorrect information that impairs maintenance activities.
- If a certain function under testing produces logs, then the tests should also validate if the logs generated are the expected, at least by matching the log messages against expected sets of patterns, this will prevent incorrect logs that could harm maintenance.

- Function mocks (i.e., functions used in the tests that replace the actual implementation of functions in other parts of the program being tested) should validate the parameters received against expected values, as using these mocks can hide potential problems upon that function call.

- If a function returns an object, unit tests should validate, to the extent that is possible, if all the expected fields are present, as unexpected null values are known to be common sources of failures in software [Laranjeiro et al., 2021]. Also, unit tests should verify if the individual values present in a returning object are the expected. This is certainly a complex task, and even not feasible in certain cases, due to the complexity or specificity of the software being tested (e.g., consider non-deterministic operations). Anyway, this type of guideline should be followed, to the extent allowed by the system and involving context.

- If the function under test is expected to call some other function under specific conditions, the unit test should assert that the function was called, or not called, according to the provided input (e.g., assert that the cache reset function was not called).

This page is intentionally left blank.

Chapter 8

Threats to Validity

In this chapter, we briefly discuss the main threats to the validity of this work and present mitigation strategies. We begin by mentioning that *we analyzed 2,048 bug reports, which is a subset of all reported bugs for OpenStack* (the OpenStack Nova project counts 10,152 reported bugs, by July 2021), mostly due to the amount of effort required to analyse the bug reports. Also, *the reported defects are associated with the Nova project*, which is just a part of OpenStack. Thus, the set of defects used may not be fully representative of OpenStack. We tried to mitigate this issue primarily by making sure that the size of the main set of defects would be kept above reference works, such as [Durães and Madeira, 2006]. The set of bug reports is actually larger than the vast majority of sets used in defect classification research [Agnelo et al., 2020]. It is also worthwhile mentioning that there may be *defects that are not reported* in the platform used by the project, which could contribute to the definition of the fault model. In this work, we use the information that is publicly available and has been produced under the general bug reporting practice in open source systems (at least in those of some dimension), which is the use of a bug tracking system for reporting software defects. We must also mention that despite referring to a part of OpenStack, the reported defects actually refer to the core component of OpenStack (i.e., Nova Compute), which, due to its importance in the system, should gather more attention from developers in what concerns Verification and Validation tasks.

The software defect analysis performed in this work is carried out through the analysis of bug reports referring to a particular component of a Python application, OpenStack Nova. Results characterize the defects present in this particular system and, in this sense, *results cannot be generalized to other Python applications or even to other OpenStack components.* Despite this, the study lays the foundation to allow future research to confirm that other components of OpenStack, or projects of similar nature, share the same distribution of faults (or otherwise diverge in such distribution).

The *software defect classification was carried out by one researcher.* This may have resulted in the introduction of some errors in the final labeled dataset, due to the large amount of human effort involved, due to the quality of the bug reports (e.g., lack of or erroneous information present in reports), or due to the technical expertise of the researcher applying the ODC method. To mitigate this issue, the Early Stage Researcher, responsible for performing the classification, was selected also based on his 5-year experience as a software engineer in the industry. Moreover, we involved an additional researcher and performed an independent classification of one fourth of the software defects, reaching substantial or almost perfect agreement, assuring the quality of the dataset. Due to the amount of effort involved, it is not viable to verify the complete dataset, still the verified portion provides excellent indications of its overall quality.

The *extended ODC classification was performed using one fourth of the whole dataset* (i.e., 512 bugs), which may not be representative of the whole set of software defects (e.g., a larger set could allow identifying further types of faults). This was due to the huge amount of effort involved, however we tried to reach a size similar to those used in reference research, e.g., [Durães and Madeira, 2006] to mitigate this issue. *The fault injection tool may hold residual software faults* that may impair its functionality (e.g., incorrectly injecting a software fault). To mitigate this issue, we manually verified the injection of all 54 fault types, in diverse contexts. While this does not ensure the tool is free of bugs, it provides higher confidence in its correct operation.

Chapter 9

Conclusion and Future Work

In this thesis, we presented FIT4Python, a tool for performing fault injection in Python applications and used it to analyse the effectiveness of OpenStack's battery of tests, which is composed by unit, functional, and integration tests. We first analysed 2,048 bug reports from OpenStack using ODC and performed an extended classification of 512 of those bugs to define a fault model tailored for Python, in the context of OpenStack. We implemented our fault injection tool, based on the created fault model, and ran it to create faulty versions of the Nova Compute API. We ran OpenStack developer's unit, functional, and integration tests against the faulty versions resulting in more than 245 million executed tests (detailed results, along with the fault injection tool, are available at [Marques et al., 2021]).

In addition to the capability of injecting different types of faults, we show the coverage and highlight limitations of the OpenStack battery of tests, with several cases of faults passing silently undetected through the tests. Moreover, we show that it would be trivial to correct or extend OpenStack tests to detect many of the injected problems, which we have shared with OpenStack developers.

As future work, we intend to make the tool simpler to use by reducing the setup effort and configurations necessary to make the application run, as well as, make the application more generic so that the fault injection technique can be applied to different python applications.

References

J. Agnelo, N. Laranjeiro, and J. Bernardino. Using orthogonal defect classification to characterize nosql database defects. *Journal of Systems and Software*, 159:110451, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110451. URL `https://www.sciencedirect.com/science/article/pii/S0164121219302250`.

J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, 2003. doi: 10.1109/TC.2003.1228509.

S. N. AS. Cosmic Ray, July 2021. URL `https://pypi.org/project/cosmic-ray`. original-date: 2015-04-18T07:44:21Z.

J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, 1990. doi: 10.1109/12.54853.

J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, 1990. doi: 10.1109/12.54853.

P. Broadwell, N. Sastry, and J. Traupman. Fig: A prototype tool for online verification of recovery mechanisms. *Workshop on Self-Healing, Adaptive and self-MANaged Systems.*, 2002.

C. Byoungju and A. P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135 – 152, 1993. ISSN 0164-1212. doi: https://doi.org/10.1016/0164-1212(93)90005-I. URL `http://www.sciencedirect.com/science/article/pii/016412129390005I`.

R. Chillarege. Orthogonal defect classification. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 9, pages 359–399. IEEE CS Press, 1996.

J. Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, Apr. 1960. ISSN 0013-1644. doi: 10.1177/001316446002000104. URL `https://doi.org/10.1177/001316446002000104`.

D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 200–211, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338916. URL `http://doi.acm.org/10.1145/3338906.3338916`. event-place: Tallinn, Estonia.

D. Cotroneo, L. D. Simone, P. Liguori, and R. Natella. ProFIPy: Programmable Software Fault Injection as-a-Service. In *2020 50th Annual IEEE/IFIP International Conference* *on Dependable Systems and Networks (DSN)*, pages 364–372, June 2020. doi: 10.1109/DSN48063.2020.00052. ISSN: 1530-0889.

P. Delgado-Pérez, S. Segura, and I. Medina-Bulo. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability*, 27(4-5):e1630, 2017. ISSN 1099-1689. doi: https://doi.org/10.1002/stvr.1630. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1630`.

A. Denisov and S. Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, Apr. 2018. doi: 10.1109/ICSTW.2018.00024.

A. Derezińska. A Quality Estimation of Mutation Clustering in C# Programs. In W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, editors, *New Results in Dependability and Computer Systems*, Advances in Intelligent Systems and Computing, pages 119–129, Heidelberg, 2013. Springer International Publishing. ISBN 978-3-319-00945-2. doi: 10.1007/978-3-319-00945-2_11.

J. A. Durães and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11), 2006.

B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Ann. Statist.*, 32(2):407–499, 04 2004. doi: 10.1214/009053604000000067. URL `https://doi.org/10.1214/009053604000000067`.

J. Fonseca, M. Vieira, and H. Madeira. Evaluation of web security mechanisms using vulnerability attack injection. *IEEE Transactions on Dependable and Secure Computing*, 11(5):440–453, Sep. 2014. ISSN 2160-9209. doi: 10.1109/TDSC.2013.45.

G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, June 2015. ISSN 1573-7616. doi: 10.1007/s10664-013-9299-z. URL `https://doi.org/10.1007/s10664-013-9299-z`.

R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 978-0-13-720384-0.

A. Hajdu, N. Ivaki, I. Kocsis, A. Klenik, L. Gönczy, N. Laranjeiro, H. Madeira, and A. Pataricza. Using fault injection to assess blockchain systems in presence of faulty smart contracts. *IEEE Access*, 8:190760–190783, 2020. doi: 10.1109/ACCESS.2020.3032239.

S. Han, K. Shin, and H. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, 1995. doi: 10.1109/IPDS.1995.395831.

K. Hałas. MutPy: Mutation testing tool for Python 3.x, 2021. URL `https://pypi.org/project/mutpy`.

A. Hovmöller. Mutmut: mutation testing for Python 3, 2021. URL `https://pypi.org/project/mutmut`.

M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, 1997. URL `http://portal.acm.org/citation.cfm?id=619017.620685&coll=Portal&dl=GUIDE&CFID=74933873&CFTOKEN=73521499`.

J. Hudak, B.-H. Suh, D. Siewiorek, and Z. Segall. Evaluation and comparison of fault-tolerant software techniques. *IEEE Transactions on Reliability*, 42(2):190–204, June 1993. ISSN 1558-1721. doi: 10.1109/24.229487.

IBM. Orthogonal Defect Classification v 5.2 for Software Design and Code, Sept. 2013. URL `https://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf`.

IEEE. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, Jan. 2010. ISSN null. doi: 10.1109/IEEESTD.2010.5399061.

R. Just and F. Schweiggert. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, 25(5-7):490–507, 2015. ISSN 1099-1689. doi: https://doi.org/10.1002/stvr.1561. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1561`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1561.

K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008. ISBN 047023055X.

W. . Kao, R. K. Iyer, and D. Tang. Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, Nov 1993. ISSN 2326-3881. doi: 10.1109/32.256857.

E. Kepner. Mutatest, June 2021. URL `https://pypi.org/project/mutatest/`. original-date: 2018-12-22T15:04:53Z.

M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering*, 44(04):308–333, Apr. 2018. ISSN 0098-5589. doi: 10.1109/TSE.2017.2684805. URL `https://www.computer.org/csdl/journal/ts/2018/04/07882714/13rRUxBa5ty`. Publisher: IEEE Computer Society.

P. Koopman and J. DeVale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000. doi: 10.1109/32.877845.

N. P. Kropp, P. J. Koopman, and D. 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 '98, pages 230–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 978-0-8186-8470-8. URL `http://dl.acm.org/citation.cfm?id=795671.796919`.

B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 571–582, New York, NY, USA, Nov. 2016. Association for Computing Machinery. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950322. URL `https://doi.org/10.1145/2950290.2950322`.

J. R. Landis and G. G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977. ISSN 0006-341X. doi: 10.2307/2529310. URL `https://www.jstor.org/stable/2529310`.

N. Laranjeiro, J. Agnelo, and J. Bernardino. A Systematic Review on Software Robustness Assessment. *ACM Computing Surveys*, 54(4):89:1–89:65, May 2021. ISSN 0360-0300. doi: 10.1145/3448977. URL https://doi.org/10.1145/3448977.

N. Li, M. West, A. Escalona, and V. H. S. Durelli. Mutation testing in practice using Ruby. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–6. IEEE Computer Society, Apr. 2015. ISBN 978-1-4799-1885-0. doi: 10.1109/ICSTW.2015.7107453. URL https://www.computer.org/csdl/proceedings-article/icstw/2015/07107453/12OmNvUaNme.

J. A. P. Lima and S. R. Vergilio. Search-Based Higher Order Mutation Testing: A Mapping Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, SAST '18, pages 87–96, New York, NY, USA, Sept. 2018. Association for Computing Machinery. ISBN 978-1-4503-6555-0. doi: 10.1145/3266003.3266013. URL https://doi.org/10.1145/3266003.3266013.

A. Mahmood, D. M. Andrews, and E. J. McCluskey. Executable assertions and flight software. In *Digital Avionics Systems Conference*, pages 346–351. American Institute of Aeronautics and Astronautics, 1984. doi: 10.2514/6.1984-2726. URL https://arc.aiaa.org/doi/abs/10.2514/6.1984-2726.

H. Marques, N. Laranjeiro, , and J. Bernardino. Injecting software faults in python applications: The openstack case study - supplemental material, Mar. 2021. URL https://doi.org/10.5281/zenodo.4581121.

E. Martins, C. Rubira, and N. Leme. Jaca: a reflective fault injection tool based on patterns. In *Proceedings International Conference on Dependable Systems and Networks*, pages 483–487, June 2002. doi: 10.1109/DSN.2002.1028934.

T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2 (4):308–320, Dec. 1976. ISSN 2326-3881. doi: 10.1109/TSE.1976.233837.

B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://doi.org/10.1145/96267.96279.

B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. 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.

A. S. Namin, J. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 351–360, 2008. doi: 10.1145/1368088.1368136.

R. Natella. *Achieving Representative Faultloads in Software Fault Injection.* PhD Thesis, Università degli Studi di Napoli Federico II, Napoli, Italy, Nov. 2011. URL http://www.fedoa.unina.it/8833/.

R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, Feb. 2016a. ISSN 0360-0300. doi: 10.1145/2841425. URL http://doi.acm.org/10.1145/2841425.

R. Natella, D. Cotroneo, and H. S. Madeira. Assessing Dependability with Software Fault Injection: A Survey. *ACM Computing Surveys*, 48(3):1–55, Feb. 2016b. ISSN 0360-0300, 1557-7341. doi: 10.1145/2841425. URL https://dl.acm.org/doi/10.1145/2841425.

W. Ng and P. Chen. The design and verification of the Rio file cache. *IEEE Transactions on Computers*, 50(4):322–337, Apr. 2001. ISSN 2326-3814. doi: 10.1109/12.919278.

OpenStack. Conceptual Architecture, 2021. URL `https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html`.

G. Petrović and M. Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 163–171, New York, NY, USA, May 2018. Association for Computing Machinery. ISBN 978-1-4503-5659-6. doi: 10.1145/3183519.3183521. URL `https://doi.org/10.1145/3183519.3183521`.

A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.07.100. URL `http://www.sciencedirect.com/science/article/pii/S0164121219301554`.

U. Praphamontripong and J. Offutt. Finding redundancy in web mutation operators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 134–142, 2017. doi: 10.1109/ICSTW.2017.30.

U. Praphamontripong and J. Offutt. Finding Redundancy in Web Mutation Operators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 134–142, Mar. 2017. doi: 10.1109/ICSTW.2017.30.

T. Rosado and J. Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering and Applications Symposium*, IDEAS '14, page 366–367, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326278. doi: 10.1145/2628194.2628195. URL `https://doi.org/10.1145/2628194.2628195`.

B. P. Sanches, T. Basso, and R. Moraes. J-SWFIT: A Java Software Fault Injection Tool. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 106–115, Apr. 2011. doi: 10.1109/LADC.2011.20.

Tiobe. TIOBE Index, Dec. 2019. URL `https://www.tiobe.com/tiobe-index/`.

B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 295–306, New York, NY, USA, July 2017. Association for Computing Machinery. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092714. URL `https://doi.org/10.1145/3092703.3092714`.

Wei-Lun Kao and R. K. Iyer. Define: a distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, June 1994. doi: 10.1109/FTPDS.1994.494497.

P. F. Wilson, L. Dell, and G. Anderson. Root cause analysis: A tool for total quality management. *ASQ Quality Press*, 1993.

A. Wood. Software Reliability Growth Models. Technical Report 96.1, Tandem Computers, 1996.

M. Woodward. Mutation testing—its origin and evolution. *Information and Software Technology*, 35(3):163 – 169, 1993. ISSN 0950-5849. doi: https://doi.org/10.1016/

0950-5849(93)90053-6. URL `http://www.sciencedirect.com/science/article/pii/0950584993900536`.

L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 92–102, 2013. doi: 10.1109/ASE.2013.6693070.

L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 235–245, New York, NY, USA, July 2013. Association for Computing Machinery. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483782.