USI

Test Case Generation and Fault Localization for Data Science Programs

Mohammad Rezaalipour

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Supervised by

Prof. Carlo A. Furia

Università della Svizzera italiana

Facoltà di scienze informatiche

Dissertation Committee

Prof. Domenico Bianculli	University of Luxembourg, Luxembourg
Prof. Gordon Fraser	University of Passau, Germany
Prof. Michele Lanza	Università della Svizzera italiana, Switzerland
Prof. Paolo Tonella	Università della Svizzera italiana, Switzerland

Dissertation accepted on 13 June 2024

Research Advisor
Prof. Carlo A. Furia

Ph.D. Program Co-Director
Prof. Walter Binder

Ph.D. Program Co-Director
Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Mohammad Rezaalipour
Lugano, 13 June 2024

“Fear is the main source of superstition, and one of the main sources of cruelty. To conquer fear is the beginning of wisdom."

— Bertrand Russell

Abstract

Data science refers to inter-disciplinary approaches designed to extract knowledge from vast amounts of data. It combines techniques from fields such as statistics and machine learning to develop novel applications for different science and engineering domains. Data science approaches are implemented as programs usually written in languages such as R or Python, collectively referred to as data science programs. Due to their inter-disciplinary usages, these programs are often written by domain experts possibly unfamiliar with the best practices of software development, and thus, they may exhibit low quality. In fact, there is evidence that these programs contain several bugs, often different in nature compared to those found in traditional programs. As a result, data science programs challenge conventional debugging techniques such as those from test generation and fault localization activities, due to the unique nature of bugs found in them. Additionally, being written in dynamically typed languages such as Python adds to the difficulties of testing and analyzing them.

These challenges call for research into new debugging techniques tailored specifically for these programs, which is the focus of the current dissertation. Precisely, this thesis aims to understand the capabilities and limitations of standard test generation and fault localization techniques on data science programs implemented in dynamic languages such as Python. To achieve this goal, the dissertation presents contributions in three areas: *i*) a test generation technique for neural network (NN) programs, a wide spread class of data science programs; *ii*) an empirical study of fault localization in Python programs; and *iii*) two debugging tools and a curated dataset of NN bugs.

In the first area, we investigated and identified the limitations of general-purpose test generation techniques on NN programs, which led to the development of aNNoTest, a novel test generation technique tailored for NN programs. We evaluated aNNoTest on 19 open-source programs, demonstrating its effectiveness at finding bugs in real-world NN programs. In the second area, we conducted the first large-scale multi-family empirical study of fault localization in Python programs. Targeting 135 bugs from 13 projects, we studied seven fault localization techniques from four families along with combinations of them. We considered different fault localization granularity levels and measured both effectiveness and efficiency in our analyses. In the third area, we developed: *i*) the aNNoTest tool, an implementation of the aNNoTest approach mentioned above; *ii*) FauxPy, to our knowledge, the first open-source multi-family fault localization tool for Python; and *iii*) a curated dataset of NN bugs, for which aNNoTest was used to generate tests.

Along with supporting the domain with the tools and techniques we developed, we hope our contributions will be beneficial to inform the development of more effective debugging techniques for Python data science programs.

Contents

Part I

Prologue

1

Introduction

"Data science" is an umbrella term that denotes new inter-disciplinary approaches to analyze vast amounts of data to extract knowledge from it. First envisioned by database pioneer Jim Gray [16, 49], data science combines techniques from statistics, machine learning, and information science in a way that helps science and engineering domains—such as economics, finance, biology, medicine, and computer science—to develop novel applications with better performance—such as designing autonomous vehicles, processing natural language, and doing image recognition. Data science techniques are typically implemented in the form of programs: a neural network that recognizes faces, a complex regressive model that is fitted on epidemiological data, and so on. This dissertation will target such data science programs with the overall goal of analyzing, improving, and helping develop them.

Since data science techniques are used to support other scientific disciplines, data science programs are often written by domain experts who may *not* be professionally trained programmers familiar with the best practices of large-scale software development. Therefore, data science programs are more likely to be poorly designed, insufficiently tested, and hence they tend to contain numerous bugs.[1] For instance, a data science program developed for simulating the spread of the Coronavirus pandemic [38] has been criticized as being unreliable and buggy. According to some critics, it was not possible to reproduce the same results using the same code and data [2]. Another study [139] reports that a large percentage of deep learning (DL) program jobs submitted to *Philly*—Microsoft's deep learning platform—by researchers and developers at Microsoft fail to finish due to code defects, which leads to the waste of expensive resources such as GPU. The increasing use of data science programs in safety-critical domains [21, 75] indicates that techniques to help developers improve the quality of these programs are strongly needed as undetected bugs within such software may lead to disasters [3].

In traditional software engineering research, there are lots of research results and techniques targeting the automation of different debugging activities, including detecting [20], locating [131], and removing [44] bugs. However, data science programs often are of a different nature compared to traditional programs, which challenges the effectiveness of the existing techniques. Data science programs are mostly unconventional programs (e.g., neural networks) and often have script-like nature (e.g., they have poor modularization). According to recent studies [57, 142], the bugs occurring in the source code of deep learning programs—a broad and important class of data science programs—are often different from bugs usually found in traditional software. Studies [58, 142] found that locating these bugs is challenging too, for instance, because the bugs often result in crashes with little information about their root cause in the program. In addition, study [58] found that the bug fix patterns required by deep learning programs are different from those of traditional software,

[1] In this dissertation, we use the terms "fault" and "bug" as synonyms.

and fixing them requires information about the architecture of the models employed within these programs (something that may not be easily accessible).

Several of these challenges follow from the fact that data science programs are usually written in dynamic programming languages such as Python and R. Automatic test generation for dynamically typed programming languages is a largely open challenge [78] as most existing techniques for statically typed languages extensively rely on the typed signatures of tested functions, which is not usually available in Python or R. All of these challenges indicate that data science programs require new analysis and debugging techniques specific to them.

1.1 Thesis Statement

Our thesis statement is formulated as follows:

> *Understanding the capabilities and limitations of standard test generation and fault localization techniques on data science programs implemented in dynamic languages such as Python informs the development of new techniques that can be more effective.*

As indicated in the thesis statement, we focus on two activities: *i*) test generation [11], and *ii*) fault localization [109, 131], which are covered in the first and second parts of this dissertation, respectively.

Neural network (NN) programs are arguably becoming a widespread class of data science programs, as, with the help of popular NN development libraries and frameworks such as Keras and Tensorflow, many engineers and researchers have been using NN in their software products. Therefore, NN programs exemplify some of the key challenges of testing and debugging such programs. Generating tests for NN programs is quite challenging given their peculiar characteristics. On the other hand, the dynamic nature of Python makes this debugging task even more challenging. In the **first part** of this dissertation, we focus on analysis of NN programs written in Python, and introduce a novel test generation technique tailored specifically for Python-based NN programs.

Several of the challenges posed by analyzing NN programs derive from a combination of the programs' domain with characteristics of the Python programming language. It is well-known that Python is one of the most widely used languages to date [5]; its vast ecosystem of heterogeneous libraries and programs, its fast evolution and adoption, and its interpreted and dynamic nature contribute to several of the challenges that we tackled in this dissertation's work. Therefore, the **second part** of this dissertation broadens the focus from Python NN programs to Python open-source programs (including data science and NN libraries, but also including more conventional kinds of applications and libraries). In this second part, we present an extensive empirical study that investigates fault localization in Python programs, laying the foundation for development of new and more effective techniques in this domain.

1.2 Contributions

The contributions of this dissertation can be grouped into the three following categories: *i*) a test generation technique for NN programs [99]; *ii*) an empirical study of fault localization in Python programs [101]; *iii*) and the supporting tools and dataset developed during this PhD [100, 102].

1.2.1 Test Generation Approach for NN Programs

On the theoretical side, this dissertation proposes an automated test generation technique tailored for NN programs.

NN programs usually work with complicated data types while typing information is not available in dynamic programming languages. As a result, existing test generation techniques are not effective on NN programs.

To address this gap, we designed ANNOTEST, an automated technique that generates test inputs for NN programs written in Python. This technique provides a simple annotation language named AN to equip a program with information about its valid inputs, required by the test generation process. Using the information provided by AN, ANNOTEST automatically generates tests for the given program. We evaluated ANNOTEST's bug-finding capabilities on 19 open-source NN projects surveyed by Islam et al. [57], indicating ANNOTEST's capability at finding widespread bugs in real-world NN programs.[2]

1.2.2 Empirical Study of Fault Localization in Python Programs

In order to better understand the capabilities of existing fault localization techniques on bugs of programs written in Python, including both traditional applications and data science frameworks, this dissertation presents the results of a large-scale empirical study.

Despite the massive amount of work on fault localization and the popularity of the Python programming language[3][4], most empirical studies of fault localization target languages like Java or C. This leaves open the question of whether Python's characteristics—such as the fact that it is dynamically typed, or that it is dominant in certain application domains such as data science—affect the capabilities of classic fault localization techniques—developed and tested primarily on different kinds of languages and programs.

This empirical study fills this knowledge gap: to our knowledge, it is the first multi-family large-scale empirical study of fault localization in real-world Python programs including both data science and traditional ones. This study investigates the effectiveness (i.e., localization accuracy), efficiency (i.e., runtime performance), and other features (e.g., different entity granularities) of seven well-known fault-localization techniques in four families (spectrum-based, mutation-based, predicate switching, and stack-trace based) on 135 faults from 13 open-source Python projects from the BUGSINPY curated collection [128].

The results of this study help inform future research on the capabilities and limitation of current fault localization techniques on data science and other programs written in a dynamic programming language like Python.

1.2.3 Supporting Tools and Dataset

On the practical side and based on the first two contributions listed above (i.e., Section 1.2.1 and Section 1.2.2), this dissertation presents two tools, ANNOTEST and FAUXPY, developed to support data science program developers with automated test generation and fault localization:

- The **ANNOTEST** tool is the implementation of the ANNOTEST technique introduced in Section 1.2.1. It is an automated unit-test generation tool for NN programs written in Python. The ANNOTEST tool is publicly available,[5] and its main repository[6] includes the tool's source code and instructions to use it.

[2] ANNOTEST also found a previously unknown bug{1} and a documentation inconsistency{2} within PyTorch Vision. Upon reporting these issues, they were promptly acknowledged and accepted by Vision's maintainers.

[3] TIOBE language popularity index: `https://www.tiobe.com/tiobe-index/`

[4] Popularity of Programming Language Index: `https://pypl.github.io/PYPL.html`

[5] `https://pypi.org/project/annotest`

[6] `https://github.com/atom-sw/annotest`

- FAUXPY is a multi-family fault localization tool for Python programs. We developed FAUXPY to conduct the study introduced in Section 1.2.2. FAUXPY supports seven well-known fault localization techniques in four families: spectrum-based, mutation-based, predicate switching, and stack trace fault localization. To the best of our knowledge, at the time of writing, FAUXPY is the only available tool for Python that offers fault localization techniques beyond spectrum-based ones. FAUXPY is publicly available,[7] and its repository[8] includes its source code and instructions to use it.

Furthermore, this dissertation presents a curated collection of real-world reproducible NN bugs,[9] which can support further work in this domain. We developed this collection while conducting the study introduced in Section 1.2.1. This collection includes the source code of 62 bugs in 19 open-source NN projects surveyed by Islam et al. [57]—which we reproduced using ANNOTEST. We curated this repository to ensure that each bug is easily reproducible using ANNOTEST—or with any other Python source-code tool.

1.3 Outline

The rest of this dissertation is structured as follows:

Chapter 2 reviewers the state-of-the-art, specifically, studies regarding bug characteristics, test-case generation, and fault localization that are relevant to this dissertation.

Chapter 3 presents ANNOTEST, our test generation technique tailored for NN programs, introduced in Section 1.2.1. This chapter is based on the following publication [99]:

- M. Rezaalipour and C. A. Furia. **An annotation-based approach for finding bugs in neural network programs.** Journal of Systems and Software, 201:111669, 2023.

Chapter 4 introduces our automated unit-test generation tool ANNOTEST[10] along with the dataset outlined in Section 1.2.3. This chapter is based on the following publication [100]:

- M. Rezaalipour and C. A. Furia. **aNNoTest: An annotation-based test generation tool for neural network programs.** In IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 574–579, 2023.

Chapter 5 presents our extensive multi-family empirical study of fault localization in Python programs, introduced in Section 1.2.2. This chapter is based on the following publication [101]:

- M. Rezaalipour and C. A. Furia. **An empirical study of fault localization in Python programs.** Empirical Software Engineering, 2024. Accepted in March 2024.

Chapter 6 presents FAUXPY, our automated, multi-family fault localization tool for Python programs.

Chapter 7 concludes the current dissertation and outlines potential avenues for future research.

[7] `https://pypi.org/project/fauxpy`
[8] `https://github.com/atom-sw/fauxpy`
[9] `https://github.com/atom-sw/annotest-subjects`
[10] Both the tool and technique are called ANNOTEST.

2

State of the Art

As highlighted by our thesis statement (Section 1.1), this dissertation aims to better understand the capabilities and limitations of *test generation* and *fault localization* debugging techniques on *data science programs*. Consequently, this chapter explores the state-of-the-art in three areas. Section 2.1 explores the nature of data science programs, focusing on the type of bugs they exhibit. Section 2.2 focuses on test generation studies. Section 2.3 highlights fault localization studies relevant to this dissertation. Finally, Section 2.4 concludes this chapter with a list of main open gaps in these three areas.

2.1 Data Science Programs

Data science programs—e.g., general machine learning and neural network programs—are often implemented in Python or R, two dynamically typed programming languages [46]. These languages provide well-maintained libraries and frameworks for various widely-used data science activities such as manipulating data and visualizing them. For instance, Python's extensive ecosystem provides the TensorFlow framework along with libraries such as NumPy and Pandas, all of which are useful for data science program development. Python also has a smooth learning curve due to its intuitive syntax. These characteristics make these languages, particularly Python, an appealing choice for data science program development as many of the practitioners in this area are experts within domains other than computer science and may lack traditional programming skills and background.

Like any other programs, data science programs have bugs too. Following the increasing in popularity of neural network and other forms of machine learning, some recent research has looked into the nature of bugs that occur in these programs to understand how they differ compared to "traditional" software. Studying these bugs can provide better understanding of the possible limitations of different debugging activities applied to data science programs.

2.1.1 Bugs in Data Science Programs

Thung et al. [122] conducted an empirical study on 500 bugs and their corresponding human-written patches found in Apache Mahout, Lucene, and OpenNLP, which are a data mining library, an information retrieval library, and a natural language processing tool, respectively. They manually categorized these bugs based on their types and investigated the relationship between these bug types and their attributes such as their severity, bug-fixing duration and bug-fixing effort. According to their results, most severe bugs in these systems are caused by incorrect implementation of certain algorithms or functions (e.g., fuzzy search). Sun et al. [119] conducted a similar study on 329 bugs found in the data analysis library Scikit-learn and two deep learning frameworks, Paddle and Caffe. They report

that unlike traditional software, compatibility bugs (e.g., conflicts with hardware, operating systems, and version of dependencies) and variable bugs (e.g., incorrect variable assignments and wrong data formats) are the most prevalent bug types in these programs.

Zhang et al. [142] studied the source code of several TensorFlow-based programs form Stack Overflow and GitHub to discover the new debugging challenges specific to deep learning programs. They report information such as the symptoms and root causes of these bugs and the challenges to detect and locate them. The root causes reported in this paper (e.g., modeling and parameter mistakes, incorrect tensor shapes, and unfamiliarity of users with the underlying computational model of TensorFlow) are all specific to machine learning programs. This observation indicates that bugs in such programs are of a peculiar nature. Similar to Zhang et al. [142], Islam et al. [57] conducted an empirical study on deep learning programs considering more deep learning frameworks and libraries, namely Caffe, Keras, Tensorflow, Theano, and Torch. They provide a classification of bug types occurring in programs using the mentioned frameworks, their root causes, and their symptoms.

Islam et al. conducted a follow-up study [58] later on the same dataset to discover the repair challenges of deep learning programs. Both of Islam et al.'s studies [57, 58] confirm some of the results indicated by Zhang et al. [142], and also report that the bug-fix patterns of deep learning programs are pretty different in nature compared to those found in traditional programs.

Humbatova et al. [53] developed a taxonomy of bugs found in deep learning programs by both manually analysing an extensive set of GitHub and Stack Overflow software artifacts and interviewing several developers and researchers. This taxonomy contains various deep learning specific bug types such as incorrect/incomplete DL models, wrong shape and types of input data, and training process issues.

2.1.2 Bugs in Neural Network Models

A neural network (NN) program, such as those reviewed above [53, 57, 58, 142], implements in code a NN *model* that is trained on some *data*, both of which can also be plagued by mistakes. Hence, traditional software engineering approaches to test generation [120], mutation testing [51, 114], fault localization [37], and even automated program repair [116] have been applied to NN models and training data to assess and improve their quality, robustness, and correctness.

Under this paradigm, bugs are revealed by *adversarial examples*, e.g., two slightly different inputs that appear identical to the human eye but result in widely different classification by a trained model [120]. Adversarial examples correspond to failing tests; and fault localization and fixing correspond to finding [37] and changing [116] neuron weights in a model. This kind of research is complementary to this dissertation. However, the focus of this dissertation is on debugging the kind of bugs reviewed in Section 2.1.1, which are those found in the code implementations of NN programs.

Summary

The studies reviewed in Section 2.1.1 indicate that bugs found in data science programs are quite different in nature compared to those found in traditional software. These programs contain plenty of more traditional bugs too. However, the inputs and tests to trigger these bugs are of a different nature (Chapter 3). Traditional fault localization techniques may also behave differently on these programs due to both the unique nature of their bugs and the dynamic nature of the programming languages that implement them (Chapter 5). In addition, although most of these studies have nice replication packages, they took bugs from Stack Overflow and older versions of data science programs. Thus, they do not provide a fully reproducible set of bugs (such as including build files, dependencies, and

test suites). To enable controlled debugging studies in this area, curated collections of reproducible data science bugs similar to for instance Defects4J [63] are required (Chapter 4).

2.2 Automated Test Generation

Testing a program comprises three main steps [9]. First, selecting concrete *inputs* (arguments and pre-state) [11]; second, *executing* the program under test on those inputs; and third, checking whether the program behaved as expected while executing—in particular, whether its *output* (return values and post-state) is as expected.

This dissertation focuses on the first step, generating valid test inputs that satisfy the validity constraints (i.e., preconditions) of the program being tested (Chapter 3). The second step is already addressed by existing unit-testing frameworks such as Pytest. In contrast, addressing the third step requires an *oracle*: a mechanism to check the outcome of test execution; thus, the problem of designing such mechanisms is known as the oracle problem [14, 87, 95, 113], briefly reviewed in Section 2.2.2. In our study, we can leverage any existing oracles (Section 3.3.5).

2.2.1 Test-input Generation

Randoop [89] and EvoSuite [39] are two of the most popular automatic test generation tools for object-oriented programs written in Java. Randoop employs random testing to generated random sequences of method calls to produce tests. While having a simple implementation and being fairly effective at finding bugs, random testing is known as not to be quite effective at producing test suites when a certain coverage is needed [88]. EvoSuite attempts to achieve high program coverage by using a genetic algorithm, while keeping the test suite size as small as possible. EvoSuite is a particular form of Search-based Testing [8], which encodes the input generation problem as an optimization problem, and then uses a metaheuristic algorithm [19, 36] to solve it.

For test generation, Randoop and EvoSuite crucially rely on the typing information of methods provided by the statically typed languages such as Java. Typing information is not available in dynamically typed programming languages such as Python and R. Data science programs are mostly written in dynamically typed programming languages. These languages have fundamental issues with automatic test generation because of their dynamic nature [78].

In fact, despite Python's popularity [5], the first widely available tools for automated test-case generation in Python appeared only in recent years [29, 78, 80]. Pynguin [78] is based on genetic algorithms like EvoSuite, and leverages Python's type hints [4] for test generation. Hypothesis [80] implements property-based testing, which generates random inputs trying to satisfy some programmer-written properties. Deal [29] is a Python library for design by contract that provides decorations to express pre- and postconditions; based on them, Deal supports both static and dynamic (i.e., test-case generation) analysis.

All the three mentioned tools leverage additional annotations to generate tests for Python programs: Pynguin supports type hints (although it can also generate tests without type annotations), whereas Hypothesis relies on user-provided properties, and Deal exploits pre-conditions for test generation.

Type hints are a way of providing typing information in Python programs. However, data science programs work with complicated data types such as NumPy arrays, TensorFlow objects and Keras models that cannot be specified properly by type hints. For instance, the type of a Keras model object given to a function as input is simply specified as *Sequence* in type hints, which omits important information about the structure of the model, and thus, resulting in the generation of bad test suites—

invalid tests (Section 3.2). Hypothesis supports NumPy arrays. However, it does not support more complex data types such as TensorFlow objects.

Although directly using Pynguin, Hypothesis, or Deal to generate tests for data science programs might be possible in principle, it would involve plenty of additional manual work to express the necessary constraints indirectly through a combination of type hints (Pynguin) and testing strategies (Hypothesis), and to program test-case generation strategies that match them (Deal). In fact, to generate effective tests for data science programs, more expressive type annotations are required (Section 3.3.1), which is a clear research gap in the literature.

2.2.2 Oracles

Similarly as for test-input generation, a key research challenge is *automating* the generation of suitable oracles, so as to reduce the required developer effort. The simplest kind of oracles are *implicit oracles*, such as the crashing oracles. More expressive automated oracles may be derived from some kind of formal specification [7], such as assertions [26] and contracts [12], as well from informal or semi-formal documentation written in natural language [18, 112].

In absence of specifications, a practical option is building *regression oracles* [134], which check whether a new version of a program retains the same input/output behavior on the test inputs as a previous version [132]; test-input generators—like the aforementioned Randoop, EvoSuite, and Pynguin—are usually capable of building some kind of regression oracles automatically.

Summary

Data science programs are usually written in dynamically typed programming languages, which do not contain typing information required by currently existing test generation tools. Type hints can provide such information; but it is not expressive enough considering the complicated data types used in data science programs. Thus, even with the existence of type hints, current test generation techniques cannot perform effectively for these programs. As a result, a more expressive set of type annotations are needed to address this problem and fill the research gap (Chapter 3).

2.3 Fault Localization

Fault localization (FL) has been a widely researched topic for more than two decades [131, 136]. FL techniques relate program failures—e.g., crashes or assertion violations—to specific faulty locations within program's source code. Each FL technique assigns a *suspiciousness score* to different program entities—e.g., statements, functions, and files—based on the information collected during the execution of program's test suite. The output of a FL technique is a list of program entities along with their respective suspiciousness scores. Program entities with higher suspiciousness scores are more likely to be responsible for the failure. Developers or automated program repair techniques [44, 82] could use this list to decide where the fixes should be applied.

A fault localization *family* is a group of techniques that utilize the same kind of information to compute suspiciousness scores. Techniques in the spectrum-based fault localization (SBFL) family [6, 61, 130] compute suspiciousness scores based on program spectra [98], which is the execution trace of the program as it runs different tests (Section 5.2.1). The mutation-based fault localization (MBFL) family [83, 91] computes suspiciousness scores relying on mutation analysis [60] (Section 5.2.2). The predicate switching (PS) family [140] finds buggy predicates by examining how changing the state of different predicates at runtime affects the output of the program's failing tests (Section 5.2.3).

The stack trace (ST) family [145] uses the stack trace information of failed tests to localize bugs (Section 5.2.4).

The Tarantula SBFL technique [61] was one of the earliest, most influential FL techniques, also thanks to its empirical evaluation showing it is more effective than other competing techniques [25, 97]. The Ochiai SBFL technique [6] improved over Tarantula, and it often still is considered the "standard" SBFL technique. These earlier empirical studies [6, 61], as well as other contemporary and later studies of FL [91], used the Siemens suite [54]: a set of seven small C programs with seeded bugs. Since then, the scale and realism of FL empirical studies has significantly improved over the years, targeting real-world bugs affecting projects of realistic size. For example, Ochiai's effectiveness was confirmed [69] on a collection of more realistic C and Java programs [35]. When Wong et al. [130] proposed DStar, a new SBFL technique, they demonstrated its capabilities in a sweeping comparison involving 38 other SBFL techniques (including the "classic" Tarantula and Ochiai). In contrast, numerous empirical results about fault localization in Java based on experiments with artificial faults were found not to hold to experiments with real-world faults [94] using the Defects4J curated collection [63].

With the introduction of novel fault localization families—most notably, MBFL—empirical comparison of techniques belonging to different families became more common [83, 91, 94, 145]. The Muse MBFL technique was introduced to overcome a specific limitation of SBFL techniques: the so-called "tie set problem". This occurs when SBFL assigns the same suspiciousness score to different program entities, simply because they belong to the same simple control-flow block (see Section 5.2.1 for details on how SBFL works). Metallaxis-FL [91] (which we simply call "Metallaxis" in this dissertation) is another take on MBFL that can improve over SBFL techniques.

The comparison between MBFL and SBFL is especially delicate given how MBFL works. As demonstrated by Pearson et al. [94], MBFL's effectiveness crucially depends on whether it is applied to bugs that are "similar" to those introduced by its mutation operators. This explains why the MBFL studies targeting artificially seeded faults [83, 91] found MBFL to outperform SBFL; whereas studies targeting real-world faults [94, 145] found the opposite to be the case—a result also confirmed by the present dissertation in Chapter 5.

Summary

There are many fault localization techniques in the literature, which offer different trade offs between accuracy and performance [135]. On the other hand, despite Python's popularity as a programming language, the vast majority of fault localization empirical studies (which we discussed above) target other languages—mostly C, C++, and Java. To our knowledge, CharmFL [55, 121] is the only available implementation of fault localization techniques for Python; the tool is limited to SBFL techniques.

We could not find any realistic-size empirical study of fault localization using Python programs comparing techniques of different families on real-world data science or even traditional Python programs. This is a gap in both the availability of tools [109] (Chapter 6) and the empirical knowledge about fault localization in Python (Chapter 5).

2.4 Conclusions and Open Research Gaps

In this chapter, we summarized studies regarding three main areas related to our thesis statement: data science programs, automated test generation techniques, and fault localization techniques. We also provided an overview of open research gaps within each of these three areas with references to the respective chapters where we address them. These research gaps are as follows:

- Bugs in data science programs, as well as the inputs and tests to trigger them, are different in nature compared to traditional programs (Chapter 3).
- Traditional fault localization techniques may behave differently when applied to data science programs (Chapter 5).
- Curated collections of reproducible data science bugs are required to facilitate controlled debugging studies (Chapter 4).
- Test generation for data science programs requires a more expressive set of type annotations (Chapter 3).
- There are no realistic-size empirical study of fault localization comparing techniques from different families on real-world data science and traditional Python programs (Chapter 5).
- There are no available open-source fault localization tools for the Python programming language to support research in this area (Chapter 6).

Part II

Test Generation

3

The ANNoTest Test Generation Approach

Neural networks are increasingly being used in safety-critical systems, underscoring crucial needs for developing effective testing techniques specialized for them. The focus of most studies is on testing neural-network *models* (Section 2.1.2); but these models are defined by writing programs, and there is evidence that these neural-network (NN) programs often exhibit bugs [57, 142]. In this chapter, we present ANNoTest, an approach for generating test inputs for this type of programs.

NN programs are typically written in dynamically-typed languages (e.g., Python), which cannot express detailed constraints about valid function inputs (e.g., matrices with certain dimensions). Without knowing these constraints, automated test-case generation produces invalid inputs, triggering spurious failures that do not reflect real bugs. To address this challenge, we introduce a simple annotation language named AN (Section 3.3.1), tailored for concisely expressing valid function inputs in neural-network programs. ANNoTest takes as input an annotated program, and uses property-based testing to generate random inputs that satisfy the validity constraints. Writing AN annotations is a manual process; we present some simple guidelines in this chapter to facilitate writing them (Section 3.3.2).

We evaluated ANNoTest on 19 open-source NN programs sourced from Islam et al.'s survey [57], demonstrating ANNoTest capability at finding widespread bugs in real-world NN programs. Based on further analyses, we show that the manual effort of annotating programs in AN is reasonable and tests generated by ANNoTest have high coverage, comparable to those written by developers.

Structure of the Chapter

The current chapter is organized as follows:

Section 3.1 provides this chapter's introduction, including its motivation and scope.

Sections 3.2 and 3.3 present how ANNoTest works.

Sections 3.4–3.6 outline our experimental design.

Sections 3.7 and 3.8 presents our experimental results and the threats to the validity of our experimental evaluation.

Section 3.9 concludes this chapter and highlights possible avenues for future work.

3.1 Introduction

Neural networks have taken the (programming) world by storm. With their capabilities of solving tasks that remain challenging for traditional software, they have become central components of software systems implementing complex functionality such as image processing, speech recognition, and natural language processing, where they can reach performance at or near human level. These tasks are widely applicable to domains such as automotive and healthcare, where safety, reliability, and correctness are critical. Therefore, the software engineering (research) community has been hard at work designing techniques to assess and ensure the dependability of software with neural network (NN) components.

Testing techniques, in particular, are being extensively developed to cater to the specific requirements of NN (and, more generally, machine learning) systems [104]. Most of this research focuses on testing NN *models*: instances of a specific NN architecture, trained on some data and then used to classify or transform new data. Testing a NN model entails assessing qualities such as its robustness and performance as a classifier. However, neural networks are programs too: a NN model is usually implemented in a programming language like Python, using frameworks such as Keras or TensorFlow. As also stated in Chapter 1, there is clear evidence that these *neural network programs* tend to be buggy [53, 57]; therefore, a technique for finding these bugs would be practically very useful and complement the extensive work on NN model testing [138]. This chapter presents a novel contribution in this direction.

NN programs may seem simple by traditional metrics of complexity: for example, the average project size of the NN projects surveyed by Islam et al. [57] is just 2165 lines of code; and the majority of the bugs they found are relatively simple ones such as crashes and API misuses. Nevertheless, other characteristics make traditional test-case generation techniques ineffective to test such programs. NN programs are written in dynamically typed languages like Python, where the type of variables is unknown statically. Without this information, generating valid inputs is challenging for generic techniques such as random testing and genetic algorithms [78]. Even if type annotations were available, NN programs routinely manipulate complex data structures—such as vectors, tensors, and other objects—whose precise "shape" is not expressible with the standard types (integers, strings, and so on). As we demonstrate in Section 3.2 and Section 3.7.4, without such precise information automated test case generation tends to generate many invalid inputs that trigger spurious failures.

Overview of aNNoTest

This chapter presents aNNoTest: an approach to automatically generating bug-finding inputs for NN program testing. A key component of aNNoTest (described in Section 3.3) is aN: a simple annotation language to concisely and precisely express the valid inputs of functions in NN programs. The aN language supports expressing the kinds of constraints that are needed in NN programs (for example: a variable should be a vector of size from 2 to 5 with components that are positive integers). aN is also easily extensible to accommodate other constraints that a specific NN program may need to encode.

Given an annotated NN program, aNNoTest automatically generates unit tests for the program that span the range of valid inputs. To this end, the current implementation of aNNoTest uses property-based testing (more precisely, the Hypothesis [80] test-case generator). Using the aN language decouples specifying the constraints from the back-end used to generate the actual tests; therefore, different back-end tools could also be used that better suite the kinds of constraints used in a project's annotations.

Overview of Experimental Results

Sections 3.4–3.7 describes an extensive experimental evaluation of ANNOTEST, targeting 19 open-source NN programs, manually analyzed by Islam et al. [57], using some of the most widely used NN frameworks (Keras, TensorFlow, and PyTorch). After we manually annotated 24 functions, ANNOTEST generated tests triggering 63 known bugs reported by Islam et al. [57] for these functions, as well as 31 previously unknown bugs. To experiment with ANNOTEST's capabilities when used extensively, we also annotated all functions in two larger NN projects; the total of 330 annotations that we wrote enabled ANNOTEST to discover 50 bugs with only 6 false positives. These experiments demonstrate that ANNOTEST can be used both extensively on a whole project, and opportunistically on only a few selected functions that are critical. Since our evaluation is based on Islam et al. [57]'s extensive survey, it can assess ANNOTEST's capabilities of finding relevant bugs in real-world NN programs.

In other experiments, we quantify the amount of annotations needed by ANNOTEST, compare it to generic (non NN-specific) test-case generators for Python, as well as to developer-written tests, so as to better understand the trade-off between programmer effort and quality assurance benefits it offers.

Contributions

In summary, this chapter presents the following contributions:

- ANNOTEST: an approach for test-case generation geared to the characteristics of NN programs.
- AN: a simple annotation language capable of concisely expressing precise constraints on the valid inputs of functions in NN programs, with basic guidelines to use it.
- An experimental evaluation of ANNOTEST's bug-finding capabilities on 19 open-source NN projects surveyed by Islam et al. [57].
- For reproducibility, the implementation of ANNOTEST[1] and all experimental artifacts are publicly available.[2]

Scope

While ANNOTEST is applicable, in principle, to any Python programs—not just NN programs— it was designed to primarily cater to the characteristics of NN programs. As we will see concretely with Section 3.2's example, NN programs often involve complex constraints on their inputs, which are impossible or highly impractical to express using Python's type hints annotations. ANNOTEST provides annotations that go beyond type hints, and hence are especially useful for the kinds of constraints that we commonly find in NN programs.

On the other hand, being able to *express* complex constraints is not sufficient to build tests automatically; as we will see in Section 3.7.4, *generating* inputs that satisfy the constraints is challenging; simple strategies such as generating input at random and then filtering them using the constraints are mostly ineffective. ANNOTEST defines suitable generators for each of its constraints, so that valid inputs can be generated efficiently and automatically even for the complex combinations of input constraints that are common in NN programs.

[1] Chapter 6 presents the ANNOTEST tool in details.
[2] Replication package: `doi.org/10.6084/m9.figshare.19082558.v1`

```
1  def DenseNet(input_shape=None, dense_blocks=3, dense_layers=-1,
2               growth_rate=12, nb_classes=None, dropout_rate=None,
3               bottleneck=False, compression=1.0, weight_decay=1e-4,
4               depth=40):
5    if nb_classes == None:
6      raise Exception('Please define number of classes')
7    if compression <= 0.0 or compression > 1.0:
8      raise Exception('Compression must be between 0.0 and 1.0.')
9    if type(dense_layers) is list:
10     if len(dense_layers) != dense_blocks:
11       raise AssertionError('Dense blocks must be the same as layers')
12   elif dense_layers == -1:
13     dense_layers = (depth - 4) / 3    # Bug:  division / returns a float
14   # ... 23 more lines of code ...
```

Listing 3.1. An excerpt of function `DenseNet` from project DenseNet. The code has a bug on line 13.

```
15 @arg(input_shape): tuples(ints(min=20, max=70),
16                           ints(min=20, max=70),
17                           ints(min=1, max=3))
18 @arg(dense_blocks): ints(min=2, max=5)
19 @arg(dense_layers): anys(-1,
20                         ints(min=1, max=5),
21                         int_lists(min_len=2, max_len=5, min=2, max=5))
22 @arg(growth_rate): ints(min=1, max=20)
23 @arg(nb_classes): ints(min=2, max=22)
24 @arg(dropout_rate): floats(min=0, max=1,
25                           exclude_min=True, exclude_max=True)
26 @arg(bottleneck): bools()
27 @arg(compression): floats(min=0, max=1, exclude_min=True)
28 @arg(weight_decay): floats(min=1e-4, max=1e-2)
29 @arg(depth): ints(min=10, max=100)
30 @require(type(dense_layers) is not list or len(dense_layers) == dense_blocks)
```

Listing 3.2. AN annotations for function `DenseNet` in Listing 3.1.

3.2 An Example of Using aNNoTest

DenseNet[3] is a small Python library that implements densely connected convolutional networks [52] (a NN architecture where each layer is directly connected to every other layer) on top of the Keras framework. Listing 3.1 shows a slightly simplified excerpt of function `DenseNet`—the main entry point to the library—in an earlier version of the project.[4]

The complete implementation of function `DenseNet` comprises 34 lines of code (excluding comments and empty lines), and follows a straightforward logic: after checking the input arguments (code in Listing 3.1), it combines suitable instances of Keras classes to model a densely connected network, and finally returns a model object to the caller. Listing 3.1's code, however, has a bug at line 13—one of the bugs collected in Islam et al. [57]'s survey. The expression assigned to `dense_layers` is a floating point number because the division operator `/` always returns a float in Python 3; however, if `dense_layer` is not an integer, a later call in `DenseNet`'s code to the Keras library fails. DenseNet's developers discovered the bug and fixed it (by adding an **`int`** conversion at

line 13) in a later project revision.{5}

`DenseNet`'s implementation is deceptively simple: despite its small size and linear structure, it only accepts input arguments in very specific ranges. Argument `input_shape`, for example, corresponds to a so-called *shape tuple* of integers; in `DenseNet`, it should be a triple of integers with first element at least 20. If the first element is less than 20, `DenseNet` eventually fails while trying to create a layer with a negative dimension—which violates an assertion of the Keras library. Another example is argument `dense_layers`, which can be an integer or an integer list; if it is the latter, its length must be equal to argument `dense_block`, or `DenseNet` terminates at line 11 with an assertion violation.

Without knowing all these details about valid inputs, testing `DenseNet` using a general-purpose automated test-case generator would trigger lots of spurious failures[3] when executing tests that call `DenseNet` with invalid inputs. The few failing but valid tests that trigger bugs such as that in Listing 3.1 would be a needle in the haystack of all invalid tests, thus essentially making automated test-case generation of little help to speed up the search for bugs.

To precisely and concisely express the complex constraints on valid inputs that often arise in NN programs, we designed the AN annotation language—which is a central component of the ANNOTEST approach. Listing 3.2 shows annotations written in AN[4] that characterize `DenseNet`'s valid inputs. Whereas Section 3.3 will present AN's features in greater detail, it should not be hard to glean the meaning of the annotations in Listing 3.2. For example, the first annotation encodes the aforementioned constraint on `input_shape`, and the last one expresses the relation between `dense_layers` and `dense_blocks` when the former is a list. It should also be clear that AN's expressiveness is much greater than what is allowed by the standard programming-language types—such as Python's type hints.

Equipped with the annotations in Listing 3.2, ANNOTEST generates and runs 36 unit tests for `DenseNet` in 53 seconds. All the tests are valid, and only one is failing, reaching Listing 3.1's line 13 and then ending with a failure due to `dense_layers` being a float that we described above—precisely revealing the bug.

The experimental evaluation of ANNOTEST—described in Sections 3.4–3.7—will analyze many more NN programs whose characteristics, input constraints, and faulty behavior are along the same lines as the example discussed in this section. This will demonstrate ANNOTEST's capabilities of precisely testing and finding bugs in NN programs.

3.3 How ANNOTEST works

Figure 3.1 overviews the overall process followed by the ANNOTEST approach. To test a NN program with ANNOTEST, we first have to annotate its functions (including member functions, that is methods) using the AN annotation language (Section 3.3.1). This is the only step that is manual, since the annotations have to encode valid inputs of the tested functions—the same kind of information that is needed to write unit tests. Section 3.3.2 provides guidelines that help structure the manual annotation process so that it only requires a reasonable amount of effort; furthermore, users do not need to annotate a whole program but only those functions that they want to test with ANNOTEST.

Then, the ANNOTEST tool takes as input an annotated program and generates unit tests for it. To this end, it encodes the constraints expressed by the AN annotations in the form of test templates for

[3] For example, Pynguin [78] generates 8 tests, all invalid and none triggering the failure at line 13. With type hints (supported by Pynguin), it generates 5 tests, 4 invalid and none triggering (any) failure. Section 3.7.4 describes more experiments with Pynguin. (As we discuss in Section 3.5, Pynguin doesn't work with the version of TensorFlow used by Listing 3.1's code; thus, we mocked the relevant library calls in this example.)

[4] The AN annotations in this dissertation use a pretty-printed and slightly simplified syntax.

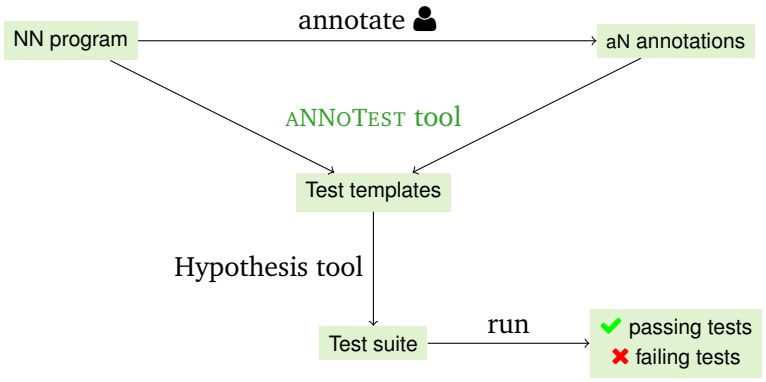


Figure 3.1. An overview of how the ANNOTEST approach works.

the property-based test-case generator Hypothesis (Section 3.3.4); then, it runs Hypothesis which takes care of generating suitable tests. Finally, the generated unit tests can be run as usual to find which are passing and which are failing—and thus expose some bugs in the NN program (Section 3.3.5).

3.3.1 The AN Annotation Language

By writing annotations in the AN language, developers can precisely express the valid inputs of a function in a NN program.[5] To this end, AN provides *type* annotations and *preconditions* as well as an extension mechanism to define arbitrarily complex constraints. In addition, AN offers a few *auxiliary* annotations, which encode other kinds of information that is practically useful for test-case generation.

Type Annotations

A *type* annotation follows the syntax `@arg(v):`*T*, where `v` is a function argument (parameter), and *T* is a *type constraint* that specifies a set of possible values for `v`. A type annotation refers to the function that immediately follows it in the source code. A function can have up to as many type annotations as it has arguments.

AN supports several different *type constraints*, which can express a broad range of constraints—from simple ones, such as those that are also expressible using Python's type hints, up to complex instances of special-purpose classes. The simplest, and most specific, type constraint uses keyword **`froms`**[6] to enumerate a list of valid values. For example, constraint **`froms`**`([0, 0.0, None, zero()])` corresponds to any of the four values: integer zero, floating-point zero, `None`, and what is returned by the call `zero()`.

Constraints for **atomic types** specify that an argument is a Boolean (**`bools`**), an integer number (**`ints`**), or a floating-point number (**`floats`**). Integer arguments can be restricted to a range between `min` and `max` values; for example, Listing 3.2's line 23 constrains `nb_classes` to be an integer between 2 and 22. Floating-point arguments can also be restricted to ranges, and the ranges can be

[5] Directly annotating the source code, rather than having a separate generator used only when testing, also helps keep the annotations consistent with the implementation.

[6] AN type constraints use names that are "pseudo-plurals" (by adding a trailing `s`) of the corresponding Python types. This avoids using reserved keywords and also conveys the idea that a type constraint identifies a set of values. This convention is also customary in property-based testing [24].

```
31 @arg(k): ints(min=1, max=1000)
32 @arg(w): ints(min=1, max=1000)
33 @arg(kwargs): dicts(keys=froms(["input_shape"]),
34                     values=np_shapes(min_dims=1, max_dims=1))
35 def dim_ordering_reshape(k, w, **kwargs):
```

Listing 3.3. An example of AN annotations for a function with keyword arguments.

open, closed, or half-open; for example, Listing 3.2's line 27 constrains `compression` to be a number in the half-open interval $(0, 1]$ which includes 1 but excludes 0. Floating-point constraints also support including or excluding the special values `NaN` and `Inf`, as well as the precision (in bits) of the generated floating point values.

Constraints for **sequences** specify that arguments are Python **`lists`**, **`tuples`**, or an array in the NumPy[7] library (which is widely used in NN programs, as well as other data-intensive applications). Lists and tuples can have any number of elements, whose possible values are also constrained using AN's type constraints. For example, Listing 3.2's line 15 specifies a tuple with 3 integer elements: the first and second one between 20 and 70, and the third one between 1 and 3. AN also includes shorthands for lists with homogeneous elements: Listing 3.2's line 21 uses shorthand **`int_lists`** to specify lists of length between 2 and 5, whose elements are integers between 2 and 5.

The *shape* of a NumPy array is a tuple of positive integers that characterize its size. For example, the tuple `(256, 256, 3)` is the shape of a 3-dimensional array whose first two dimensions have size 256 and whose last dimension has size 3; arrays with this shape can represent 256x256 pixel color pictures. Type constraint **`np_shapes`** specifies arguments that represent shapes with a certain range of possible dimensions and sizes. For example, **`np_shapes`**`(min_dims=3, max_dims=3)` are the shapes of all 3-dimensional arrays whose dimensions can have any size.

Type constraint **`np_arrays`** specifies NumPy array arguments with any shape and whose elements have any of the valid NumPy types. The shape can be constrained by an **`np_shapes`** annotation or given directly as a tuple. For example, using the shape mentioned in the previous paragraph, **`np_arrays`**`(np_type=dtype("uint32"), shape=(256,256,3))` specifies 256x256x3 arrays whose components are unsigned 32-bit integers (one of NumPy's data-types), which could represent random color pictures.

Type constraints for **maps** specify Python's widely used associative dictionaries: **`dicts`**$(K, V,$ `min_size, max_size`$)$ corresponds to all subsets of the Cartesian product $K \times V$ with between `min_size` and `max_size` elements, where K and V are type constraints that apply to the keys and values respectively. A typical usage of this is to constraint Python's optional keyword argument `**kwargs`. For example, Listing 3.3 shows how we used **`dicts`** to constrain the `**kwargs` argument of function `dim_ordering_reshape`[{6}] (from a project using NN models to simulate multi-player games), so that it simply consists of all mappings from string `"input_shape"` to singletons representing the shapes of monodimensional arrays.

To express the **unions** of several type constraints, AN includes the **`anys`** type constraint, which specifies the union of its arguments. For example, Listing 3.2's line 19 says that `dense_layers` can be any of: *i*) the number -1, *ii*) an integer between 1 and 5, or *iii*) an integer list with between 2 and 5 elements that are between 2 and 5.

[7] NumPy: `https://numpy.org`

```
36 @arg(generator): objs(gan_gens)
37 @arg(discriminator): objs(gan_discs)
38 @arg(name): froms(["gan1", "gan2", "gan3", "gan4", "gan5"])
39 def build_gan(generator, discriminator, name="gan"):
40   # ...
41
42 @generator
43 @exclude
44 @arg(latent_dim): ints(min_value=1, max_value=1000)
45 @arg(input_shape): np_shapes(min_dims=2)
46 def gan_gens(latent_dim, input_shape):
47   from examples.example_gan import model_generator
48   generator = model_generator(latent_dim, input_shape)
49   return generator
```

Listing 3.4. An example of using type constraint **objs** and a custom generator function.

Custom Generators

While AN's type annotations can define a broad range of frequently used constraints, they cannot cover all cases that one may encounter in practice. To support *arbitrary* type constraints, AN includes the **objs**(gen) annotation. This is used as a type constraint, and identifies all values that are produced by the user-provided *generator* function gen. Function gen must be visible at the entry of the functions whose annotations refer to it; gen itself is marked with the annotation **@generator**.

For instance, Listing 3.4 shows the annotations we wrote for function build_gan{7} (from the same project as Listing 3.3). The function combines two Keras model instances, generator[8] and discriminator, to build GANs (Generative Adversarial Networks [45]). These instances are complex objects that are built by calls to the Keras library; therefore, we introduced two custom generators, gan_gens and gan_discs, that construct such instances for testing build_gan. Listing 3.4 shows gan_gens's implementation: the generator's input are constrained by using AN's type annotations as usual; ANNOTEST will use gan_gens's output as input for build_gan.

Whereas generators such as gan_gens may look daunting to write at first, we found that they simply encapsulate existing snippets of the project that call the function under test (build_gan in Listing 3.4). Based on this observation, Section 3.3.3 presents a simple process to build generators by combining common refactoring steps; this drastically alleviates the effort to write generators, reducing it to just selecting the right snippets of client code in the project.

Preconditions

Argument annotations constrain each function argument individually. *Preconditions* may express constraints that affect multiple arguments simultaneously: **@require**(*P*), where *P* is a Python Boolean expression, specifies that a function's arguments must be such that *P* evaluates to true. A precondition refers to the function that immediately follows it in the source code. Expression *P* may refer to any arguments of the specified function, as well as to any other program element that is visible at the function's entry (such as other class members). A function can have any number of preconditions, all of which constraint the function's argument. For example, Listing 3.2's line 30 requires that, whenever argument dense_layer is a list, it should have as many elements as the value of integer argument dense_blocks.

[8] It is just a coincidence that one argument is also named "generator".

```
50 @arg(image_path): froms(["image1.png", "image2.png",
51                          "image3.png", "image4.png"])
52 @arg(generator): objs(grids)
53 @arg(cmap): froms(['gray', 'bone', 'pink', 'spring', 'summer', 'cool'])
54 @cc_example(["image1.png", grids(3, 6, 6, 3), 'gray'])
55 def __init__(self, image_path, generator, cmap='gray'):
56   # ...
```

Listing 3.5. An example of using the `cc_example` auxiliary annotation on the constructor of class `ImageGridCallback`.

Auxiliary Annotations

The AN language includes a few more features to control the test-generation process. Functions marked with **`@exclude`** are not tested (such as generator `gan_gens` in Listing 3.4). Annotation **`@timeout`** introduces a timeout to the unit tests generated for the function it refers to.

Python modules may include snippets of code that is not inside any functions or methods but belongs to an implicit "main" environment. ANNOTEST will generate tests for this environment for any module that is annotated with **`@module_test`**. Since modules don't have arguments, these tests simply import and execute the main environment. This is a simple feature, but practically useful since some of the NN program bugs that were surveyed in [57] are located in the main environment.

To test an instance method `m`, one needs to generate an instance `o` of `m`'s class `C` to use as target of the call to `m`. To this end, `C`'s constructor is called. The constructor may also be equipped with AN annotations; as a result, testing `m` entails also testing `C`'s constructor. This can be a problem if the constructor has bugs that prevent a correct execution of `m`. To handle this scenario, AN includes the annotation **`@cc_example`**, which supplies a constructor with a list of concrete inputs for it. If `C`'s constructor is equipped with this annotation, ANNOTEST will only call it using the inputs given by the **`@cc_example`** annotation when it needs to create instances to test any methods of `C`. This way, one can effectively decouple testing a class's constructor from testing the class's (regular) methods, so that any bugs in the former do not prevent testing of the latter. For example, the constructor of class `ImageGridCallback`[8] shown in Listing 3.5 is regularly tested through its type annotations; however, when it is used to construct instances of the class to test other methods, it is only called with the more restricted set of inputs specified by the **`@cc_example`** annotation. The example also demonstrates that a generator function (`grids` in this case) can also be used as a regular function (second component of **`@cc_example`**).

3.3.2 Annotation Guidelines

To test a NN program using ANNOTEST, one must first annotate the functions to be tested using the language described in Section 3.3.1. Ultimately, writing suitable annotations requires knowledge about the program's specification—that is, its intended behavior. The very same knowledge is necessary to write *unit tests* for the programs; the only difference is that a test supplies individual (valid) inputs, whereas an annotation can capture a range of possible (valid) inputs.

This entails that the effort of writing annotations (or tests) for a project depends on whether the programmer already has this knowledge—typically, because they are developers of the project under test—or is trying to test a project they are not familiar with. In this section, we focus on the latter, more challenging scenario. To help such a process of "discovery"—figuring out suitable annotations for NN programs written by others—and to make it cost-effective, we present some simple guidelines that suggest which artifacts to inspect and in which order. In the experiments described in Sections

SOURCE	ANNOTATIONS
1 calls of `f` in its project `P`	basic type annotations `@arg`
2 assertions and exceptions raised by `f`'s implementation	refined type annotations `@arg`, preconditions `@require`
3 calls of NN framework functions in `f`'s implementation	refined type annotations `@arg`, preconditions `@require`, custom generators
4 calls of other functions `g` in `P`	annotations of `g`

Table 3.1. Guidelines to inspect the implementation of a NN function `f` to suggest how to annotate it using AN's annotation language. Each SOURCE of information in `f` or elsewhere in `f`'s project `P` suggests matching AN ANNOTATIONS.

3.4–3.7, we followed these guidelines to annotate NN projects systematically and with reasonable effort—despite our previous lack of familiarity with those codebases.

Consider a Python function `f` in some NN project `P` that we would like to test. If `f`'s behavior (and, in particular, the constraints on its inputs) is documented in the project, this documentation should be the first source of information to write AN annotations. However, if `f` lacks any (precise) documentation,[9] we will have to inspect its implementation. Table 3.1 lists four sources of information about `f`'s valid inputs in increasing level of detail.

To bootstrap the process, we inspect any usage of `f` within the NN program `P`. Since we focus on testing *programs*, not libraries, it's likely that every major function is called somewhere in `P`. These calls of `f` provide basic examples of valid inputs, which we loosely encode using AN's type annotations of Section 3.3.1. In Listing 3.1's example, looking at usages of `DenseNet` indicates that `input_shape` should be a triple of **`int`**, `compression` should be a **`float`**, and so on.[10]

Next, we look into `f`'s implementation for any (implicit or explicit) *input validation*. Often, a function uses exceptions or assertions to signal invalid input arguments. This information is useful to *refine* the basic type annotations, and may also suggest constraint that involve multiple arguments—which we can encode using AN's preconditions of Section 3.3.1. In Listing 3.1's example, `DenseNet`'s initial validation clearly indicates, among other things, `compression`'s precise interval of validity, and the precondition on line 30 in Listing 3.2.

The library functions from some NN framework used in `f`'s implementation may also (indirectly) introduce requirements on `f`'s inputs or otherwise suggest plausible ranges of variability. Indirect constraints may be more complex, and may even require custom generators (Section 3.3.3). In the running example, a call to Keras's `Convolution2D` constructor in `DenseNet` (not in Listing 3.1) suggests the range for argument `weight_decay` at line 28 in Listing 3.2.

Whenever `f`'s implementation calls other functions in the same project, this process can be repeated for these other functions, thus ensuring the consistency of the other functions' and `f`'s annotations. In the running example, `DenseNet` calls in a loop another function `dense_block` in the same project, passing `growth_rate` as argument and then incrementing it in each iteration. The input constraints of `dense_block`, once figured out, indirectly suggest the validity range for `DenseNet`'s `growth_rate` at line 22 in Listing 3.2.

The guidelines we described are flexible and remain useful even if they are not followed in full. For example, sometimes we found it useful to start from very narrow annotations (merely encoding the available examples of usages of `f` in `P`) and relax them as we discovered more information—rather

[9] Many of the NN programs we used in Sections 3.4–3.7's experiments are sparsely documented.

[10] For example, the `README.md` file in DenseNet's repository presents an example of using function `DenseNet` where argument `input_shape` is set to the triple `(28, 28, 1)`.

```
57 @arg(net): objs(generator_G_convblock)
58 def G_convblock(net, num_filter, filter_size, actv, init, pad='same',
59                 use_wscale=True, use_pixelnorm=True, use_batchnorm=False,
60                 name=None):
61   # ... 24 lines of body code ...
```

Listing 3.6. Signature of project GANS's function `G_convblock`, whose first argument `net` requires a custom generator.

than going from basic to specific as we did in most examples—since this allowed us to generate some sample tests early on. The guidelines are also applicable with different levels of exhaustiveness, regardless of whether your goal is to annotate as much as possible in a project, or just test a few selected functions. In the former case, it is advisable to start annotating the simplest, shortest functions, so that their annotations can then suggest how to annotate the more complex, longer ones.

3.3.3 Building Custom Generators by Refactoring

As presented in Section 3.3.1, annotation `@arg(a): @objs(f)` tells ANNOTEST to use a *custom generator* function `f` in order to build suitable inputs for some argument `a`. In principle, `f` may be an arbitrarily complex piece of code; in practice, we found that the very projects we are annotating already include snippets of code that can be reused as generators of complex objects. In this section, we demonstrate, on an example, how to build such generators by applying a few refactoring operations to the relevant snippets of code. Modern IDEs such as PyCharm[11] can automate such refactoring steps. This drastically reduces the effort of building custom generators to just selecting the right snippets of code and doing some copy-pasting in the IDE.

Listing 3.6 shows the signature of function `G_convblock`{9} in project GANS (described in Section 3.5); the first function argument `net` expects objects encoding Keras network architectures. This complex type is not directly supported by AN's built-in annotations; thus, we should define a custom generator function `generator_G_convblock` that builds valid instances of the type.

To this end, we first look for any *client code* of `G_convblock`. Another function `Generator` in project GANS, shown in Listing 3.7, calls `G_convblock` (line 87) after building a suitable network architecture object (line 85). Thus we can use parts of `Generator` to build `generator_G_convblock`: the "extract function" refactoring[12] applied to lines 77–85 in Listing 3.7 outputs Listing 3.8's generator function. Now, `generator_G_convblock` is a new function, which we can annotate like any other functions that is processed by ANNOTEST.

In this example it was easy to identify a contiguous sequence of statements and extract it into a generator function. In other cases, the relevant client code may mix statements useful for the generator with others that pertain to a different functionality. In these cases, we can simply extract a larger snippets of code, and then refactor it to remove unused statements. In Listing 3.7's example, we could extract all lines 67–85 into a new function; then, all statements before line 77 are not used by the final line 85, and thus can be removed from the generator (leading to the same generator as in Listing 3.8). In all the experiments of this chapter, these simple refactoring steps were sufficient to build all necessary custom generator functions.

[11] The PyCharm Python IDE: `https://www.jetbrains.com/pycharm`
[12] Extract function refactoring: `https://refactoring.com/catalog/extractFunction.html`

```
62  def Generator(num_channels=1, resolution=32, label_size=0,
63                fmap_base=4096, fmap_decay=1.0, fmap_max=256,
64                latent_size=None, normalize_latents=True, use_wscale=True,
65                use_pixelnorm=True, use_leakyrelu=True,
66                use_batchnorm=False, tanh_at_end=None, **kwargs):
67    R = int(np.log2(resolution))
68    assert resolution == 2 ** R and resolution >= 4
69    cur_lod = K.variable(np.float32(0.0), dtype='float32', name='cur_lod')
70
71    def numf(stage): return min(int(fmap_base /
72                                    (2.0 ** (stage * fmap_decay))), fmap_max)
73    if latent_size is None:
74      latent_size = numf(0)
75    (act, act_init) = (lrelu, lrelu_init) if use_leakyrelu else (relu, relu_init)
76
77    inputs = [Input(shape=[latent_size], name='Glatents')]
78    net = inputs[-1]
79
80    if normalize_latents:
81      net = PixelNormLayer(name='Gnorm')(net)
82    if label_size:
83      inputs += [Input(shape=[label_size], name='Glabels')]
84      net = Concatenate(name='G1na')([net, inputs[-1]])
85    net = Reshape((1, 1,K.int_shape(net)[1]), name='G1nb')(net)
86
87    net = G_convblock(net, numf(1), 4, act, act_init, pad='full',
88                      use_wscale=use_wscale, use_batchnorm=use_batchnorm,
89                      use_pixelnorm=use_pixelnorm, name='G1a')
90    # ... 20 more lines of code ...
```

Listing 3.7. An excerpt of project GANS's function `Generator`, a client of Listing 3.6's function `G_convblock`.

```
91  @generator
92  @exclude
93  @arg(latent_size): ints(min=1)
94  @arg(normalize_latents): bools()
95  @arg(label_size): ints()
96  def generator_G_convblock(label_size, latent_size, normalize_latents):
97    inputs = [Input(shape=[latent_size], name='Glatents')]
98    net = inputs[-1]
99    if normalize_latents:
100     net = PixelNormLayer(name='Gnorm')(net)
101   if label_size:
102     inputs += [Input(shape=[label_size], name='Glabels')]
103     net = Concatenate(name='G1na')([net, inputs[-1]])
104   net = Reshape((1, 1, K.int_shape(net)[1]), name='G1nb')(net)
105   return net
```

Listing 3.8. The custom generator for argument `net` of Listing 3.6's function `G_convblock`, built by factoring out lines 77–85 in Listing 3.7.

3.3.4 Test Generation

The annotations written in the AN language supply all the information that is needed to generate unit tests for every annotated function. In principle, we could use any technique for test-case generation and then filter any generated tests, keeping only those that comply with the annotations. However, the experiments reported in Section 3.7.4 indicate that such an aimless strategy would be inefficient, especially given the dynamically typed nature of Python.

Instead, ANNOTEST uses *property-based test-case generation* to actively match the constraints introduced by AN annotations. More precisely, the current implementation of ANNOTEST uses the Hypothesis property-based test-case generator [80] through its API. To test a Python function using Hypothesis, we have to write a *test template*, which consists of a *parametric* unit test method that calls a collection of *strategies*. A strategy is a sort of generator function, which outputs values of a certain kind. A parametric test method calls some of the strategies, combines their outputs, and uses them to call the function under test.

ANNOTEST automatically builds a suitable Hypothesis strategy for each `@arg` annotation. Hypothesis provides built-in strategies that cover basic type annotations, such as Python's atomic types and tuples. ANNOTEST reuses the built-in strategies whenever possible, and combines them to generate values for more complex or specialized constraints (such as **`int_lists`**). For instance, Listing 3.9 shows parts of the parametric tests generated by ANNOTEST to encode the annotations in Listing 3.2's running example. ANNOTEST reuses Hypothesis's built-in strategies `integers` (line 108) and `floats` (line 115); and combines Hypothesis strategies `lists` and `integers` (lines 131–137) to render AN's **`int_lists`** type constraint.

To encode arbitrary **`objs`** annotations (Section 3.3.1), ANNOTEST first builds strategies for the annotations of each user-written custom generator function, as if it was testing the generator; then, it combines them to build a new strategy that follows the generator's implementation to output the actual generated objects—used as inputs for the function under test.

To encode `@require` annotations (preconditions), ANNOTEST uses Hypothesis's `assume` function. When test-case generation reaches an `assume`, it checks whether its Boolean argument evaluates to true: if it does, generation continues as usual; if it does not, the current test input is discarded, and the process restarts with a new test. Thus, `assume`s can effectively act as filters to further discriminate between test inputs—a feature that ANNOTEST leverages to enforce precondition constraints where appropriate in a parametric test. Line 126 in Listing 3.9 shows an example of using `assume` to encode the running example's precondition (line 30 in Listing 3.2).

After translating the annotations into suitable test templates, ANNOTEST simply runs Hypothesis on those templates. The property-based test-case generator "runs" the templates to build unit tests that satisfy the encoded properties; it also runs these unit tests, and reports any failure to the user. Hypothesis's output is also ANNOTEST's final output to the user.

Alternative back-ends. ANNOTEST's current implementation uses Hypothesis as back-end, since property-based testing is a framework for defining testing properties in a naturally *generative* way. However, using other test-input generation engines as back-end is possible in principle. Automatically translating all AN annotations to preconditions (Boolean predicates) is straightforward, which could be passed to a tool like Deal [29]. As we demonstrate in Section 3.7.4, Deal is not very effective at *generating* inputs that satisfy the preconditions, when these encode the complex combinations of constraints that are common in NN programs; however, Deal can also use preconditions for *static checking*, which would provide a complementary usage of ANNOTEST's annotations. Pynguin [78] is a general-purpose test-case generator for Python. In order to use it as a back-end for ANNOTEST, we could leverage its genetic algorithm, which tries to maximize the *branch coverage* of the tests it

```
@given(input_shape=tuples(integers(min_value=20, max_value=70),
                          integers(min_value=20, max_value=70),
                          integers(min_value=1, max_value=3)),
       dense_blocks=integers(min_value=2, max_value=5),
       dense_layers=one_of(just(-1),
                           integers(min_value=1, max_value=5),
                           int_lists_an(min_len=2, max_len=5, min=2, max=5)),
       growth_rate=integers(min_value=1, max_value=20),
       nb_classes=integers(min_value=2, max_value=22),
       dropout_rate=floats(min_value=0, max_value=1,
                           exclude_min=True, exclude_max=True),
       bottleneck=booleans(),
       compression=floats(min_value=0, max_value=1, exclude_min=True),
       weight_decay=floats(min_value=0.0001, max_value=0.01),
       depth=integers(min_value=10, max_value=100))
@settings(deadline=None, suppress_health_check=[HealthCheck.filter_too_much,
                                                HealthCheck.too_slow])
def test_DenseNet(input_shape, dense_blocks, dense_layers, growth_rate,
                  nb_classes, dropout_rate, bottleneck, compression,
                  weight_decay, depth):
  assume(type(dense_layers) is not list or len(dense_layers) == dense_blocks)
  DenseNet(input_shape, dense_blocks, dense_layers, growth_rate,
       nb_classes, dropout_rate, bottleneck, compression,
       weight_decay, depth)

@defines_strategy()
def int_lists_an(min_len=1, max_len=None, min=1, max=None):
  if max_len is None:
    max_len = min_len + 2
  if max is None:
    max = min + 5
  return lists(integers(min, max), min_size=min_len, max_size=max_len)
```

Listing 3.9. Hypothesis test template built by ANNOTEST for `DenseNet`'s annotations in Listing 3.2.

generates. As done with EvoSuite (a test-case generation tool for Java that is also based on genetic algorithms) in related work [40, 43], one could express the input constraints as a series of branches in the instrumented program, so that Pynguin would be driven to find inputs that "pass" all the constraints—the valid inputs that we are looking for.

3.3.5 Failing Tests and Oracles

The ANNOTEST approach, and the AN annotation language on which it is based, works independent of how a test is classified as failing or passing. In other words, ANNOTEST generates test inputs that are consistent with the annotations; determining whether the resulting program behavior is correct requires an *oracle* [14]. In our study, we only ran the tests generated by ANNOTEST with *crashing* oracles: an execution is *failing* when it cannot terminate normally, that is it leads to an assertion violation, an unhandled exception, or some other low-level abrupt termination.

While crashing bugs are the most frequent ones, NN programs also exhibit other kinds of bugs such as performance loss, data corruption, and incorrect output [57]. In principle, if we equipped the NN programs with oracles suitable to detect such kinds of bugs, ANNOTEST could still be used to

generate test inputs. However, some of these bug categories may be easier to identify by testing a NN at a different level than the bare program code. For example, bugs that lead to poor robustness of a NN classifier involve testing a fitted model rather than the model's implementation [51, 114, 120]. Revisiting the ANNOTEST approach to make it applicable to different kinds of oracles belongs to future work.

3.4 Research Questions

The experimental evaluation in this chapter aims at determining whether the ANNOTEST approach is effective at detecting real bugs in NN programs, and whether it requires a reasonable annotation effort. Precisely, we address the following research questions:

RQ1. Does ANNOTEST generate tests that expose bugs with few false positives (invalid tests)?

RQ2. Can ANNOTEST reproduce known, relevant bugs (that were discovered and confirmed by expert manual analysis)?

RQ3. How many annotations does ANNOTEST need to be effective?

RQ4. How does ANNOTEST compare to other generic (non-NN specific) test-case generation techniques?

RQ5. How does ANNOTEST compare to manual-written tests in terms of coverage?

3.5 Experimental Subjects

To include a broad variety of real-world NN projects, we selected our experimental subjects following Islam et al. [57]'s extensive survey of bugs and their replication package,[13] which collects hundreds of NN program bugs from Stack Overflow posts and public GitHub projects. The former are unsuitable to evaluate ANNOTEST, since they usually consist of short, often incomplete, snippets of code that punctuate a natural-language text. In contrast, the GitHub projects provide useful subjects for our evaluation.

The survey [57] lists 557 bugs in 127 GitHub projects using the NN frameworks Keras, TensorFlow, PyTorch, Theano, and Caffe. With 350 bugs in 42 projects, Keras is the most popular project in this list; we target it for the bulk of our evaluation. Starting from all 42 Keras projects, we excluded: *i*) 3 projects that were no longer publicly available; *ii*) 7 projects with no bugs classified as "crashing" (see Section 3.3.5); *iii*) and 5 projects that still use Python 2. While it could be modified to run with Python 2, we developed ANNOTEST primarily for Python 3, which is the only supported major version of the language at the time of writing. We excluded another 4 projects whose repositories were missing some components necessary to execute them (such as data necessary to train or test the NN model, or to otherwise run the NN program). Finally, 7 projects did not include any reproducible crashing bugs (see Section 3.6 for how we determined these). This left 16 projects using Keras, which we selected for our evaluation.

To demonstrate that ANNOTEST is applicable also to other NN frameworks, we also selected 2 projects based on TensorFlow and 1 project based on PyTorch; these are among the largest projects using those frameworks analyzed by Islam et al. [57]. The leftmost columns of Table 3.3 list all selected 19 projects used in our evaluation, and their size in lines of code and number of functions.

[13] Islam et al.'s NN bugs dataset: `https://lab-design.github.io/papers/ESEC-FSE-19`

These projects (and their known bugs) are based on Islam et al. [57]'s detailed survey of real-world NN bugs; this ensures that our subjects are representative of realistic NN programs and of the bugs that commonly affect them.

Comparison with Pynguin. To answer RQ4, we want to compare ANNOTEST to Pynguin (a general-purpose test-case generator for Python programs) on generating tests for realistic NN programs. Unfortunately, all the NN projects that we use for ANNOTEST's evaluation are incompatible with Python 3.8 (mainly because they require TensorFlow 1.x), whereas Pynguin only runs with Python 3.8 (or later versions). Therefore, we considered PyTorch's machine vision project Vision:[14] an actively maintained open-source NN program that is compatible with Python 3.8 and includes type hints (used by Pynguin). Pynguin can only generate tests for 40 of Vision's 104 modules; current limitations[15] of its implementation prevent it from running correctly on the other 64 modules. For our experiments, we selected module `mnist`{11} in package `torchvision.dataset`—one of the largest among those that Pynguin can analyze.

Comparison with manual tests. Manually writing AN annotations, and then letting ANNOTEST generate tests automatically, is an alternative to the usual approach of writing unit tests manually. Thus, RQ5 compares manually-written tests to those generated by ANNOTEST in terms of coverage. Unfortunately, none of the 19 projects selected by Islam et al. [57] contains any unit tests.[16] Therefore, we resorted to project Vision again, as it contains an extensive manually-written test suite. For our experiments, we selected three Vision modules of substantial size that are tested in different ways: module `backbone_utils` is among the most thoroughly tested (the project's test suite reaches 96% branch coverage); module `image` is fairly well tested (79% branch coverage, which is an average coverage figure among the project's modules); and module `_video_opt` is scarcely tested (16% branch coverage, and is only tested indirectly by the unit tests of other client modules).

3.6 Experimental Setup

This section describes how we setup each project before applying ANNOTEST; and the experiments we conducted to answer the RQs.

3.6.1 Project Setup

As first step, we created an Anaconda[17] environment for each project in Section 3.5 to configure and run it independent of the others. Every project has *dependencies* that involve specific libraries. Collecting all required dependencies can be tricky: a project may work only with certain library versions, older versions of a library may no longer be available, and newer backward-compatible versions may conflict with other dependencies. A handful of projects detail the specific versions of the libraries they need in a `setup.py`, `requirements.txt`, or Jupiter Notebook file—or at least in a human-readable `readme`. In many cases, none of these were available, so we had to follow a trial-and-error process: *i*) search the source code for **`import`** `L` statements; *ii*) retrieve the version of library `L` that was up-to-date around the time of the project's analyzed commit; *iii*) in case that

[14] Vision (0.11.2): `https://github.com/pytorch/vision/tree/v0.11.2`
[15] Including bugs, one of which we reported to Pynguin's maintainers who fixed it.{10}
[16] Project ADV includes a single integration test; the other projects include no tests at all.
[17] Anaconda: `https://www.anaconda.com`

version is no longer available or conflicts with other libraries, try a slightly more recent or slightly older version of L.

NN programs usually need *datasets* to run. When a suitable dataset was not available in a project's repository, we inspected the source code and its comments to find references to public datasets that could be used, fetched them, and added them to the project's environment. In a few cases, the project included functions to generate a sample dataset, which was usually suitable to be able to at least test the project. For a few projects using very large datasets, we shrank them by removing some data points so that certain parts of the project's code ran more efficiently. Whenever we did this, we ascertained that using the modified dataset did not affect general program behavior in terms of *reachability*—which is what matters for detecting the crashing bugs that we target in our evaluation.

Properly setting up all NN programs so that they can be automatically run and tested was quite time-consuming at times, since several of the projects' repositories are incomplete, outdated, and poorly documented. Our replication package includes all required dependencies, which can help support future work in this area.

3.6.2 Experimental Process

To address RQ1, we selected the latest versions of two projects among the largest and most popular ones (ADV and GANS in Table 3.2) and followed the guidelines described in Section 3.3.2 to fully annotate them with AN. "Fully annotate" means that we tried to annotate every function of the project's source code, and to write annotations that are as accurate as possible: neither unnecessarily constraining (skipping some valid inputs) nor too weak (allowing invalid inputs).

To address RQ2, we tried to use ANNOTEST to reproduce the bugs reported by Islam et al. [57] for the selected projects. More precisely, Islam et al. [57]'s companion dataset identifies each bug b by a triple (ℓ, b^-, b^+): line ℓ in commit b^- is the faulty statement, which is fixed by the (later) commit b^+. As we mentioned above, Islam et al. [57]'s dataset was collected by manual analysis, and thus some of the bugs are not (no longer) reproducible, are duplicate, or are otherwise outside ANNOTEST's scope. For our evaluation, we selected only *unique reproducible crashing bugs*: *i*) "crashing" means that the fault triggers a runtime program failure, which we use as oracle;[18] the crashing location c may be different from the bug location ℓ; *ii*) "reproducible" means that we could manually run the program to trigger the failure; *iii*) "unique" means that we merged bugs that are indistinguishable by a crashing oracle (for example, they crash at the same program point, or they fail the same assertion) or that refer to the very same triple in Islam et al. [57]'s dataset.

Out of all 213 bugs in Islam et al. [57] for the 19 selected projects, we identified 81 unique reproducible crashing bugs. For each such bug $b = (\ell, c, b^-, b^+)$ we annotated the project's commit b^- starting from the function (or method) `f` where location ℓ is, and continuing with the other functions that depend on `f`. We stopped annotating as soon as the annotations where sufficient to exercise function `f` (including, in particular, reaching ℓ and/or crash location c). Then, we ran ANNOTEST to generate tests for `f` and any other functions that we annotated. We count bug b as *reproduced* if some of the generated tests fails at crashing location c, and doesn't fail if run on the patched version b^+.

To address RQ3, we measured the annotations we wrote for RQ1's and RQ2's experiments; and we compared the size (in lines of code) of these manually-written annotations to the Hypothesis code generated automatically by ANNOTEST from the annotations.

To address RQ4, we compared ANNOTEST to Pynguin and Deal. As we discussed in Chapter 2, Pynguin [78] is a state-of-the-art unit-test generator for Python that leverages type hints to improve

[18] While Islam et al. [57] classify some bugs as "crashing", we also included bugs in other categories provided they can eventually generate a crash.

its effectiveness (although it also works without type hints); Deal [29] is a Python library for Design by Contract, supporting annotations such as preconditions, as well as test-case generation and static analysis based on them. For the comparison with Pynguin, we annotated the functions in Vision's module `mnist` (see Section 3.5) using AN similarly to what done for RQ1, writing 21 regular annotations and 1 generator for 23 functions under test; then, we compared Pynguin's generated tests to ANNOTEST's. For the comparison with Deal, we took all functions in our running examples Listings 3.1–3.5 and added preconditions in Deal's syntax that express the same input constraints as our annotations in AN's syntax; then, we compared Deal's generated tests to ANNOTEST's.

To address RQ5, we annotated the functions in Vision's modules `backbone_utils`, `image`, and `_video_opt` (see Section 3.5) using AN similarly to what done for RQ1. Since the goal is comparing to manually written tests, we ignored the tests when writing AN annotations, and only considered examples of function usages in the library implementation or comments. Using tool *Coverage.py*[19] we measured the branch coverage achieved on each module by: *i*) the manually-written unit tests in Vision's test suite; *ii*) the tests generated by ANNOTEST from the annotations. We used branch coverage but note that, on these subjects, this metric correlates very strongly (Pearson correlation coefficient: 0.94) with statement coverage; thus, using either coverage metric would lead to the same findings.

Annotation effort. As we mentioned in Section 3.3.2, gaining an accurate understanding of a program's behavior is necessary regardless of the approach one follows to build tests. In our experiments, we found that finding plausible ranges for a function's inputs requires only modest effort in the majority of cases. This is in accordance with the so-called *locality principle* [31], which implies that a significant part of a program's behavior often can be understood by observing only a small number of program inputs [33]. Regardless of whether one is targeting a program that is easy or hard to test, ANNOTEST can support the tester's job by providing a means of expressing the input constraints, of exercising them with automatic test generation.

3.7 Experimental Results

This section presents the experimental results and addresses our research questions outlined in Section 3.4.

3.7.1 RQ1: Precision

Table 3.2 shows the results of applying ANNOTEST to the latest commits[20] of projects ADV and GANS. With the goal of annotating the projects as thoroughly as possible, we ended up writing some AN annotations for 42% of their 249 functions. Most of the functions that we left without annotations do not need any special constraints to be tested—usually because they either are simple utility functions that are only called in specific ways by the rest of the project or have no arguments. There are a few additional cases of functions that are not used anywhere in the project and whose intended usage we could not figure out in any other way; in these cases, we did not annotate them (and excluded them from testing). With these annotations, ANNOTEST reported 56 crashes, 50 of which we confirmed as genuine unique crashing bugs; this corresponds to a precision of 89%.

[19] Coverage.py v. 6.5.0: `https://github.com/nedbat/coveragepy`

[20] The projects are however no longer maintained; therefore, we did not submit any of the found bugs to the projects' repositories.

PROJECT	LOC	FUNCTIONS	ANNOTATIONS			BUGS		
			#A	%F	%G	TRUE	SPURIOUS	PRECISION
ADV	1 421	100	1.58	49%	7%	33	5	87%
GANS	2 496	149	1.15	37%	6%	17	1	94%
overall	3 917	249	1.33	42%	7%	50	6	89%

Table 3.2. Two projects fully annotated with ANNOTEST and the found bugs. Each row shows data about a PROJECT (identified by an acronym; see Table 3.3 for the URL of their GitHub repositories): its size in lines of code LOC and number of FUNCTIONS (including methods); the average (per function) number #A of annotations we added to the project, the percentage %F of functions with at least one annotation, and the percentage %G of annotations that use custom generators; and the number of unique crashing BUGS found by generating tests based on the templates—split into confirmed TRUE bugs, SPURIOUS bugs (triggered by invalid inputs), and the corresponding PRECISION = TRUE/(TRUE + SPURIOUS).

As previously reported [119], bugs due to project dependency conflicts are quite common in NN programs. An interesting example is a crash that occurs in ADV when it accesses attribute `W`{12} in Keras's class `Dense`.{13} This attribute was renamed to `kernel`{14} in Keras version 2.0. Since ADV explicitly supports this major version of Keras, this crash is a true positive. Another confirmed bug we found was due to a function in ADV still using tuple parameter unpacking{15}—a Python 2 feature removed in Python 3. The ADV project developers probably forgot to update this one instance consistently with how they updated the rest of the project,{16} which is indeed designed to work with Python 3.

A tricky example of false positive occurred in project GANS's function `create_celeba_channel_last`,{17} which creates an HDF5[21] file for the CelebA dataset [76]. One of the tests generated by ANNOTEST crashes{18} as it is unable to create a file. However, the failure does not happen if we run the function manually using the very same inputs; thus, the testing environment is responsible for the spurious failure.

These experiments suggest that ANNOTEST can be quite effective to pin down bugs, problems, and inconsistencies in NN programs, thus helping systematically improve their quality.

Applied to two fully-annotated open-source NN programs,
ANNOTEST *generated tests revealing 50 bugs with 89% precision.*

3.7.2 RQ2: Recall

Table 3.3 shows the results of applying ANNOTEST to detect 81 unique reproducible crashing bugs in 19 projects surveyed by Islam et al. [57] and selected as explained in Section 3.5. Using the annotations we provided, ANNOTEST reproduced 63 of these bugs without generating any spurious failing tests. This corresponds to a 100% precision and 78% recall relative to the unique reproducible known bugs from Islam et al. [57]. With the same annotations, ANNOTEST also revealed another 31 failures that we confirmed as additional crashing bugs in the same projects.[22]

While ANNOTEST was quite effective at reproducing the known bugs in these projects, it's interesting to discuss the issues that prevented it from achieving 100% recall. We identified several scenarios: *i*) masking; *ii*) scripting code; *iii*) nested functions; *iv*) lazy features; *v*) and inaccessible code.

[21] HDF5 (Hierarchical Data Format 5) for Python: `https://www.h5py.org`
[22] Islam et al. [57]'s survey is not meant to be an exhaustive catalog of all bugs in these projects.

PROJECT	LOC	FUNCTIONS		REV	ANNOTATIONS			BUGS					
		TOTAL	TESTED		#A	%F	%G	KNOWN	REP	OTHER	SPURIOUS	PRECISION	RECALL
K NAAS[19]	140	7	0	2	–	0%	0%	2	2	1	0	100%	100%
K ADV[20]	1 421	100	4	2	1.5	4%	0%	8	6	3	0	100%	75%
K DN[21]	82	5	2	1	14.0	40%	0%	2	2	2	0	100%	100%
K DCF[22]	748	35	1	1	4.0	3%	0%	1	0	0	0	–	0%
K KIS[23]	2 050	92	2	1	1.5	2%	0%	6	5	0	0	100%	83%
K FRCNN[24]	1 643	55	3	1	1.7	5%	0%	6	3	0	0	100%	50%
K CONV[25]	350	20	0	1	–	0%	–	1	0	0	0	–	0%
K mCRNN[26]	225	1	0	1	–	0%	0%	1	1	5	0	100%	100%
K IR[27]	306	38	0	1	–	0%	–	2	0	0	0	–	0%
K RE[28]	966	25	1	1	15.0	4%	0%	1	1	5	0	100%	100%
K CAR[29]	353	21	1	1	7.0	5%	0%	1	1	1	0	100%	100%
K GANS[30]	2 496	149	2	1	12.5	1%	4%	6	4	5	0	100%	67%
K KAX[31]	227	15	0	1	–	0%	–	1	0	0	0	–	0%
K VSA[32]	630	38	2	1	6.0	5%	0%	2	2	4	0	100%	100%
K UN[33]	440	28	3	2	3.3	11%	30%	6	2	1	0	100%	33%
K LSTM[34]	477	27	0	1	–	0%	–	1	0	0	0	–	0%
F TC[35]	285	7	0	2	–	0%	0%	9	9	2	0	100%	100%
F TPS[36]	286	2	2	1	4.0	100%	87%	24	24	0	0	100%	100%
T DAF[37]	1 094	70	1	1	9.0	1%	67%	1	1	2	0	100%	100%
overall	14 219	735	24	23	6.0	3%	12%	81	63	31	0	100%	78%

Table 3.3. Bugs from Islam et al. [57] that ANNOTEST could reproduce. Each row shows data about a PROJECT (identified by an acronym and the URL of its GitHub repository): its DNN framework (Keras, TensorFlow, Torch), its size in lines of code LOC and the number of TOTAL and TESTED functions (including methods); the number of its different REVisions that we analyzed, the average (per tested function) number #A of annotations we added, the percentage %F of functions with at least one annotation, and the percentage %G of annotations that use custom generators; and the number of crashing BUGS found by generating tests based on the templates—the number of *reproducible* KNOWN bugs reported by Islam et al. [57], how many of these the tests REPproduced, how many OTHER confirmed true bugs and SPURIOUS bugs (triggered by invalid inputs) the tests also reported in the same experiments, and the corresponding PRECISION = (REP + OTHERS)/(REP + OTHERS + SPURIOUS) and RECALL = REP/KNOWN.

Masking occurs when an earlier crash prevents program execution from reaching the location of another bug b'. Masking is usually not a problem when the earlier crash is determined by a known bug b: in this case, we can just run tests on the project commit b^+ where b has been fixed, so that execution can reach the other bug b'. However, if a bug b' is masked by an unknown bug (column OTHER in Table 3.3), and we don't know how to fix the unknown bug to allow the program to continue, b' is effectively unreachable. We could not reproduce 4 known bugs because of masking. One of them occurrs[38] in project GANS, and is masked by an unexpected crash[39] occurring in the same function `Discriminator`. In project UN, some missing statements make it impossible to distinguish three known bugs,[40],[41],[42] since they all crash the same test. Therefore, we consider 1 of them reproduced and 2 not reproduced due to masking. One of the tests produced for project GANS stopped before finding a known bug,[43] with a `SIGKILL` (triggered by memory-related issues).

ANNOTEST generates unit tests, which target specific functions in a program's source code. This excludes any code snippets in the "main" section of a Python file (under `if __name__=='__main__'`), which executes when the file is run as a script from the command line. Therefore, ANNOTEST could

not reproduce 6 bugs affecting this *scripting code*, such as one known bug in project CONV.{44} Another example is the only known bug{45} in project KAX, which occurs in a function that depends on command line arguments.

ANNOTEST can test *nested functions* only indirectly, that is when they are called by a top-level function as part of testing the latter. It does not support annotating nested functions and generating unit tests for them since they are not accessible outside their parent functions. We could not reproduce 3 known bugs because they affected nested functions. An example is in project FRCNN's function `rpn_loss_regr_fixed_num`,{46} which is defined inside top-level function `rpn_loss_regr`.

Functions using Python's `yield` statement are *lazy*, that is their evaluation is delayed. This means that they may not be executed by ANNOTEST's unit testing environment (or rather its Hypothesis backend's). We could not reproduce 1 known bug{47} in project KIS because it uses `yield` to build a lazy iterator.

As we remarked above, a bug's crashing location c may differ from the actual error location ℓ in commit b^-. If c is in a portion of the code that is *not accessible* to the testing environment, ANNOTEST cannot reproduce the bug even if it is reproducible in principle. This scenario occurred for 3 known bugs that ANNOTEST didn't reproduce. Two of them are in project UN{48},{49} and only crash in a module whose implementation is incomplete in that program revision. Another one{50} occurs in project IR: we tried to no avail to reproduce it at a different, accessible location.

Finally, we could not reproduce 1 bug{51} in project IR simply because we could not figure out suitable type constraints to properly exercise the corresponding function.

> ANNOTEST *generated tests revealing 63 known NN bugs in 19 NN programs, with a recall of 78%.*

3.7.3 RQ3: Amount of Annotations

For the ANNOTEST approach to be practical, it is important that it requires a reasonable amount of manual annotations. We leave to future work a detailed empirical evaluation of the time and expertise that is needed to write AN annotations. Here, we discuss quantitative measures of ANNOTEST's annotation overhead. We focus on RQ1's experiments (Section 3.7.1), which analyzed projects ADV and GANS in full, as they give a better idea of the effort needed to use ANNOTEST systematically on whole projects.[23]

Annotation Amount

The amount of annotations that we wrote was usually limited. In RQ1's experiments, we wrote 2 annotations[24] per project function on average (median); 80% of functions have 3 annotations or less. Annotations are mostly concise: 96% of them fit a single line, and only 10% (12) of all functions have annotations that span more than 5 lines (usually decorating functions with several complex arguments). Figure 3.2 pictures the distributions, overall functions, of number of annotations (left) and lines of code (LOC, middle) of annotations; since most annotations are a single line, these two distributions are nearly identical.

The average number of annotations per tested function is higher (6.0) in Table 3.3 since in each of those experiment we annotated a limited portion of a project focusing on a specific function that had a known bug; therefore, several of the annotations are duplicated or only slightly modified from

[23] The figures for RQ2's experiments are, however, generally similar.

[24] An annotation is any instance of the kinds presented in Section 3.3.1.

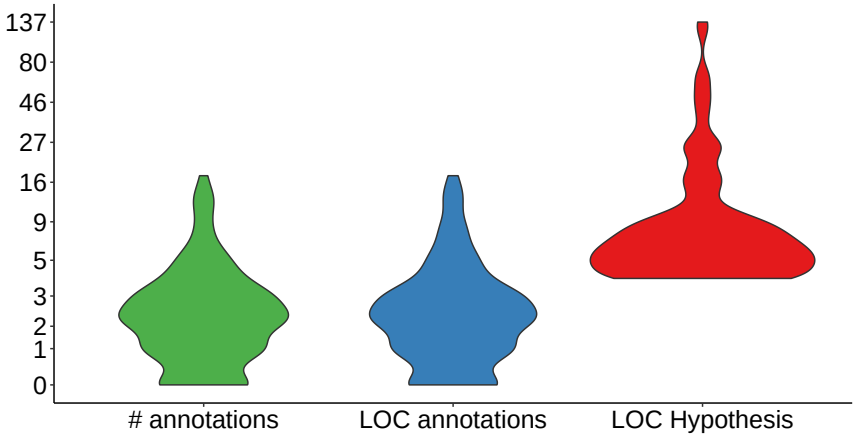


Figure 3.2. Distribution of the number of AN annotations, lines of code (LOC) of AN annotations, and LOC of generated Hypothesis templates for RQ1's experiments.

one experiment to the other. If we had fully annotated the projects, we would have likely amortized some of this annotation effort.

In terms of time, we spent, on average, 10–15 minutes to write the annotations of each function. This time includes inspecting the project's source code to become familiar with how it works. As pointed out in Section 3.6, this effort is amortized over various related functions, and is unevenly distributed, with a few "complex" functions taking considerably more time to understand than most "simple" functions. As mentioned in Section 3.3.2, we consider the overall effort comparable to the time to manually write unit tests for the same functions.

Another way of quantifying the effort-benefit trade-off is measuring the amount of annotations per detected bugs: this ratio is 6.6 = 330/50 for the fully-annotated projects in Table 3.2 and 1.5 = 145/94 for the experiments in Table 3.3. These are encouraging figures, if we think of the amount of manually-written tests that may have been necessary to discover the same bugs (see also Section 3.7.5).

The percentage of annotations using generators is higher (12%) for the projects in Table 3.3. More precisely, the two projects in Table 3.2 use 15 generators, 73% (11/15) of which generate NN models. Among the 16 generators built for the projects in Table 3.3, 31% (5/16) generate NN models, 37% (6/16) provide TensorFlow's tensor objects, and 25% (4/16) load datasets from disk. The one remaining generator function loads an image from hard disk, turns it into a NumPy array and passes it to a function. As we explained in Section 3.3.3, we built all generators by applying light refactoring operations to suitable portions of existing client code within the same project.

Hypothesis Overhead

Since ANNOTEST translates AN annotations to Hypothesis templates, we can quantify how concise AN is compared to directly encoding constraints in Hypothesis. The rightmost plot in Figure 3.2 pictures the distribution of LOC of generated Hypothesis code. Clearly, Hypothesis code is considerably more verbose than AN annotations: Hypothesis templates are 5.5 (median overhead) times longer—11.6 times longer in terms of mean overhead—than the AN annotations they encode, which points to the benefits of using AN's concise language.

In our experiments, ANNOTEST *used 2 annotations per function on average; 96% of all annotations fit a single line.*

3.7.4 RQ4: Comparison to Generic Test-Case Generators

We designed ANNOTEST not as a general-purpose testing tool but as one specifically geared towards NN programs. Therefore, we expect ANNOTEST to outperform generic test-case generators for Python when generating tests for these programs.

As we discussed in Section 3.5, we ran **Pynguin** on module `mnist` in project Vision; the module includes type hints annotations, which Pynguin uses to improve the accuracy of its generated tests. Pynguin[25] generated 19 tests, reporting 6 tests as passing (they terminate without errors), and 13 tests as failing (they throw an exception). By manual inspection, we determined that: *i*) 2 of the 6 passing tests and 10 of the 13 failing tests are actually *invalid*, since they call functions with input values that are not valid according to the functions' docstring, type hints,[26] or other available documentation; *ii*) the other 3 failing tests should be classified as *passing*, since throwing an exception is the functions' expected behavior in those cases. In all, 63% $((2+10)/19)$ of the tests generated by Pynguin are invalid, and 79% $((3+2+10)/19)$ are misclassified. We cannot expect Pynguin to perform better, since it simply lacks the information to precisely characterize valid inputs; in contrast, leveraging the AN annotations' information, ANNOTEST generated 11 tests for module `mnist`: all of them are valid and passing.[27]

Deal's expressive annotation language is capable of concisely encoding most of the AN annotations as preconditions (`@deal.pre`). Then, Deal's test-case generation engine draws inputs randomly and uses preconditions to filter them; therefore, the stronger a precondition is, the more it will struggle to find any valid inputs. In all our examples (Listings 3.1–3.5), Deal could not generate a single valid input that satisfies all constraints. Even after removing some of the most complex constraints (for example, the first one in Listing 3.2), Deal's built-in test-case generator couldn't generate valid inputs. Here too, we cannot expect Deal to perform better, since, unlike ANNOTEST, its test-case generation process is not built around the kinds of complex constraints that arise in NN programs.[28]

> ANNOTEST *outperforms other test-case generation techniques that are not designed specifically for NN programs.*

These results are another manifestation of the trade-off between specification accuracy and test effectiveness: precise tests require precise knowledge of the expected program constraints (and behavior), regardless of whether this knowledge is formalized as annotations, as executable code, or is applied directly by the programmer.

3.7.5 RQ5: Code Coverage

Table 3.4 compares the manually-written tests in three of project Vision's modules (see Section 3.5) to those generated by ANNOTEST after annotating the functions in these modules.

Module `_video_opt` is scarcely tested in Vision: there are no unit tests for this module (column UNIT in Table 3.4), but tests in other modules still indirectly exercise 16% of its branches (column INDIRECT). In contrast, ANNOTEST reaches a 76% coverage after annotating 8 functions in this module. Vision's unit tests for module `image` achieve a 79% coverage; ANNOTEST reaches a higher 84% coverage. Finally, Vision's unit tests for module `backbone_utils` achieve a 82% coverage, the same

[25] We report experiments that used Pynguin's default configuration; however, using other generation strategies did not significantly change the outcome.

[26] Pynguin may violate type hints whose format it does not support.

[27] While experimenting with testing the Vision project using ANNOTEST, we found a bug in a module that Pynguin cannot test. We submitted a fix as a pull request{52} that was promptly accepted. Interestingly, the affected function{53} already included a developer-written parameterized test,{54} which nonetheless did not detect "our" bug; this further demonstrates ANNOTEST's practical effectiveness.

[28] In addition, Deal focuses on using annotations for static analysis.

	PROJECT TEST SUITE					ANNOTEST			
	COVERAGE		#FUNCTIONS	TEST SIZE		#TESTS	COVERAGE	ANNOTATION SIZE	
MODULE	UNIT	INDIRECT		LOC	CHARS	ANNOTATED		LOC	CHARS
`_video_opt`	0%	16%	0	0	0	8	76%	25	1 566
`image`	79%	0%	15	359	13 381	11	84%	40	1 858
`backbone_utils`	82%	14%	9	238	8 478	3	82%	23	1 044
overall	50%	5%	24	597	21 859	22	80%	88	4 468

Table 3.4. A comparison (part of) Vision's programmer-written test suite and ANNOTEST's generated tests in terms of coverage. For each MODULE, the table reports the branch COVERAGE of the programmer-written PROJECT TEST SUITE on the module (split between UNIT tests directly targeting the module's functions, and coverage achieved INDIRECTly by other modules' tests calling the module); the number of unit test FUNCTIONS directly exercising the module; and the size of these tests in lines of code LOC and number of characters CHARS; the number of functions we ANNOTATED; the COVERAGE achieved by ANNOTEST on these functions; and the size of these annotations in lines of code LOC and number of characters CHARS.

as ANNOTEST. The whole test suite in Vision actually further exercises module `backbone_utils`, as tests in other modules indirectly add an additional 14% of coverage. Overall, ANNOTEST-generated tests achieve a high coverage—comparable to or often higher than that of the programmer-written test suite.[29]

In order to achieve this coverage, what is the amount of code (manual tests) or annotations (ANNOTEST) that is required? Vision's unit tests for Table 3.4's three modules consist of 24 tests, spanning 597 lines of code or 21859 characters; ANNOTEST's annotations are only needed for 22 functions, and span 88 lines or 4468 characters. This confirms that ANNOTEST's annotations are concise—considerably more concise than unit tests achieving a lower coverage. Naturally, the sheer size of a piece of code is an imperfect measure of the effort needed to produce it; however, AN annotations encode essentially the same information as parametric tests, and their succinctness is an advantage.

100% coverage? Neither ANNOTEST nor the programmer-written test suite managed to cover 100% of the branches in the three modules. In a few cases, increasing the coverage would be possible by simply writing more unit tests or more general annotations. For instance, none of the manual tests for module `backbone_utils` instantiates class `BackboneWithFPN` by passing argument `None` to its constructor's parameter `extra_blocks`; the corresponding branch{57} is thus never covered (but it is by ANNOTEST). Conversely, ANNOTEST does not test function `_read_video_from_memory`{58} in module `_video_opt` because we could not find meaningful examples of its usage.[30] In other cases, however, achieving a 100% coverage is impractical due to constraints in the test execution environment. For instance, a branch{59} in function `decode_jpeg` of module `image` requires running the module on a machine with a GPU supporting the CUDA API.[31] There is actually a manual test{60} covering this branch, but it was not activated in our experiments since we did not run them with CUDA. We found a few other examples of this scenario{61},{62} where increasing the test coverage

[29] While testing module `image` in these experiments, ANNOTEST detected a failure in function `decode_jpeg`{55} (which is already thoroughly tested in the project's test suite). Reporting this failure{56} to the project maintainers prompted them to modify the function's documentation so as to more accurately reflect its intended, implemented behavior.

[30] As discussed in Section 3.6, we did not consider the manually-written tests when writing AN annotations, so that the comparison in terms of coverage is fair and meaningful.

[31] CUDA: `https://developer.nvidia.com/cuda-zone`

requires specific hardware or system libraries.

Bug density. Users of ANNOTEST write annotations to then generate unit tests automatically. In RQ1's experiments, ANNOTEST generated 5649 (valid) tests overall; only 1% of them fail and expose a bug. Thus, bugs in NN programs are *rare* [106]. This suggests that directly writing tests that selectively expose these bugs may be challenging even for programmers knowledgeable of the program under test. The same knowledge is sufficient to write AN annotations and generate tests from them.

> ANNOTEST *achieves high code coverage,*
> *comparable to that of manually-written test cases.*

3.8 Threats to Validity

Identifying valid test inputs, and distinguishing between spurious and authentic bugs, is crucial to ensure *construct validity* (i.e., the experimental measures are adequate). Unfortunately, a reliable and complete ground truth is not available: the documentation of NN programs is often incomplete (when it exists), so we had to manually discover the intended behavior of NN programs from examples, manual code analysis, and background knowledge. Our reference—Islam et al. [57]'s survey—was also compiled by purely manual analysis; therefore, it does not aim at completeness, and includes bugs that are not reproducible (see Section 3.6). These limitations imply that we cannot make claims of completeness ("we found all bugs"); nevertheless, we still have a good confidence in the correctness of our results ("we found real bugs"): since we focused on bugs detected by crashing oracles, most bugs we found with ANNOTEST are clear violations of the program's requirements.

Since ANNOTEST uses manually-written annotations, quantifying the annotation effort is needed for *internal validity* (i.e., the experimental results are suitable to support the findings). We mostly reported simple measures (number of annotations, number of functions that require annotations, etc.) which are unambiguous. In contrast, we do not make any strong claims about the time and relative effort needed by programmers to annotate: these heavily depend on a programmer's knowledge of the NN program and of the domain; precisely assessing them would require controlled experiments and user studies, which are outside our study's scope. However, we remark that expressing AN annotations requires a knowledge of the program under test of the same kind that is needed to write effective unit tests.

Picking experimental subjects from Islam et al. [57]'s extensive survey of real-world NN bugs helps *external validity* (i.e., the findings generalize). As we discussed in Section 3.5, we excluded some projects for practical reasons (e.g., no longer available or incomplete) and we focused on those using the Keras NN framework. While this focus does not seem especially restrictive (the majority of projects in the survey uses Keras, and we also analyzed projects using other frameworks), applying ANNOTEST to very different kinds of NN programs may require different kinds of annotations or other changes in the approach. The AN annotation language is extensible with generators (Section 3.3.1), which can further help generalizability. Furthermore, in addition to Islam et al. [57]'s subjects, we also extensively analyzed the *latest* versions of projects ADV and GANS (Section 3.7.1), so that our evaluation did not only include projects with known bugs.

3.9 Conclusions and Future Work

The chapter presented the ANNOTEST approach to generate inputs that test NN programs written in Python. ANNOTEST relies on code annotations that precisely and succinctly describe the range of

valid inputs for the functions under test. Using this information, aNNoTest can generate tests that avoid spurious failures, and thus have a good chance of exposing actual bugs. In an experimental evaluation targeting 19 open-source NN programs, aNNoTest was able to reveal 94 bugs (including 63 previously known ones) with an overhead of 6 annotations per tested function on average.

Future work

A natural continuation of the work on aNNoTest is extending aN to support more kinds of constraints. As discussed in Section 3.7.3, most of the generator functions we wrote for our experiments generate complex NN model objects such as tensors; being able to specify such objects concisely would further increase the applicability and convenience of using aNNoTest.

This chapter's contributions address the test-input generation problem, which is largely independent of the test-oracle problem (see Section 3.3.5 and Section 2.2.2). In future work, we may extend aNNoTest to add support for other kinds of oracles. Since aNNoTest is based on annotations—a form of lightweight formal specification—adding *postconditions* would be a natural way to do so. Unlike the annotations currently supported by aNNoTest, which act as constraints on the pre-state and hence require a matching generation mechanism, postconditions are evaluated on a test's post-state, and hence can simply be evaluated to determine whether the test is passing or failing.

Regression oracles are another kind of oracles that are commonly supported by test generation tools such as Pynguin [78]; aNNoTest could add support for a similar mechanism to generate *regression tests*, whose assertion capture the post-state of the program under test, and can be re-run on future versions of the program to determine whether its expected behavior has changed. Given aNNoTest's focus, it could target regression oracles that capture NN-specific properties [34, 85, 137].

4

The ANNoTest Tool and Dataset

We proposed ANNoTest in Chapter 3, our novel test generation technique tailored for neural network (NN) programs, highlighting its capabilities at finding NN bugs. We demonstrated that using ANNoTest's domain-specific annotation language AN, instead of directly writing Hypothesis templates and strategies, has advantages in terms of conciseness and better alignment with the characteristics of NN programs.

This chapter discusses more details about the tool, also called ANNoTest, that implements the testing technique, including some elements of its design, and a few practical details from a user's perspective. Furthermore, we also present a curated dataset of NN bugs collected and prepared during the research detailed in Chapter 3, and explain how this artifact can be a useful resource for further studies in the field.

Structure of the Chapter

The current chapter is organized as follows:

Section 4.1 provides the motivation behind developing the ANNoTest tool.

Section 4.2 overviews how to use ANNoTest, using a simple example.

Section 4.3 highlights details of ANNoTest's design and implementation.

Section 4.4 introduces a curated dataset of NN bugs we sourced from [57]'s survey.

Section 4.5 concludes this chapter.

4.1 Introduction

As neural network (NN) programs are increasingly deployed in safety-critical systems, developing techniques to effectively test them becomes paramount. Most of the research on NN testing focused on exercising model-level properties like robustness and training performance [70, 104]. However, additional challenges come from the way in which NN programs are usually written.

NN programs are often developed by domain experts, written in dynamic programming languages such as Python, using libraries and frameworks such as Keras or TensorFlow. There is evidence that such NN programs are prone to suffer from various defects [57, 142], including low-level bugs—such as type errors and other kinds of runtime failures—that derive from the dynamically typed nature of Python but would be caught by the compiler in a statically-typed language. Python's dynamic type system is also a challenge for automated test-case generation tools, which tend to be

more novel and less developed than for statically-typed languages like Java [39, 66, 89]. The few Python test-generation tools that exist—such as Pynguin [78]—are general-purpose, and hence may not be effective to generate tests for NN programs.

As we demonstrate in Section 4.2, using a general-purpose test-case generation tool on a NN program is likely to generate many invalid inputs, which trigger spurious crashes without finding any actual bug. General-purpose tools implement several approaches to mitigate the challenges of targeting a dynamically typed language like Python—for instance, leveraging type hints [4]. However, NN programs manipulate complex, precisely-constrained objects such as tensors and vectors, which compounds the challenges of dealing with a dynamically typed language and makes general-purpose annotations such as type hints insufficient for precise test-input generation.

To address these challenges, we present ANNOTEST: a unit-test generation tool for NN programs written in Python. ANNOTEST [99] relies on a simple, domain-specific annotation language called AN, which one can use to precisely and concisely express the complex input constraints of a NN program's functions to be tested. Given some annotations, ANNOTEST automatically generates unit tests with high precision. The ANNOTEST tool is available as open source. The simplest way to use it is by installing it from PyPI[1] with `pip install annotest`. ANNOTEST's main repository[2] includes the tool's source code and instructions to use it. A short demo of ANNOTEST is available on Youtube.[3]

4.2 Using ANNOTEST

In this section, we briefly demonstrate how to use ANNOTEST to generate unit tests that can expose bugs in neural-network programs written in Python. We also discuss how ANNOTEST is more effective than other, general-purpose test-generation tools for Python when testing NN programs with their particular constraints on inputs. To this end, we discuss using ANNOTEST in comparison with Pynguin [78] and Deal [29]—the only general-purpose state-of-the-art automated test generation tools for Python available at the time of writing. As we will demonstrate in the rest of this chapter, ANNOTEST is designed as *complementary* to these two tools: while its annotations are applicable, in principle, to any Python program, ANNOTEST is primarily a specialized tool for NN programs; Pynguin and Deal are general-purpose tools suitable for all sorts of Python programs.

Consider function `model_discriminator` from project ADV (Keras Adversarial Models) [1] whose implementation is in Listing 4.1. When Python 3's interpreter evaluates the expression on line 145, the execution *crashes*, because `hidden_dim / 2` returns a **`float`** but the constructor `Dense` only works correctly if its first argument is an **`int`**.[4] This bug was among those surveyed by Islam et al. [57], and was eventually fixed with an explicit **`int`** conversion in a later revision of the ADV project. We selected this example because it is simple, yet realistic; using it makes for a clearer presentation, but we stress that the challenges that we highlight become much more problematic as soon as one target larger and more complex examples (such as those that we discussed in Section 3.5).

On the face of it, generating valid tests that exercise function `model_discriminator`, and trigger the bug, should be straightforward: the function's implementation is short, consists of straight-line code (no branching), and only takes four arguments. Furthermore, *every valid* input would trigger the bug, and valid input values for three out of four arguments are provided as defaults. And yet, automatically generating several valid inputs for this function turns out to be surprisingly hard for general-purpose testing tools—as we now discuss in detail.

[1] `https://pypi.org/project/annotest/`

[2] `https://github.com/atom-sw/annotest`

[3] A demo of ANNOTEST: `https://youtu.be/3Y1sraVajIA`

[4] The function has a similar bug on line 147; we focus on line 145's bug, which masks the bug on line 147.

4.2.1 General-Purpose Testing Tools

Pynguin

Consider Pynguin [78], a general-purpose automated test generation tool for Python. Pynguin's test-generation strategy is based on a genetic algorithm that tries to maximize branch coverage. For a simple, straight-line function like `model_discriminator`, Pynguin only generates[5] one test input consisting of string `"!b)p"` as actual value for argument `input_shape`. This input is *invalid*, since `input_shape` must be a *shape*—basically, a tuple of nonnegative integers that denote the dimensions of a multi-dimensional array.

It should be no surprise that Pynguin is ineffective on this example: Python does not require function arguments to be annotated with their intended types, and this crucial piece of information is thus not available to the test-case generation tool. Unfortunately, even if `model_discriminator` were annotated using Python's *type hints*—which Pynguin partially supports—these are not expressive enough to precisely encode the range of `model_discriminator`'s valid inputs. The best we can do is annotating `input_shape` with type `Tuple[int, int]`, `hidden_dim` with type `int`, `reg` with type `Callable`, and `output_activation` with type `Literal`. These constraints are both too loose to identify valid inputs only, and not fully compatible with Pynguin's generation algorithm. As a result, Pynguin generates two test inputs, both invalid: in one, `input_shape` is the lone integer `-1262`; in another one, it is the tuple `(345, True)`; neither of them is a valid shape.

Deal

Encoding arbitrarily complex constraints is possible using a tool like Deal [29], which offers an expressive language to encode function *preconditions* (as well as other design-by-contract annotations).

Using Deal's annotation language, we can precisely express `model_discriminator`'s input constraints, which we can elicit from examples of client code in other parts of project ADV (as well as from the default values for some of the arguments). In particular, `input_shape` should be a two-element tuple of integers in the range 1–28,[6] and `hidden_dim` should be a positive integer, typically in the range 1024–2048. For simplicity, we can tentatively ignore the more complex constraints on arguments `reg` and `output_activation`, and simply use the default values of those two arguments.

Even though we can precisely express the constraints on `input_shape` and `hidden_dim`, Deal still fails to generate any valid inputs for `model_discriminator`. This is because Deal's input generation algorithm produces *random* inputs, and then uses the constraints/preconditions to filter a posteriori the random inputs. Thus, it's exceedingly unlikely that a random value (among all possible Python types) happens to satisfy detailed constraints such as those required by `model_discriminator`.

In hindsight, it is unsurprising that Pynguin and Deal—two state-of-the-art *general-purpose* test-case generation tools for Python—are ineffective on this deceptively simple example. NN programs often consist of structurally simple, usually short, functions with complex constraints on their numerous input arguments [58]. In order to generate valid tests for these programs, we need a *specialized* approach which supports both *i*) concisely *expressing* the complex input constraints; and *ii*) using those constraints to *drive the generation* of possible valid inputs. This is what aNNoTest is designed for.

[5] We ran Pynguin 0.33.0 (the latest version at the time of writing) with its default configuration: DynaMOSA for test-suite generation and MUTATION_ANALYSIS for assertion generation.

[6] Precisely, the tuple's components would be valid even if they were larger than 28, but the 1–28 range is sufficient to generate a wide variety of valid inputs that reflect typical usage.

```
138 def model_discriminator(input_shape, hidden_dim=1024,
139                          reg=lambda: l1l2(1e-5, 1e-5),
140                          output_activation="sigmoid"):
141   return Sequential([
142     Flatten(name="discriminator_flatten", input_shape=input_shape),
143     Dense(hidden_dim, name="discriminator_h1", W_regularizer=reg()),
144     LeakyReLU(0.2),
145     Dense(hidden_dim / 2, name="discriminator_h2", W_regularizer=reg()), # bug
146     LeakyReLU(0.2),
147     Dense(hidden_dim / 4, name="discriminator_h3", W_regularizer=reg()),
148     LeakyReLU(0.2),
149     Dense(1, name="discriminator_y", W_regularizer=reg()),
150     Activation(output_activation)], name="discriminator")
```

Listing 4.1. Function `model_discriminator` from project ADV (Keras Adversarial Models).

```
151 @arg(input_shape): np_shapes(min_dims=2, max_dims=2, min=1, max=28)
152 @arg(hidden_dim): ints(min=1024, max=2048)
```

Listing 4.2. Annotations for function `model_discriminator` in Listing 4.1. The annotations' syntax is slightly simplified for readability.

4.2.2 ANNOTEST

ANNOTEST is a unit-test generation tool specifically designed to be effective on Python implementations of NN programs. To this end, it offers AN: a simple annotation language suitable to concisely express the typical constraints on function inputs in such programs.

Annotations

Listing 4.2 shows some AN annotations for function `model_discriminator`, which precisely characterize valid input values for arguments `input_shape` and `hidden_dim`, as we discussed them above. Users of ANNOTEST can decide how much annotation effort to spend, depending on which functions they want to focus on testing, and on how thorough the testing should be.

Concretely, ANNOTEST's distribution includes a module `an_language.py` with concrete syntax for the AN annotation language. To annotate a function, users of ANNOTEST import this module, and then use the imported decorators just before each function to be annotated, directly in the source code.

In this example, we did not annotate two arguments, and just relied on their default values. If we wanted a more extensive test generation process, we could provide sets of possible suitable functions (argument `reg`) and library function names (argument `output_activation`), and constrain ANNOTEST to pick input values from these sets.

Test generation

Writing annotations is the only manual part of using ANNOTEST. After adding function Listing 4.2's annotations to the source code, we call ANNOTEST with `annotest <program_root>`. The tool scans through the whole program and generates tests for all annotated functions.

When an argument is left without AN annotations, ANNOTEST uses the argument's default value. Thus, ANNOTEST can only test a function if all its arguments have some AN annotations or a default

$$
\begin{aligned}
\mathit{An} \rightarrow\ & \texttt{@arg}(\mathit{var})\texttt{:}\ \mathit{TypeConstr} \mid \texttt{@require}(\mathit{BooleanExpr}) \\
\mathit{TypeConstr} \rightarrow\ & \texttt{froms}(\texttt{list}) \mid \texttt{bools}() \mid \texttt{ints}(\texttt{min=-Inf}, \texttt{max=+Inf}) \\
& \mid \texttt{floats}\left(\begin{array}{l}\texttt{min=-Inf}, \texttt{max=+Inf}, \texttt{exclude\_min=False}, \\ \texttt{exclude\_max=False}, \texttt{exclude\_NaN=True}, \\ \texttt{exclude\_Inf=True}\end{array}\right) \\
& \mid \texttt{tuples}(\mathit{TypeConstr}^{*}) \\
& \mid \texttt{np\_shapes}(\texttt{min\_dims=1}, \texttt{max\_dims=1}, \texttt{min=1}, \texttt{max=1}) \\
& \mid \texttt{int\_lists}(\texttt{min=-Inf}, \texttt{max=+Inf}, \texttt{min\_len=0}, \texttt{max\_len=10}) \\
& \mid \texttt{np\_arrays}(\texttt{np\_type}, \texttt{shape=}\mathit{TypeConstr}) \\
& \mid \texttt{dicts}(\texttt{keys=}\mathit{TypeConstr}, \\
& \qquad \texttt{values=}\mathit{TypeConstr}, \texttt{min\_size=0}, \texttt{max\_size=+Inf}) \\
& \mid \texttt{anys}(\mathit{TypeConstr}^{+}) \mid \texttt{objs}(\texttt{function})
\end{aligned}
$$

Figure 4.1. The main annotations supported by ANNOTEST.

value. In the running example, arguments `reg` and `output_activation` are not annotated; thus, ANNOTEST always sets `reg` to `l1l2(1e-05, 1e-05)` and `output_activation` to `"sigmoid"` (see Listing 4.1).

ANNOTEST generates Hypothesis test templates [80] using the format recognized by Python's `unittest` testing framework, which can be used to actually expand and run the tests. In our running example, ANNOTEST generates test templates nearly instantaneously; running them with `unittest` takes 0.3 seconds, and produces 14 distinct valid test inputs (one of which triggers the bug at Listing 4.1's line 145).

4.3 Design and Implementation

ANNOTEST inputs an annotated NN program and outputs Hypothesis test templates that encode all the information to generate unit tests that satisfy the annotated constraints. This section first describes the main features of the AN annotation language, and then how ANNOTEST aggregates the annotation information and uses it to generate tests.

4.3.1 The AN Annotation Language

Figure 4.1 shows the main kinds of annotations supported by ANNOTEST's AN annotation language. The largest class of constraints are *type constraints*: `@arg`(*var*)`:` *TypeConstr* constrains argument *var* to values of a specific type and range. These can be subsets of booleans, integers, and floating points (**`bools`**, **`ints`**, **`floats`**), as well as compound types such as tuples and dictionaries (**`tuples`**, **`dicts`**). Type constraints **`np_shapes`**, **`int_lists`**, and **`np_arrays`** specify lists and tuples of numeric values (including those supported by the NumPy library), which feature frequently in NN programs. Finally, **`froms`** supplies a list of concrete values to sample from; **`anys`** is the union of multiple type constraints; and **`objs`** introduces custom *generator functions* to generate arbitrarily complex objects.

Preconditions are constraints that involve multiple arguments at once or need to be conditional.[7] In AN, `@require`(p) specifies a precondition p, where p is an arbitrary Python Boolean expression

[7] Even though ANNOTEST's preconditions are similar to Deal's, the bulk of NN annotations can be expressed using ANNOTEST's type constraints and other, simpler annotations that crucially support the efficient generation of input values.

involving the annotated function's arguments. The AN language includes a few other annotations (which we do not discuss here for brevity), such as to skip testing certain functions, to add a test timeout, and to use constructors in test code (Section 3.3.1).

4.3.2 Testable Functions

By default, ANNOTEST generates tests for all functions in a NN program about which it has sufficient information—either through user-provided annotations or through default values.

More precisely, an argument `a` of a function `f` is *testable* if at least one of the following conditions holds: *i*) `f` includes an AN annotation that constrains `a`; *ii*) `a` has a default value (it is an optional argument); *iii*) `a` is `self`, and the enclosing class `C` has a constructor that is testable. *iv*) `a` is a non-keyword (`*args`) or keyword (`**kwargs`) variable-number argument. Thus, if `a` is testable, it means that ANNOTEST knows how to generate at least one valid value for it: one that satisfies an annotation, a default, one that can be constructed, or that can simply be omitted. A function `f` is testable if all its arguments are testable.

4.3.3 Strategies

For each *testable* argument, ANNOTEST generates a suitable Hypothesis strategy: a custom object-generating function. Several of AN's type constraints match some of Hypothesis's built-in strategies. For instance, Hypothesis strategies `array_shapes` and `integers` are suitable to easily encode AN's constraints **`np_shapes`** and **`ints`** used in Listing 4.2's running example. ANNOTEST can also reuse the default values of arguments using Hypothesis's `just` strategy.

More complex type constraints do not have a one-to-one matching Hypothesis strategy. In these cases, ANNOTEST automatically combines available strategies or even generates new strategies. For example, this is the case of constraint **`int_lists`**, which ANNOTEST encodes into a generator function for integer lists with suitable characteristics.

Another interesting case are **`objs`**`(gen)` type annotations, where `gen` is a user-defined function that should be used to generate values. In this case, ANNOTEST embeds and adapts `gen`'s implementation into Hypothesis code, so that it can be used as a generation strategy for the corresponding argument.

4.3.4 Templates

For each testable function `f`, ANNOTEST combines the strategies for each of `f`'s arguments—built as discussed in the previous section—into a *test template*. For a function `f` that is an instance method of some class `C`, ANNOTEST generates a template that first instantiates object `o` of class `C` using `C`'s constructor and the strategies recursively assigned to the constructor's argument. Then, it calls `o.f(...)` passing the values obtained by the strategies of `f`'s other arguments. To encode **`@require`** annotations (preconditions), ANNOTEST uses Hypothesis's `assume` function.

4.3.5 Executing the Tests

The final output of a run of ANNOTEST consists of several Hypothesis test templates—one for each testable function in the analyzed NN program. ANNOTEST uses `unittest`'s format to encode the test templates. Thus, users invoke `unittest` to pass ANNOTEST's output to Hypothesis, which actively generates concrete input values using the strategies, runs the tested functions, and reports any test failure.

4.3.6 Implementation Limitations

ANNOTEST cannot test nested functions—functions that are defined within other functions. This limitation, which also applies to manually written tests, is simply due to the fact that a nested function is invisible outside its host function; hence, we cannot test the nested function unless we also test the host function.

ANNOTEST's implementation relies on the Python `ast` library to extract information from a project's source code; thus, ANNOTEST can only process code that is syntactically valid. If a module fails a syntax check, ANNOTEST skips it and provides a warning message.

4.4 Curated Dataset of NN Bugs

In Chapter 3, Table 3.3 presented findings about the bugs we reproduce using ANNOTEST as described in Sections 3.5 and 3.6. We started with 19 open-source NN projects surveyed by Islam et al. [57]; but, we managed to reproduce bugs in 14 projects as indicated by column REP in Table 3.3. We prepared a repository[8] including the source code of 62 bugs within these 14 open-source NN projects.[9]

We curated this repository to ensure that each bug is easily reproducible using ANNOTEST—or with any other Python source-code tool. In particular, each bug comes with an installation script that creates a virtual environment with all dependencies, our AN annotations, the bug-triggering tests generated by ANNOTEST, and scripts to rerun the tests or ANNOTEST on the program. Besides documenting several realistic examples of using ANNOTEST, this repository can support further work in this area, by providing a curated collection of real-world reproducible NN bugs.

4.5 Conclusions and Future Work

In this chapter, we described ANNOTEST: an automated test generation tool for NN programs. We provided the motivation behind ANNOTEST, and outlined how to use it. ANNOTEST is fundamentally oracle agnostic: it can use any user-provided oracle to identify faults. The experiments we conducted so far (Sections 3.4–3.6) mostly use implicit crash oracles, or library assertion failures; this is consistent with the focus on NN bugs, where low-level failures and crashes are widespread [58]. As future work, we may extend ANNOTEST with oracle-generation capabilities. Given its current design, a natural choice would be extending AN with syntax for *postconditions*, which can be used as oracles of correctness. Another interesting direction would be generating regression tests, similarly to what tools such as Pynguin (or Java's Evosuite [39]) already do.

[8] `https://github.com/atom-sw/annotest-subjects`

[9] ANNOTEST's current release at the time of writing is version 0.1; this is a more recent version than the one used for the experiments in Chapter 3 that presented the ANNOTEST technique [99], which fixes several minor bugs and introduces minor improvements to its functionality. However, this newer version, which is the version we introduced in this chapter, cannot reproduce one bug in project Visual_Semantic_Alignments (referred to as VSA in Table 3.3). Consequently, instead of 63 bugs as indicated in Table 3.3, the current version of our curated dataset contains 62 bugs.

Part III

Fault Localization

5

An Empirical Study of Fault Localization in Python Programs

Python is one of the most popular programming languages these days,[1][2] especially in novel domains like data science programs. However, there is comparatively little research about fault localization that targets Python. Even though it is plausible that several findings about programming languages like C/C++ and Java—the most common choices for fault localization research—carry over to other languages, whether the dynamic nature of Python and how the language is used in practice affect the capabilities of classic fault localization approaches remain open questions to investigate.

In this chapter, we broaden the dissertation's focus from Python neural network programs to Python open-source programs by presenting the first multi-family large-scale empirical study of fault localization on real-world Python programs and faults. Using Zou et al.'s recent large-scale empirical study of fault localization in Java [145] as the basis of our study, we investigate the effectiveness (i.e., localization accuracy), efficiency (i.e., runtime performance), and other features (e.g., different entity granularity levels) of seven well-known fault-localization techniques in four families (spectrum-based, mutation-based, predicate switching, and stack-trace based) on 135 faults from 13 open-source Python projects from the BugsInPy curated collection [128]. The results replicate for Python several results known about Java, and shed light on whether Python's peculiarities affect the capabilities of fault localization.

Structure of the Chapter

The current chapter is organized as follows:

Section 5.1 highlights the introduction and motivation of the chapter.

Section 5.2 provides the details of various fault localization families used in our study along with a brief overview of FauxPy, the fault localization tool we developed to conduct this chapter's empirical study.

Sections 5.3–5.8 outline our experimental design.

Sections 5.9–5.11 present our experimental results and discussions, along with any limitations and threats to the validity of our findings.

Section 5.12 concludes this chapter with a high-level discussion of the main results and presents possible avenues for future research.

[1] TIOBE language popularity index: `https://www.tiobe.com/tiobe-index/`
[2] Popularity of Programming Language Index: `https://pypl.github.io/PYPL.html`

5.1 Introduction

It is commonplace that debugging is an activity that takes up a disproportionate amount of time and resources in software development [81]. This also explains the popularity of *fault localization* as a research subject in software engineering: identifying locations in a program's source code that are implicated in some observed failures (such as crashes or other kinds of runtime errors) is a key step of debugging. This chapter contributes to the empirical knowledge about the capabilities of fault localization techniques, targeting the Python programming language.

Despite Python's popularity as a programming language, the vast majority of fault localization empirical studies target other languages—mostly C, C++, and Java. To our knowledge, CharmFL [55, 121] is the only available implementation of fault localization techniques for Python; the tool is limited to spectrum-based fault localization (SBFL) techniques. We could not find any realistic-size empirical study of fault localization using Python programs comparing techniques of different families. This gap is in both the availability of tools [109] and the empirical knowledge about fault localization in Python.

This chapter fills this knowledge gap: to our knowledge, it is the first multi-family large-scale empirical study of fault localization in real-world Python programs. The starting point is Zou et al.'s recent extensive study [145] of fault localization for Java. This chapter's main contribution is a differentiated conceptual replication [65] of Zou et al.'s study, sharing several of its features: *i*) it experiments with several different families (spectrum-based, mutation-based, predicate switching, and stack-trace-based) of fault localization techniques; *ii*) it targets a large number of faults in real-world projects (135 faults in 13 projects); *iii*) it studies fault localization effectiveness at different granularities (statement, function, and module); *iv*) it considers combinations of complementary fault localization techniques. The fundamental novelty of our replication is that it targets the Python programming language; furthermore, *i*) it analyzes fault localization effectiveness of different kinds of faults and different categories of projects; *ii*) it estimates the contributions of different fault localization features by means of regression statistical models; *iii*) it compares its main findings for Python to Zou et al.'s [145] for Java.

The main *findings* of our Python fault localization study are as follows:

1. Spectrum-based fault localization techniques are the most effective, followed by mutation-based fault localization (MBFL) techniques.

2. Predicate switching (PS) and stack-trace (ST) fault localization are considerably less effective, but they can work well on small sets of faults that match their characteristics.

3. Stack-trace is by far the fastest fault localization technique, predicate switching and mutation-based fault localization techniques are the most time consuming.

4. Bugs in data-science related projects tend to be harder to localize than those in other categories of projects.

5. Combining fault localization techniques boosts their effectiveness with only a modest hit on efficiency.

6. The main findings about relative effectiveness still hold at all granularity levels.

7. Most of Zou et al. [145]'s findings about fault localization in Java carry over to Python.

A practical challenge to carry out a large-scale fault localization study of Python projects was that, at the time of writing, there were no open-source tools that support a variety of fault localization

techniques for this programming language. Thus, to perform this study, we implemented FAUXPY: a fault-localization tool for Python that supports seven fault localization techniques in four families, is highly configurable, and works with the most common Python unit testing frameworks (such as Pytest and Unittest). Note that the study presented in this chapter is not about FAUXPY. We present FAUXPY in detail in Chapter 6 and briefly in Section 5.2.6. Nevertheless, we make the tool available as part of this chapter's replication package—which also includes all the detailed experimental artifacts and data that support further independent analysis and replicability.[3]

Scope

As noted earlier, the study presented in this chapter is designed based on Zou et al.'s empirical comparison of fault localization on Java programs [145]. We chose their study because it is fairly recent (it was published in 2021), it is comprehensive (it targets 11 fault localization techniques in seven families, as well as combinations of some of these techniques), and it targets realistic programs and faults (357 bugs in five projects from the Defects4J curated collection).

We target a comparable number of subjects (135 BUGSINPY [128] bugs vs. 357 Defects4J [63] bugs) from a wide selection of projects (13 real-world Python projects vs. five real-world Java projects). We study [145]'s four main fault localization families SBFL, MBFL, PS, and ST, but we exclude three other families that featured in their study: DS (dynamic slicing [48]), IRBFL (Information retrieval-based fault localization [144]), and HBFL (history-based fault localization [96]).

IRBFL and HBFL were shown to be scarcely effective by Zou et al. [144], and rely on different kinds of artifacts that may not always be available when dynamically analyzing a program as done by the other "mainstream" fault localization techniques. Namely, IRBFL analyzes bug reports, which may not be available for all bugs; HBFL mines commit histories of programs. In contrast, our study only includes techniques that solely rely on *tests* to perform fault localization; this help make a comparison between techniques consistent.

Finally, we excluded DS for practical reasons: implementing it requires accurate data- and control-dependency static analyses [136]. These are available in languages like Java through widely used frameworks like Soot [68, 123]; in contrast, Python currently offers few mature static analysis tools (e.g, Scalpel [71]), none with the features required to implement DS for the whole Python language. Unfortunately, dynamic slicing has been implemented for Python in the past [23] but no implementation is publicly available; and building it from scratch is outside the present dissertation's scope.

Deep learning models have recently been applied to the software fault localization problem. The key idea of techniques such as DeepFL [72], GRACE [77], and DEEPRL4FL [74] is to train a deep learning model to identify suspicious locations, giving it as input coverage information, as well as other encoded information about the source code of the faulty programs (such as the data and control-flow dependencies). While these approaches are promising, we could not include them in our empirical study since they do not have the same level of maturity as the other "classic" FL techniques we considered. First, DeepFL and GRACE only work at function-level granularity, whereas the bulk of FL research targets statement-level granularity. Second, there are no reference implementations of techniques such as DEEPRL4FL that we can use for our experiments.[4] Third, the performance of a deep learning-based technique usually depends on the training set. Fourth, training a deep learning model is usually a time consuming process; how to account for this overhead when comparing efficiency is tricky.

Nevertheless, our empirical study does feature one FL technique that is based on machine learning: CombineFL, which is Zou et al.'s application of learning to rank to fault localization [145]. The

[3] Replication package: `https://doi.org/10.6084/m9.figshare.23254688`

[4] The replication package of DEEPRL4FL [74] is not available at the time of writing.

same paper also discusses how CombineFL outperforms other state-of-the-art machine learning-based fault localization techniques such as MULTRIC [73], Savant [13], TraPT [73], and FLUCCS [117]. Therefore, CombineFL is a valid representative of the capabilities of pre-deep learning machine learning FL techniques.

Note that numerous recent empirical studies looked into fault localization for deep-learning models implemented in Python [37, 47, 110, 125, 141, 143]. This is a very different problem, using very different techniques, than "classic" program-based fault localization, which is the topic of this chapter.

5.2 Fault Localization and FauxPy

Fault localization techniques [131, 136], an intensely researched topic for over two decades [131], relate program failures (such as crashes or assertion violations) to faulty locations in the program's source code that are responsible for the failures. Concretely, a fault localization technique L assigns a *suspiciousness score* $L_T(e)$ to any program entity e—usually, a location, function, or module—given test inputs T that trigger a failure in the program. The suspiciousness score $L_T(e)$ should be higher the more likely e is the location of a fault that is ultimately responsible for the failure. Thus, a list of all program entities $e_1, e_2, \ldots$ ordered by decreasing suspiciousness score $L_T(e_1) \geq L_T(e_2) \geq \ldots$ is fault localization technique L's overall output.

Let $T = P \cup F$ be a set of tests partitioned into passing P and failing F, such that $F \neq \emptyset$—there is at least one failing test—and all failing tests originate in the same fault. Tests T and a program p are thus the target of a single fault localization run. Then, fault localization techniques differ in what kind of information they extract from T and p to compute suspiciousness scores. A fault localization *family* is a group of techniques that combine the same kind of information according to different formulas.

Sections 5.2.1–5.2.4 describe four common FL families that comprise a total of seven FL techniques. These are the techniques that we used in our experiments. As Section 5.2.5 further explains, a FL technique's *granularity* denotes the kind of program entities it analyzes for suspiciousness—from individual program locations to functions or files/modules. Some FL techniques are only defined for a certain granularity level, whereas others can be applied to different granularity levels.

While FL techniques are usually applicable to any programming language, we could not find any comprehensive implementation of the most common fault localization techniques for Python at the time of writing. Therefore, we implemented FauxPy—an automated fault localization tool for Python implementing several widely used techniques—and used it to perform the empirical study described in this chapter. Section 5.2.6 outlines FauxPy's main features and some details of its implementation.

5.2.1 Spectrum-Based Fault Localization

Techniques in the spectrum-based fault localization (SBFL) family compute suspiciousness scores based on a program's spectra [98]—in other words, its concrete execution traces. The key heuristics of SBFL techniques is that a program entity's suspiciousness is higher the more often the entity is covered (reached) by failing tests and the less often it is covered by passing tests. The various techniques in the SBFL family differ in what formula they use to assign suspiciousness scores based on an entity's coverage in passing and failing tests.

Given tests $T = P \cup F$ as above, and a program entity e: *i*) $P^+(e)$ is the number of passing tests that cover e; *ii*) $P^-(e)$ is the number of passing tests that do not cover e; *iii*) $F^+(e)$ is the number of failing tests that cover e; *iv*) and $F^-(e)$ is the number of failing tests that do not cover e. Figure 5.1

$$\text{Tarantula}_T(e) = \frac{F^+(e)/|F|}{F^+(e)/|F| + P^+(e)/|P|} \tag{5.1}$$

$$\text{Ochiai}_T(e) = \frac{F^+(e)}{\sqrt{|F| \times (F^+(e) + P^+(e))}} \tag{5.2}$$

$$\text{DStar}_T(e) = \frac{(F^+(e))^2}{P^+(e) + F^-(e)} \tag{5.3}$$

Figure 5.1. SBFL formulas to compute the suspiciousness score of an entity e given tests $T = P \cup F$ partitioned into passing P and failing F. All formulas compute a score that is higher the more failing tests $F^+(e)$ cover e, and lower the more passing tests $P^+(e)$ cover e—capturing the basic heuristics of SBFL.

$$\text{Metallaxis}_T(m) = \frac{F^k_{\sim}(m)}{\sqrt{|F| \times (F^k_{\sim}(m) + P^k(m))}} \quad \text{Metallaxis}_T(e) = \max_{\substack{m \in M \\ m \text{ mutates } e}} \text{Metallaxis}_T(m) \tag{5.4}$$

$$\text{Muse}_T(m) = \frac{F^k(m) - P^k(m) \times \sum_{n \in M} F^k(n) / \sum_{n \in M} P^k(n)}{|F|} \quad \text{Muse}_T(e) = \operatorname{mean}_{\substack{m \in M \\ m \text{ mutates } e}} \text{Muse}_T(m) \tag{5.5}$$

Figure 5.2. MBFL formulas to compute the suspiciousness score of a mutant m given tests $T = P \cup F$ partitioned into passing P and failing F. All formulas compute a score that is higher the more failing tests $F^k(m)$ kill m, and lower the more passing tests $P^k(m)$ kill m—capturing the basic heuristics of mutation analysis. On the right, MBFL formulas to compute the suspiciousness score of a program entity e by aggregating the suspiciousness score of all mutants $m \in M$ that modified e in the original program.

shows how Tarantula [61], Ochiai [6], and DStar [130]—three widely used SBFL techniques [94]—compute suspiciousness scores given this coverage information. DStar's formula (5.3), in particular, takes the second power of the numerator, as recommended by other empirical studies [130, 145].[5]

5.2.2 Mutation-Based Fault Localization

Techniques in the mutation-based fault localization (MBFL) family compute suspiciousness scores based on mutation analysis [60], which generates many *mutants* of a program p by applying random transformations to it (for example, change a comparison operator $<$ to $\leq$ in an expression). A mutant m of p is thus a variant of p whose behavior differs from p's at, or after, the location where m differs from p. The key idea of mutation analysis is to collect information about p's runtime behavior based on how it differs from its mutants'. Accordingly, when a test t behaves differently on p than on m (for example, p passes t but m fails it), we say that t *kills* m.

[5] Suspiciousness score formulas are typically expressed as *ratios*; when a denominator is zero, this leads to an undefined score. There are different strategies to account for suspiciousness scores in these degenerate cases [109]. In our experiments, we implicitly add a small constant $\epsilon = 0.1$ to the denominator of every suspiciousness score formula. When the denominator is not zero, adding ϵ is practically irrelevant; when the denominator is zero and the numerator is also zero, adding ϵ gives a very low suspiciousness of zero (which reflects that the entity is hardly covered by any tests); when the denominator is zero and the numerator is positive, adding ϵ gives a large suspiciousness (which reflects that the entity is only covered by failing tests).

To perform fault localization on a program p, MBFL techniques first generate a large number of mutants $M = \{m_1, m_2, \ldots\}$ of p by systematically applying each mutation operator to each statement in p that is executed in any failing test F. Then, given tests $T = P \cup F$ as above, and a mutant $m \in M$: *i*) $P^k(m)$ is the number of tests that p passes but m fails (that is, the tests in P that *kill* m); *ii*) $F^k(m)$ is the number of tests that p fails but m passes (that is, the tests in F that *kill* m); *iii*) and $F^k_{\sim}(m)$ is the number of tests that p fails and behave differently on m, either because they pass on m or because they still fail but lead to a different stack trace (this is a *weaker* notion of tests that *kill* m [91]). Figure 5.2 shows how Metallaxis [91] and Muse [83]—two widely used MBFL techniques—compute suspiciousness scores of each *mutant* in M.

Metallaxis's formula (5.4) is formally equivalent to Ochiai's—except that it is computed for each mutant and measures *killing* tests instead of *covering* tests. In Muse's formula (5.5), $\sum_{n \in M} F^k(n)$ is the total number of failing tests in F that kill any mutant in M, and $\sum_{n \in M} P^k(n)$ is the total number of passing tests in P that kill any mutant in M (these are called *f2p* and *p2f* in Muse's paper [83]).

Finally, MBFL computes a suspiciousness score for a program entity e by aggregating the suspiciousness scores of all mutants that modified e in the original program p; when this is the case, we say that a mutant m *mutates* e. The right-hand side of Figure 5.2 shows Metallaxis's and Muse's suspiciousness formulas for entities: Metallaxis (5.4) takes the largest (maximum) mutant score, whereas Muse (5.5) takes the average (mean) of the mutant scores.

5.2.3 Predicate Switching

The predicate switching (PS) [140] fault localization technique identifies *critical predicates*: branching conditions (such as those of **`if`** and **`while`** statements) that are related to a program's failure. PS's key idea is that if forcibly changing a predicate's value turns a failing test into passing one, the predicate's location is a suspicious program entity.

For each failing test $t \in F$, PS starts from t's execution trace (the sequence of all statements executed by t), and finds t's subsequence $b^t_1\, b^t_2 \ldots$ of *branching* statements. Then, by instrumenting the program p under analysis, it generates, for each branching statement b^t_k, a new execution of t where the *predicate* (branching condition) c^t_k evaluated by statement b^t_k is forcibly *switched* (negated) at runtime (that is, the new execution takes the *other* branch at b^t_k). If switching predicate c^t_k makes the test execution pass, then c^t_k is a *critical predicate*. Finally, PS assigns a (positive) suspiciousness score to all critical predicates in all tests F: $\mathsf{PS}_F(c^t_k)$ is higher, the fewer critical predicates are evaluated between c^t_k and the failure location when executing $t \in F$ [145].[6] For example, the most suspicious program entity e will be the location of the last critical predicate evaluated before any test failure.

PS has some distinctive features compared to other FL techniques. First, it only uses failing tests for its dynamic analysis; any passing tests P are ignored. Second, the only program entities it can report as suspicious are locations of predicates; thus, it usually reports a shorter list of suspicious locations than SBFL and MBFL techniques. Third, while MBFL mutates program code, PS dynamically mutates individual *program executions*. For example, suppose that a loop **`while`** $c : B$ executes its body B twice—and hence, the loop condition c is evaluated three times—in a failing test. Then, PS will generate three variants of this test execution: *i*) one where the loop body never executes (c is false the first time it is evaluated); *ii*) one where the loop body executes once (c is false the second time it is evaluated); *iii*) one where the loop body executes three or more times (c is true the third time it is evaluated).

[6] The actual value of the suspiciousness score is immaterial, as long as the resulting ranking is consistent with this criterion. In our experiments, $\mathsf{PS}_F(c^t_k) = 1/10^d$, where d is the number of critical predicates *other than* c^t_k evaluated after c^t_k in t.

5.2.4 Stack Trace Fault Localization

When a program execution fails with a crash (for example, an uncaught exception), the language runtime usually prints its stack trace (the chain of methods active when the crash occurred) as debugging information to the user. In fact, it is known that stack trace information helps developers debug failing programs [17]; and a bug is more likely to be fixed if it is close to the top of a stack trace [111]. Based on these empirical findings, Zou et al. [145] proposed the stack trace fault localization technique (ST), which uses the simple heuristics of assigning suspiciousness based on how close a program entity is to the top of a stack trace.

Concretely, given a failing test $t \in F$, its *stack trace* is a sequence $f_1 f_2 \ldots$ of the stack frames of all functions that were executing when t terminated with a failure, listed in reverse order of execution; thus, f_1 is the most recently called function, which was directly called by f_2, and so on. ST assigns a (positive) suspiciousness score to any program entity e that belongs to any function f_k in t's stack trace: $\mathsf{ST}_t(e) = 1/k$, so that e's suspiciousness is higher, the closer to the failure e's function was called.[7] In particular, the most suspicious program entities will be all those in the function f_1 called in the top stack frame. Then, the overall suspiciousness score of e is the maximum in all failing tests F: $\mathsf{ST}_F(e) = \max_{t \in F} \mathsf{ST}_t(e)$.

5.2.5 FL Granularities

Fault localization *granularity* refers to the kinds of program entity that a FL technique ranks. The most widely studied granularity is *statement-level*, where each statement in a program may receive a different suspiciousness score [94, 130]. However, coarser granularities have also been considered, such as *function-level* (also called method-level) [13, 133] and *module-level* (also called file-level) [107, 144].

In practice, implementations of FL techniques that support different levels of granularity focus on the finest granularity (usually, statement-level granularity), whose information they use to perform FL at coarser granularities. Namely, the suspiciousness of a function is the maximum suspiciousness of any statements in its definition; and the suspiciousness of a module is the maximum suspiciousness of any functions belonging to it [145].

5.2.6 FauxPy: Features and Implementation

Despite its popularity as a programming language, we could not find off-the-shelf implementations of fault localization techniques for Python at the time of writing [109]. The only exception is CharmFL [55]—a plugin for the PyCharm IDE—which only implements SBFL techniques. Therefore, to conduct an extensive empirical study of FL in Python, we implemented FauxPy: a fault localization tool for Python programs.

FauxPy supports all seven FL techniques in four families described in Sections 5.2.1–5.2.4, which are the spectrum-based (SBFL) techniques DStar [130], Ochiai [6], and Tarantula [61]; the mutation-based (MBFL) techniques Muse [83] and Metallaxis [91]; and the predicate switching (PS) [140] and stack trace (ST) [145] fault localization families/techniques. FauxPy can localize faults at the level of statements, functions, or modules (Section 5.2.5). To make FauxPy a flexible and extensible tool, easy to use with a variety of other commonly used Python development tools, we implemented it as a stand-alone command-line tool that works with tests in the formats supported by Pytest, Unittest, and Hypothesis [80]—three popular Python testing frameworks.

[7] As in PS, the actual value of the suspiciousness score is immaterial, as long as the resulting ranking is consistent with this criterion.

While running, FAUXPY stores intermediate analysis data in an SQLite database; upon completing a FL localization run, it returns to the user a human-readable summary—including suspiciousness scores and ranking of program entities. The database improves performance (for example by caching intermediate results) but also facilitates *incremental* analyses—for example, where we provide different batches of tests in different runs.

FAUXPY's implementation uses Coverage.py [15]—a popular code-coverage measurement library—to collect the execution traces needed for SBFL and MBFL. It also uses the state-of-the-art mutation-testing framework Cosmic Ray [27] to generate mutants for MBFL; since Cosmic Ray is easily configurable to use some or all of its mutation operators—or even to add new user-defined mutation operators—FAUXPY's MBFL implementation is also fully configurable. To implement PS in FAUXPY, we developed an instrumentation library that can selectively change the runtime value of predicates in different runs as required by the PS technique. Chapter 6 focuses on FAUXPY as a tool, providing a thorough examination of its features and implementation.

5.3 Research Questions

Our experiments assess and compare the effectiveness and efficiency of the seven FL techniques supported by FAUXPY (see Section 5.2.6), as well as of their combinations, on real-world Python programs and faults. To this end, we target the following research questions:

RQ1. How *effective* are the fault localization techniques?
RQ1 compares fault localization techniques according to how accurately they identify program entities that are responsible for a fault.

RQ2. How *efficient* are the fault localization techniques?
RQ2 compares fault localization techniques according to their running time.

RQ3. Do fault localization techniques behave differently on *different* faults?
RQ3 investigates whether the fault localization techniques' effectiveness and efficiency depend on which kinds of faults and programs it analyzes.

RQ4. Does *combining* fault localization techniques improve their effectiveness?
RQ4 studies whether combining the information of different fault localization techniques for the same faults improves the effectiveness compared to applying each technique in isolation.

RQ5. How does program entity *granularity* impact fault localization effectiveness?
RQ5 analyzes the relation between effectiveness and granularity: does the relative effectiveness of fault localization techniques change as they target coarser-grained program entities?

RQ6. Are fault localization techniques as effective on Python programs as they are on *Java* programs?
RQ6 compares our overall results to Zou et al. [145]'s, exploring similarities and differences between Java and Python programs.

5.4 Experimental Subjects

To have a representative collection of realistic Python bugs, we used BUGSINPY [128], a curated dataset of real bugs collected from real-world Python projects, with all the information needed to reproduce the bugs in controlled experiments. Table 5.1 overviews BUGSINPY's 501 bugs from 17 projects.

PROJECT	KLOC	|F|	|M|	BUGS	SUBJECTS	TESTS	TEST KLOC	CATEGORY	DESCRIPTION
ansible	82.6	3 713	493	18	0	1 830	103.1	DEV	IT automation platform
black	93.5	421	27	23	13	153	6.8	DEV	Code formatter
cookiecutter	1.6	62	18	4	4	218	4.1	DEV	Developer tool
fastapi	4.7	160	40	16	13	595	16.8	WEB	Web framework for building APIs
httpie	3.5	197	34	5	4	217	2.4	CL	Command-line HTTP client
keras	6.7	150	119	45	18	616	13.6	DS	Deep learning API
luigi	22.0	2 004	120	33	13	1 508	21.2	DEV	Pipelines of batch jobs management tool
matplotlib	99.6	5 526	147	30	0	2 484	34.9	DS	Plotting library
pandas	128.0	5 466	234	169	18	12 226	200.9	DS	Data analysis toolkit
PySnooper	0.7	60	7	3	0	49	3.9	DEV	Debugging tool
sanic	7.3	462	61	5	3	466	8.3	WEB	Web server and web framework
scrapy	15.7	1 509	179	40	0	1 572	24.5	WEB	Web crawling and web scraping framework
spaCy	97.2	852	415	10	6	986	13.4	DS	Natural language processing library
thefuck	4.7	604	203	32	16	614	7.3	CL	Console command tool
tornado	17.9	1 124	35	16	4	926	13.1	WEB	Web server
tqdm	3.3	200	28	9	7	120	2.7	CL	Progress bar for Python and CLI
youtube-dl	125.0	3 078	818	43	16	237	5.1	CL	Video downloader
total	714.0	25 588	2 978	501	135	24 817	482.1		

Table 5.1. Overview of projects in BUGSINPY. For each PROJECT, the table reports the project's overall size in KLOC (thousands of non-empty non-comment lines of code, excluding tests), the number |F| of functions (excluding test functions), the number |M| of modules (excluding test modules), the number of BUGS included in BUGSINPY, how many we selected as SUBJECTS for our experiments, the corresponding number of TESTS (i.e., test functions), their size in kLOC (TEST KLOC, thousands of non-empty non-comment lines of test code), the CATEGORY the project belongs to (CL: command line; DEV: development tools; DS: data science; WEB: web tools), and a brief DESCRIPTION of the project. Consistently with what done by the authors of BUGSINPY [128], the project statistics reported here refer to the *latest* version of the projects on 2020-06-19.

Project category. Columns CATEGORY in Table 5.1 and Table 5.2 partition all BUGSINPY projects into four non-overlapping categories:

Command line (CL) projects consist of tools mainly used through their command line interface.

Development (DEV) projects offer libraries and utilities useful to software developers.

Data science (DS) projects consist of machine learning and numerical computation frameworks.

Web (WEB) projects offer libraries and utilities useful for web development.

We classified the projects according to their description in their respective repositories, as well as how they are presented in BUGSINPY. Like any classification, the boundaries between categories may be somewhat fuzzy, but the main focus of most projects is quite obvious (such as DS for keras and pandas, or CL for youtube-dl).

Unique bugs. Each bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ in BUGSINPY consists of: *i*) a *faulty* version p_b^- of the project, such that tests in F_b all fail on it (all due to the same root cause); *ii*) a *fixed* version p_b^+ of the project, such that all tests in $F_b \cup P_b$ pass on it; *iii*) a collection of *failing* F_b and *passing* P_b tests, such that tests in P_b pass on both the faulty p_b^- and fixed p_b^+ versions of the project, whereas tests in F_b fail on the faulty p_b^- version and pass on the fixed p_b^+ version of the project.

Bug selection. Despite BUGSINPY's careful curation, several of its bugs cannot be reproduced because their dependencies are missing or no longer available; this is a well-known problem that plagues reproducibility of experiments involving Python programs [84]. In order to identify which BUGSINPY bugs were reproducible at the time of our experiments on our infrastructure, we took the following steps for each bug b:

i) Using BUGSINPY's scripts, we generated and executed the faulty p_b^- version and checked that tests in F_b fail whereas tests in P_b pass on it; and we generated and executed the fixed p_b^+ version and checked that all tests in $F_b \cup P_b$ pass on it. Out of all of BUGSINPY's bugs, 120 failed this step; we did not include them in our experiments.

ii) Python projects often have two sets of dependencies (*requirements*): one for users and one for developers; both are needed to run fault localization experiments, which require to instrument the project code. Another 39 bugs in BUGSINPY miss some development dependencies; we did not include them in our experiments.

iii) Two bugs resulted in an empty ground truth (Section 5.5): essentially, there is no way of localizing the fault in p_b^-; we did not include these bugs in our experiments.

This resulted in $501 - 120 - 39 - 2 = 340$ bugs in 13 projects (all but ansible, matplotlib, PySnooper, and scrapy) that we could reproduce in our experiments.

However, this is still an impractically large number: just *reproducing* each of these bugs in BUGSINPY takes nearly a full week of running time, and each FL experiment may require to rerun the same tests several times (hundreds of times in the case of MBFL). Thus, we first discarded 27 bugs that each take more than 48 hours to reproduce. We estimate that including these 27 bugs in the experiments would have taken over 14 CPU-months just for the MBFL experiments—not counting other FL techniques, nor the time for setup and dealing with unexpected failures.

Running all the fault localization experiments for each of the remaining $313 = 340 - 27$ bugs takes approximately eleven CPU-hours, for a total of nearly five CPU-months. We selected 135 bugs out of the 313 using stratified random sampling with the four project categories as the "strata", picking: 43 bugs in category CL, 30 bugs in category DEV, 42 bugs in category DS, and 20 bugs in category WEB. This gives us a still sizable, balanced, and representative[8] sample of all bugs in BUGSINPY, which we could exhaustively analyze in around two CPU-months worth of experiments. In all, we used this selection of 135 bugs as our empirical study's subjects. Table 5.2 gives some details about the selected projects and their bugs.

As a side comment, note that our experiments with BUGSINPY were generally more time consuming than Zou et al.'s experiments with Defects4J. For example, the average per-bug running time of MBFL in our experiments (15 774 seconds in Table 5.6) was 3.3 times larger than in Zou et al.'s (4800 seconds in [145, Table 9]). Even more strikingly, running all fault localization experiments on the 357 Defects4J bugs took less than one CPU-month;[9] in contrast, running MBFL on just 27 "time

[8] For example, this sample size is sufficient to estimate a ratio with up to 5.5% error and 90% probability with the most conservative (i.e., 50%) a priori assumption [28].

[9] The sum of column AVERAGE in [145, Table 9] multiplied by 357 gives 2.04 million seconds or 0.79 months.

CATEGORY	PROJECT	BUGS (SUBJECTS)		TESTS		GROUND TRUTH	
		C	*P*	*C*	*P*	*C*	*P*
CL	httpie	43	4	1188	217	139	12
	thefuck		16		614		55
	tqdm		7		120		22
	youtube-dl		16		237		50
DEV	black	30	13	1879	153	300	208
	cookiecutter		4		218		19
	luigi		13		1508		73
DS	keras	42	18	13828	616	186	111
	pandas		18		12226		64
	spaCy		6		986		11
WEB	fastapi	20	13	1987	595	174	156
	sanic		3		466		6
	tornado		4		926		12
	total	135	135	18882	18882	799	799

Table 5.2. Selected BUGSINPY bugs used in this chapter's experiments. The PROJECTs are grouped by CATEGORY; the table reports—for each project individually (column *P*), as well as for all projects in the category (column *C*)—the number of BUGS selected as SUBJECTS for our experiments, the corresponding number of TESTS (i.e., test functions), and the total number of program locations that make up the GROUND TRUTH (described in Section 5.5).

consuming" bugs in BUGSINPY takes over 14 CPU-months. This difference may be partly due to the different characteristics of projects in Defects4J vs. BUGSINPY, and partly to the dynamic nature of Python (which is run by an interpreter).

5.5 Faulty Locations: Ground Truth

A fault localization technique's effectiveness measures how accurately the technique's list of suspicious entities matches the actual fault locations in a program—fault localization's *ground truth*. It is customary to use programmer-written patches as ground truth [94, 145]: the program locations modified by the patches that fix a certain bug correspond to the bug's actual fault locations.

Concretely, here is how to determine the ground truth of a bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ in BUGSINPY. The programmer-written fix p_b^+ consists of a series of *edits* to the faulty program p_b^-. Each edit can be of three kinds: *i*) *add*, which inserts into p_b^+ a new program location; *ii*) *remove*, which deletes a program location in p_b^-; *iii*) *modify*, which takes a program location in p_b^- and changes parts of it, without changing its location, in p_b^+. Take, for instance, the program in Figure 5.3b, which modifies the program in Figure 5.3a; the edited program includes two adds (lines 175, 184), one remove (line 188), and one modify (line 181).

Bug b's *ground truth* $\mathscr{F}(b)$ is a set of locations in p_b^- that are affected by the edits, determined as follows. First of all, ignore any blank or comment lines, since these do not affect a program's behavior and hence cannot be responsible for a fault. Then, finding the ground truth locations corresponding to removes and modifies is straightforward: a location ℓ that is removed or modified in p_b^+ exists by definition also in p_b^-, and hence it is part of the ground truth. In Figure 5.3, line 162 is modified and line 169 is removed by the edit that transforms Figure 5.3a into Figure 5.3b; thus 162 and 169 are

```
153         a = 3
154
155
156
157
158         c = 5
159
160         # Function foo
161         def foo(y):
162         if y > 3:
163         a = y
164         y = y * 2
165
166
167         # Function bar
168         def bar(z):
169         z = z + 2
170         return z
171
```

(a) Faulty program version. Lines with colored background are the ground truth locations. Extra blank lines are added for readability.

```
172         a = 3
173
174         # Global variable b
175         b = None          # add
176
177         c = 5
178
179         # Function foo
180         def foo(y):
181         if y > 100:       # modify
182         a = y
183         y = y * 2
184         a = y             # add
185
186         # Function bar
187         def bar(z):
188         z = z + 2         # remove
189         return z
190
```

(b) Fixed program version, which edits Figure 5.3a's program with two adds, one modify, and one remove.

Figure 5.3. An example of program edit, and the corresponding ground truth faulty locations.

part of the example's ground truth.

Finding the ground truth locations corresponding to *adds* is more involved [109], because a location ℓ that is added to p_b^+ does not exist in p_b^-: b is a fault of omission [94].[10] A common solution [94, 145] is to take as ground truth the location in p_b^- that immediately *follows* ℓ. In Figure 5.3, line 158 corresponds to the first non-blank line that follows the assignment statement that is added at line 175 in Figure 5.3b; thus 158 is part of the example's ground truth. However, an add at ℓ is actually a modification between two other locations; therefore, the location that immediately *precedes* ℓ should also be part of the ground truth, since it identifies the same insertion location. In Figure 5.3, line 153 precedes the assignment statement that is added at line 175 in Figure 5.3a; thus 153 is also part of the example's ground truth.

A location's *scope* poses a final complication to determine the ground truth of adds. Consider line 184, added in Figure 5.3b at the very end of function `foo`'s body. The (non-blank, non-comment) location that follows it in Figure 5.3a is line 168; however, line 168 marks the beginning of another function `bar`'s definition. Function `bar` cannot be the location of a fault in `foo`, since the two functions are independent—in fact, the fact that `bar`'s declaration follows `foo`'s is immaterial. Therefore, we only include a location in the ground truth if it is within the same *scope* as the location ℓ that has been added. If ℓ is part of a function body (including methods), its scope is the function declaration; if ℓ is part of a class outside any function (e.g., an attribute), its scope is the class declaration; and otherwise ℓ's scope is the module it belongs to. In Figure 5.3, both lines 153 and 158 are within the same module as the added statement at line 175 in Figure 5.3a. In contrast, line 168 is within a different scope than the added statement at line 184 in Figure 5.3a. Therefore, lines 153, 158, and 164 are part of the ground truth, but not line 168.

Our definition of ground truth refines that used in related work [94, 145] by including the lo-

[10] In BugsInPy, 41% of all fixes include at least one add edit.

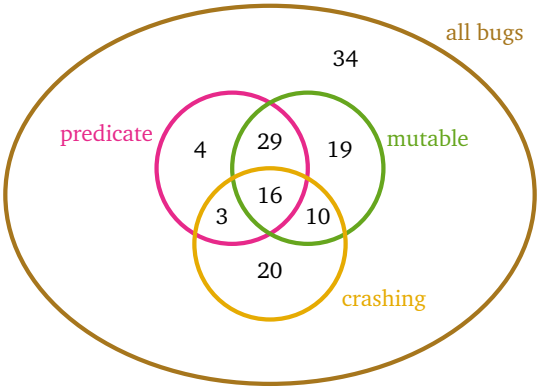


Figure 5.4. Classification of the 135 BUGSINPY bugs used in our experiments into three categories.

cation that precedes an add, and by considering only locations within scope. We found that this definition better captures the programmer's intent and their corrective impact on a program's behavior.

How to best characterize bugs of omissions (fixed by an add) in fault localization remains an open issue [109]. Pearson et al.'s study [94] proposed the first viable solution: including the location following an add. Zou et al. [145] followed the same approach, and hence we also include the location following an add in our ground truth computation. We also noticed that, by also including the location preceding an add, and by taking scope into account, our ground truth computation becomes more comprehensive; in particular, it also works for statements added at the very end of a file—a location that has no following lines.

While our approach is usually more precise, it is not necessarily the preferable alternative in all cases. Consider again, for instance, the add at line 184 in Figure 5.3; if we ignored the scope (and the preceding statement), only line 168 would be included in its ground truth. If this fault localization information were consumed by a developer, it could still be useful and actionable even if it reports a line outside the scope of the actual add location: the developer would use the location as a starting point for their inspection of the nearby code; and they may prefer a smaller, if slightly imprecise, ground truth to a larger, redundant one. However, our study's focus is strictly evaluating the effectiveness of FL techniques as rigorously as possible—for which our stricter ground truth computation is more appropriate.

5.6 Classification of Faults

Bug kind. The information used by each fault localization technique naturally captures the behavior of different *kinds* of faults. Stack trace fault localization analyzes the call stack after a program terminates with a crash; predicate switching targets branching conditions as program entities to perform fault localization; and MBFL crucially relies on the analysis of mutants to track suspicious locations.

Correspondingly, we classify a bug $b = \langle p_b^-, p_b^+, F_b, P_b \rangle$ as:

Crashing bug if any failing test in F_b terminates abruptly with an unexpected uncaught exception.

Predicate bug if any faulty entity in the ground truth $\mathscr{F}(b)$ includes a branching predicate (such as an `if` or `while` condition).

Mutable bug if any of the mutants generated by MBFL's mutation operators mutates any locations in the ground truth $\mathscr{F}(b)$. Precisely, a bug b's *mutability* is the percentage of all mutants of p_b^- that mutate locations in $\mathscr{F}(b)$; and b is mutable if its mutability is greater than zero.

The notion of crashing and predicate bugs is from Zou et al. [145]. We introduced the notion of mutable bug to try to capture scenarios where MBFL techniques have a fighting chance to correctly localize bugs. Since MBFL uses mutant analysis for fault localization, its capabilities depend on the mutation operators that are used to generate the mutants. Therefore, the notion of mutable bugs is somewhat dependent on the applied mutation operators.[11] Our implementation of FAUXPY uses the standard operators offered by the popular Python mutation testing framework Cosmic Ray [27]. As we discuss in Section 5.10.2, Cosmic Ray features a set of mutation operators that are largely similar to several other general-purpose mutation testing frameworks—all based on Offut et al.'s well known work [86]. These strong similarities between the mutation operators offered by most widely used mutation testing frameworks suggest that our definition of "mutable bug" is not strongly dependent on the specific mutation testing framework that is used. Correspondingly, bugs that we classify as "mutable" are likely to remain amenable to localization with MBFL provided one uses (at least) this standard set of core mutation operators. Conversely, we expect that devising new, specialized mutation operators may extend the number of bugs that we can classify as "mutable", and hence that are more likely to be amenable to localization with MBFL techniques.

Figure 5.4 shows the kind of the 135 BUGSINPY bugs we used in the experiments, consisting of 49 crashing bugs, 52 predicate bugs, 74 mutable bugs, and 34 bugs that do not belong to any of these categories.

Project category. Another, orthogonal classification of bugs is according to the project *category* they belong to, explained in Section 5.4. We classify a bug b as a CL, DEV, DS, or WEB bug according to the category of project (Table 5.2) b belongs to.

5.7 Evaluation Metrics

In this section, we detail the evaluation metrics used in our experiments, including both established classic metrics from the literature and our custom-designed metrics. We also provide insights into our statistical analysis methodologies.

5.7.1 Ranking Program Entities

Running a fault localization technique L on a bug b returns a list of program entities $\ell_1, \ell_2, \ldots$, sorted by their decreasing suspiciousness scores $s_1 \geq s_2 \geq \ldots$. The programmer (or, more realistically, a tool [44, 92]) will go through the entities in this order until a faulty entity (that is an $\ell \in \mathscr{F}(b)$ that matches b's ground truth) is found. In this idealized process, the earlier a faulty entity appears in the list, the less time the programmer will spend going through the list, the more effective fault localization technique L is on bug b. Thus, a program entity's *rank* in the sorted list of suspicious entities is a key measure of fault localization effectiveness.

[11] In this sense, "mutable" is a qualitatively different attribute than "crashing" and "predicate". Whether a bug b is "crashing" exclusively depends on the failing tests that trigger the bug; whether b is a "predicate" bug depends on the branching syntactic structure of b's program and how it relates to b. In contrast, whether b is a "mutable" bug depends on the mutation operators used to analyze b, and on whether they can change the program so as to effectively affect b's buggy behavior.

	PROGRAM ENTITY ℓ									
	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7	ℓ_8	ℓ_9	ℓ_{10}
suspiciousness score s of ℓ	10	7	4	4	4	3	3	2	2	2
$\ell \in \mathscr{F}(b)$?		✖		✖				✖	✖	
start(ℓ)	1	2	3	3	3	6	6	8	8	8
ties(ℓ)	1	1	3	3	3	2	2	3	3	3
faulty(ℓ)	0	1	1	1	1	0	0	2	2	2
$\mathscr{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$	1.0	2.0	4.0	4.0	4.0	6.0	6.0	8.3	8.3	8.3

Table 5.3. An example of calculating the E_{inspect} metric $\mathscr{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$ for a list of 10 suspicious locations $\ell_1, \ldots, \ell_{10}$ ordered by their decreasing suspiciousness scores $s_1, \ldots, s_{10}$. For each location ℓ, the table reports its suspiciousness score s, and whether ℓ is a faulty location $\ell \in \mathscr{F}(b)$; based on this ranking of locations, it also shows the lowest rank start(ℓ) of the first location whose score is equal to ℓ's, the number ties(ℓ) of locations whose score is equal to ℓ's, the number of faulty locations among these, and the corresponding E_{inspect} value $\mathscr{I}_b(\ell, L)$—computed according to (5.6).

Computing a program entity ℓ's rank is trivial if there are no *ties* between scores. For example, consider Table 5.3's first two program entities ℓ_1 and ℓ_2, with suspiciousness scores $s_1 = 10$ and $s_2 = 7$. Obviously, ℓ_1's rank is 1 and ℓ_2's is 2; since ℓ_2 is faulty ($\ell_2 \in \mathscr{F}(b)$), its rank is also a measure of how many entities will need to be inspected in the aforementioned debugging process.

When several program entities tie the same suspiciousness score, their relative order in a ranking is immaterial [30]. Thus, it is a common practice to give all of them the same *average* rank [109, 118], capturing an average-case number of program entities inspected while going through the fault localization output list. For example, consider Table 5.3's first five program entities $\ell_1, \ldots, \ell_5$; ℓ_3, ℓ_4, and ℓ_5 all have the same suspiciousness score $s = 4$. Thus, they all have the same average rank $4 = (3 + 4 + 5)/3$, which is a proxy of how many entities will need to be inspected if ℓ_4 were faulty but ℓ_2 were not.

Capturing the "average number of inspected entities" is trickier still if more than one entity is faulty among a bunch of tied entities. Consider now all of Table 5.3's ten program entities; entities ℓ_8, ℓ_9, and ℓ_{10} all have the suspiciousness score $s = 2$; ℓ_8 and ℓ_9 are faulty, whereas ℓ_{10} is not. Their average rank $9 = (8+9+10)/3$ overestimates the number of entities to be inspected (assuming now that these are the only faulty entities in the output), since two entities out of three are faulty, and hence it is more likely that the faulty entity will appear before rank 9.

To properly account for such scenarios, Zou et al. [145] introduced the E_{inspect} metric, which ranks a program entity ℓ within a list $\langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of program entities $\ell_1, \ldots, \ell_n$ with suspiciousness scores $s_1 \geq \ldots \geq s_n$ as:

$$\mathscr{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle) \quad = \quad \mathsf{start}(\ell) + \sum_{k=1}^{\mathsf{ties}(\ell)-\mathsf{faulty}(\ell)} k \, \frac{\binom{\mathsf{ties}(\ell)-k-1}{\mathsf{faulty}(\ell)-1}}{\binom{\mathsf{ties}(\ell)}{\mathsf{faulty}(\ell)}} \tag{5.6}$$

In (5.6), start(ℓ) is the position k of the first entity among those with the same score as ℓ's; ties(ℓ) is the number of entities (including ℓ itself) whose score is the same as ℓ's; and faulty(ℓ) is the number of entities (including ℓ itself) that tie ℓ's score and are faulty (that is $\ell \in \mathscr{F}(b)$). Intuitively, the E_{inspect} rank $\mathscr{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$ is thus an average of all possible ranks where tied and faulty entities are shuffled randomly. When there are no ties, or only one entity among a group of ties is faulty, (5.6) coincides with the average rank. Henceforth, we refer to a location's E_{inspect} rank $\mathscr{I}_b(\ell, \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle)$ as simply its *rank*.

Better vs. worse ranks. A clarification about terminology: a *high* rank is a rank that is close to the top-1 rank (the first rank), whereas a *low* rank is a rank that is further away from the top-1 rank. Correspondingly, a high rank corresponds to a small numerical ordinal value; and a low rank corresponds to a large numerical ordinal value. Consistently with this standard usage, the rest of the chapter refers to "better" ranks to mean "higher" ranks (corresponding to smaller ordinals); and "worse" ranks to mean "lower" ranks (corresponding to larger ordinals).

5.7.2 Fault Localization Effectiveness Metrics

E_{inspect} effectiveness. Building on the notion of *rank*—defined in Section 5.7.1—we measure the *effectiveness* of a fault localization technique L on a bug b as the rank of the first faulty program entity in the list $L(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of entities and suspiciousness scores returned by L running on b—defined as $\mathscr{I}_b(L)$ in (5.7). $\mathscr{I}_b(L)$ is L's E_{inspect} rank on bug b, which estimates the number of entities in L's one has to inspect to correctly localize b.

Generalized E_{inspect} effectiveness. What happens if a FL technique L cannot localize a bug b—that is, b's faulty entities $\mathscr{F}(b)$ do not appear at all in L's output? According to (5.6) and (5.7), $\mathscr{I}_b(L)$ is *undefined* in these cases. This is not ideal, as it fails to measure the effort wasted going through the location list when using L to localize b—the original intuition behind all rank metrics. Thus, we introduce a generalization L's E_{inspect} rank on bug b as follows. Given the list $L(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle$ of entities and suspiciousness scores returned by L running on b, let $L^{\infty}(b) = \langle \ell_1, s_1 \rangle \ldots \langle \ell_n, s_n \rangle \langle \ell_{n+1}, s_0 \rangle \langle \ell_{n+2}, s_0 \rangle \ldots$ be $L(b)$ followed by all *other entities* $\ell_{n+1}, \ell_{n+1}, \ldots$ in program p_b^- that are not returned by L, each given a suspiciousness $s_0 < s_n$ lower than any suspiciousness scores assigned by L.

With this definition, $\mathscr{I}_b(L) = \widetilde{\mathscr{I}}_b(L)$ whenever L can localize b—that is some entity from $\mathscr{F}(b)$ appears in L's output list. If some technique L_1 can localize b whereas another technique L_2 cannot, $\widetilde{\mathscr{I}}_b(L_2) > \widetilde{\mathscr{I}}_b(L_1)$, thus reflecting that L_2 is worse than L_1 on b. Finally, if neither L_1 nor L_2 can localize b, $\widetilde{\mathscr{I}}_b(L_2) > \widetilde{\mathscr{I}}_b(L_1)$ if L_2 returns a longer list than L_1: all else being equal, a technique that returns a shorter list is "better" than one that returns a longer list since it requires less of the user's time to inspect the output list. Accordingly, $\widetilde{\mathscr{I}}_b(L)$ denotes L's *generalized* E_{inspect} rank on bug b—defined as in (5.7).

Exam score effectiveness. Another commonly used effectiveness metric is the *exam score* $\mathscr{E}_b(L)$ [129], which is just a FL technique L's E_{inspect} rank on bug b over the number of program entities $|p_b^-|$ of the analyzed buggy program p_b^-—as in (5.7). Just like $\mathscr{I}_b(L)$, $\mathscr{E}_b(L)$ is undefined if L cannot localize b.

Effectiveness of a technique. To assess the overall effectiveness of a FL technique over a set B of bugs, we aggregate the previously introduced metrics in different ways—as in (5.8). The $L@_B n$ metric counts the number of bugs in B that L could localize within the top-n positions (according to their E_{inspect} rank); $n = 1, 3, 5, 10$ are common choices for n, reflecting a "feasible" number of entities to inspect. Then, the $L@_B n\% = 100 \cdot L@_B n/|B|$ metric is simply $L@_B n$ expressed as a percentage of the number $|B|$ of bugs in B. $\widetilde{\mathscr{I}}_B(L)$ is L's average generalized E_{inspect} rank of bugs in B. And $\mathscr{E}_B(L)$ is L's average exam score of bugs in B (thus ignoring bugs that L cannot localize).

Location list length. The $|L_b|$ metric is simply the number of suspicious locations output by FL technique L when run on bug b; and $|L_B|$ is the average of $|L_b|$ for all bugs in B. The location list length

$$\mathscr{I}_b(L) = \min_{\ell \in L(b) \cap \mathscr{F}(b)} \mathscr{I}_b(\ell, L(b)) \qquad \widetilde{\mathscr{I}}_b(L) = \min_{\ell \in L^{\infty}(b) \cap \mathscr{F}(b)} \mathscr{I}_b(\ell, L^{\infty}(b)) \qquad \mathscr{E}_b(L) = \frac{\mathscr{I}_b(L)}{|p_b^-|} \tag{5.7}$$

$$L@_B n = \left|\left\{b \in B \mid \mathscr{I}_b(L) \leq n\right\}\right| \qquad \widetilde{\mathscr{I}}_B(L) = \frac{1}{|B|} \sum_{b \in B} \widetilde{\mathscr{I}}_b(L) \qquad \mathscr{E}_B(L) = \frac{1}{|B|} \sum_{b \in B} \mathscr{E}_b(L) \tag{5.8}$$

Figure 5.5. Definitions of common FL effectiveness metrics. The top row shows two variants $\mathscr{I}$, $\widetilde{\mathscr{I}}$ of the E_{inspect} metric, and the exam score $\mathscr{E}$, for a generic bug b and fault localization technique L. The bottom row shows cumulative metrics for a set B of bugs: the "at n" metric $L@_B n$, and the average $\widetilde{\mathscr{I}}$ and $\mathscr{E}$ metrics.

metric is not, strictly speaking, a measure of effectiveness; rather, it complements the information provided by other measures of effectiveness, as it gives an idea of how much output a technique produces to the user. All else being equal, a shorter location list length is preferable—provided it is not empty. In practice, we'll compare the location list length to other metrics of effectiveness, in order to better understand the trade-offs offered by each FL technique.

Different FL families use different kinds of information to compute suspiciousness scores; this is also reflected by the entities that may appear in their output location list. SBFL techniques include all locations executed by any tests T_b (passing or failing) even if their suspiciousness is zero; conversely, they omit all locations that are *not* executed by the tests. MBFL techniques include all locations executed by any *failing* tests F_b, since these locations are the targets of the mutation operators. PS includes all locations of *predicates* (branching conditions) that are executed by any failing tests F_b and that are *critical* (as defined in Section 5.2.3). ST includes all locations of all functions that appear in the stack trace of any crashing test in F_b.

Effectiveness metrics: limitations. Despite being commonly used in fault localization research, the effectiveness metrics presented in this section rely on assumptions that may not realistically capture the debugging work of developers. First, they assume that a developer can understand the characteristics of a bug and devise a suitable fix by examining just one buggy entity; in contrast, debugging often involves disparate activities, such as analyzing control and data dependencies and inspecting program states with different inputs [93]. Second, debugging is often not a *linear* sequence of activities [67] as simple as going through the ranked list of entities produced by fault localization techniques. Despite these limitations, we still rely on this section's effectiveness metrics: on the one hand, they are used in practically all related work on fault localization (in particular, Zou et al. [144]); thus, they make our results comparable to others. On the other hand, there are no viable, easy-to-measure alternative metrics that are also fully realistic; devising such metrics is outside this dissertation's scope and belongs to future work.

5.7.3 Comparison: Statistical Models

To quantitatively compare the capabilities of different fault localization techniques, we consider several standard statistics.

Pairwise comparisons. Let $M_b(L)$ be any metric M measuring the capabilities of fault-localization technique L on bug b; M can be any of Section 5.7.2's effectiveness metrics, or L's wall-clock running time $T_b(L)$ on bug b as performance metric. Similarly, for a fault-localization family F, $M_b(F)$ denotes the average value $\sum_{k \in F} M_b(k)/|F|$ of M_b for all techniques in family F. Given a

set $B = \{b_1, \ldots, b_n\}$ of bugs, we compare the two vectors $M_B(F_1) = \langle M_{b_1}(F_1) \ldots M_{b_n}(F_1)\rangle$ and $M_B(F_2) = \langle M_{b_1}(F_2) \ldots M_{b_n}(F_2)\rangle$ using three statistics:

Correlation τ between $M_B(F_1)$ and $M_B(F_2)$ computed using Kendall's τ statistics. The absolute value $|\tau|$ of the correlation τ measures how closely changes in the value of metric M for F_1 over different bugs are *associated* to changes for F_2 over the same bugs: if $0 \leq |\tau| \leq 0.3$ the correlation is *negligible*; if $0.3 < |\tau| \leq 0.5$ the correlation is *weak*; if $0.5 < |\tau| \leq 0.7$ the correlation is *medium*; and if $0.7 < |\tau| \leq 1$ the correlation is *strong*.

P-value p of a paired Wilcoxon signed-rank test—a nonparametric statistical test comparing $M_B(F_1)$ and $M_B(F_2)$. A small value of p is commonly taken as evidence against the "null-hypothesis" that the distributions underlying $M_B(F_1)$ and $M_B(F_2)$ have different medians:[12] usually, $p \leq 0.05$, $p \leq 0.01$, and $p \leq 0.001$ are three conventional thresholds of increasing strength.

Cliff's δ effect size—a nonparametric measure of how often the values in $M_B(F_1)$ are larger than those in $M_B(F_2)$. The absolute value $|\delta|$ of the effect size δ measures how much the values of metric M differ, on the same bugs, between F_1 and F_2 [105]: if $0 \leq |\delta| < 0.147$ the differences are *negligible*; if $0.145 \leq |\delta| < 0.33$ the differences are *small*; if $0.33 \leq |\delta| < 0.474$ the differences are *medium*; and if $0.474 \leq |\delta| \leq 1$ the differences are *large*.

Regression models. To ferret out the individual impact of several different factors (fault localization family, project category, and bug kind) on the capabilities of fault localization, we introduce two varying effects regression models with normal likelihood and logarithmic link function.

$$\begin{bmatrix} E_b \\ T_b \end{bmatrix} \sim \mathsf{MVNormal}\left(\begin{bmatrix} e_b \\ t_b \end{bmatrix}, S\right) \quad \log(e_b) = \alpha + \alpha_{family[b]} + \alpha_{category[b]} \quad \log(t_b) = \beta + \beta_{family[b]} + \beta_{category[b]} \tag{5.9}$$

$$E_b \sim \mathsf{Normal}(e_b,\ \sigma) \qquad \log(e_b) = \begin{pmatrix} \alpha & + \alpha_{family[b]} + \alpha_{category[b]} \\ & + c_{family[b]}\, crashing_b \\ & + p_{family[b]}\, predicate_b \\ & + m_{family[b]}\, \log(1 + mutability_b) \end{pmatrix} \tag{5.10}$$

Model (5.9) is multivariate, as it simultaneously captures effectiveness and runtime cost of fault localization. For each fault localization experiment on a bug b, (5.9) expresses the vector $[E_b, T_b]$ of standardized[13] E_{inspect} metric E_b and running time T_b as drawn from a multivariate normal distribution whose means e_b and t_b are log-linear functions of various *predictors*. Namely, $\log(e_b)$ is the sum of a base intercept α; a family-specific intercept $\alpha_{family[b]}$, for each fault-localization family SBFL, MBFL, PS, and ST; and a category-specific intercept $\alpha_{category[b]}$, for each project category CL, DEV, DS, and WEB. The other model component $\log(t_b)$ follows the same log-linear relation.

Model (5.10) is univariate, since it only captures the relation between bug kinds and effectiveness. For each fault localization experiment on a bug b, (5.10) expresses the standardized E_{inspect} metric E_b as drawn from a normal distribution whose mean e_b is a log-linear function of a base intercept α; a family-specific intercept $\alpha_{family[b]}$; and a category-specific intercept $\alpha_{category[b]}$; a varying intercept $c_{family[b]}crashing_b$, for the interactions between each family and crashing bugs; a varying intercept $p_{family[b]}predicate_b$, for the interactions between each family and predicate bugs; and a

[12] The practical usefulness of statistical hypothesis tests has been seriously questioned in recent years [10, 41, 126]; therefore, we mainly report this statistics for conformance with standard practices, but we refrain from giving it any serious weight as empirical evidence.

[13] We standardize the data since this simplifies fitting the model; for the same reason, we also log-transform the running time in seconds.

varying slope $m_{family[b]} \log(1 + mutability_b)$, for the interactions between each family and bugs with different mutability.[14] Variables *crashing* and *predicate* are indicator variables, which are equal to 1 respectively for crashing or predicate-related bugs, and 0 otherwise; variable *mutability* is instead the mutability percentage defined in Section 5.6.

Completing regression models (5.9) and (5.10) with suitable priors and fitting them on our experimental data[15] gives a (sampled) distribution of values for the coefficients α's, c, p, m, and β's, which we can analyze to infer the effects of the various predictors on the outcome. For example, if the 95% probability interval of α_F's distribution lies entirely below zero, it suggests that FL family F is consistently associated with below-average values of E_{inspect} metric $\mathscr{I}$; in other words, F tends to be more effective than techniques in other families. As another example, if the 95% probability interval of β_C's distribution includes zero, it suggests that bugs in projects of category C are not consistently associated with different-than-average running times; in other words, bugs in these projects do not seem either faster or slower to analyze than those in other projects.

5.8 Experimental Methodology

To answer Section 5.3's research questions, we ran FAUXPY using each of the 7 fault localization techniques described in Section 2.3 on all 135 selected bugs (described in Section 5.4) from BUGSINPY v. b4bfe91, for a total of $945 = 7 \times 135$ FL experiments. Henceforth, the term "*standalone* techniques" refers to the 7 classic FL techniques described in Section 5.2; whereas "*combined* techniques" refers to the four techniques introduced for RQ4.

Test selection. The test suites of projects such as keras (included in BUGSINPY) are very large and can take more than 24 hours to run even once. Without a suitable test selection strategy, large-scale FL experiments would be prohibitively time consuming (especially for MBFL techniques, which rerun the same test suite hundreds of times). Therefore, we applied a simple test selection strategy to only include tests that directly target the parts of a program that contribute to the failures.[16]

As we mentioned in Section 5.4, each bug b in BUGSINPY comes with a selection of failing tests F_b and passing tests P_b. The failing tests are usually just a few, and specifically trigger bug b. The passing tests, in contrast, are much more numerous, as they usually include all non-failing tests available in the project. In order to cull the number of passing tests to only include those that expressly target the failing code, we applied a simple dependency analysis: for each BUGSINPY bug b used in our experiments, we built the module-level call graph $G(b)$ for the whole of b's project;[17] each node in $G(b)$ is a module of the project (including its tests), and each edge $x_m \rightarrow y_m$ means that module x_m directly uses some entities defined in module y_m. Consider any of b's project *test module* t_m; we run the tests in t_m in our experiments if and only if: *i)* t_m includes at least one of the *failing* tests in F_b; *ii) or*, $G(b)$ includes an edge $t_m \rightarrow f_m$, where f_m is a module that includes at least one of b's faulty locations $\mathscr{F}(b)$ (see Section 5.5). In other words: we include *all failing* tests for b, as well as the passing tests that directly exercise the parts of the project that are faulty. This simple heuristics substantially reduced the number of tests that we had to run for the largest projects, without meaningfully affecting the fault localization's scope.

[14] We log-transform *mutability* in this term, since this smooths out the big differences between mutability scores in different experiments (in particular, between zero and non-zero), which simplifies modeling the relation statistically. We add one to *mutability* before log-transforming it, so that the logarithm is always defined.

[15] The replication package includes all details about the regression models, as well as their validation [42].

[16] The Defects4J curated collection also includes a selection of so-called *relevant* tests [64].

[17] To build the call graph we used Python static analysis framework Scalpel [71], which in turn relies on PyCG [108] for this task.

Our test selection strategy does not include test modules that *indirectly* involve failing locations (unless they include any *failing* tests): if the tests in a module t_m only call directly an application module x_m, and then some parts of module x_m call another application module y_m (i.e., $t_m \rightarrow x_m \rightarrow y_m$ in the module-level call graph), x_m does not include any faulty locations, and y_m does include some faulty locations, then we do *not* include the tests in t_m in our test suite; instead, we will include *other* test modules u_m that directly call y_m (i.e., $u_m \rightarrow y_m$).

To demonstrate that our more aggressive test selection strategy does not exclude any relevant tests, and is unlikely to affect the quantitative fault localization results, we first computed, for each bug b used in our experiments: *i*) the set S_b^0 of tests selected using the strategy described above; and *ii*) the set $S_b^+ \supseteq S_b^0$ of tests selected by including also *indirect* dependencies (i.e., by taking the transitive closure of the module-level use relation). For 48% of the 135 bugs used in our experiments, $S_b^+ = S_b^0$, that is both test selection strategies select the same tests. However, there remain a long tail of bugs for which including indirect dependencies leads to many more tests being selected; for example, for 40 bugs in 7 projects, considering indirect dependencies leads to selecting more than 50 additional tests—which would significantly increase the experiments' running time. Thus, we randomly selected one bug for each project among those 40 bugs for which indirect dependencies would lead to including more than 50 additional tests. For each bug b in this sample, we performed an additional run of our fault localization experiments with SBFL and MBFL techniques[18] using all tests in S_b^+, for a total of 35 new experiments. We found that none of the key fault localization effectiveness metrics significantly changed compared to the same experiments using only tests in S_b^0.[19] This confirms that our test selection strategy does not alter the general effectiveness of fault localization, and hence we adopted it for the rest of the experiments.

Table 5.4 shows statistics about the fraction of tests that we selected for our experiments according to the test selection strategy. Those data indicate that test selection has a disproportionate impact on projects that have very large test suites, such as those in the DS category. In these projects, it happens often that the vast majority of tests are irrelevant for the portion of the project where a failure occurred; therefore, excluding these tests from our experiments is instrumental in drastically bringing down execution times without sacrificing experimental accuracy.

Experimental setup. Each experiment ran on a node of USI's HPC cluster,[20] each equipped with 20-core Intel Xeon E5-2650 processor and 64 GB of DDR4 RAM, accessing a shared 15 TB RAID 10 SAS3 drive, and running CentOS 8.2.2004.x86_64. We provisioned three CPython Virtualenvs with Python v. 3.6, 3.7, and 3.8; our scripts chose a version according to the requirements of each BUGSINPY subject. The experiments took more than two CPU-months to complete—not counting the additional time to setup the infrastructure, fix the execution scripts, and repeat any experiments that failed due to incorrect configuration.

This chapter's detailed replication package includes all scripts used to ran these experiments, as well as all raw data that we collected by running them. The rest of this section details how we analyzed and summarized the data to answer the various research questions.[21]

[18] Since PS and ST only use failing tests, their behavior does not change as S_b^0 always includes the same failing tests as S_b^+.

[19] Precisely, in 20 of these 35 experiments the E_{inspect} score did not change at all. As for the remaining experiments, the E_{inspect} score changed but only for bugs that were not effectively localized: the bugs localized in the top-1, top-3, top-5, and top-10 positions did not change, except for a single bug whose $\mathscr{I}_b$(Metallaxis) went from 13 to 9 when we added the extra tests.

[20] Managed by USI's Institute of Computational Science (`https://intranet.ics.usi.ch/HPC`).

[21] Research questions RQ1, RQ2, RQ3, RQ4, and RQ6 only consider *statement-level* granularity; in contrast, RQ5 considers all granularities (see Section 5.2.5).

CATEGORY	PROJECT	MIN				MEDIAN				MEAN				MAX			
		C		*P*		*C*		*P*		*C*		*P*		*C*		*P*	
		#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
CL	httpie			1	0.8			7.0	4.7			8.0	7.2			17	100.0
	thefuck	1	0.6	3	0.6	17.0	15.2	5.0	1.7	40.6	17.9	7.4	2.0	126	100.0	18	5.4
	tqdm			1	1.7			63.0	95.2			47.9	82.1			77	100.0
	youtube-dl			15	14.3			89.5	51.0			78.7	43.8			126	55.8
DEV	black			16	88.5			91.0	91.0			83.3	91.2			129	93.5
	cookiecutter	2	0.2	11	5.2	80.0	27.6	44.0	26.3	80.8	19.9	39.8	20.9	198	93.5	60	28.0
	luigi			2	0.2			91.0	11.3			90.8	11.6			198	33.2
DS	keras			18	3.0			58.0	10.5			76.6	13.9			288	51.6
	pandas	1	0.0	1	0.0	67.5	3.2	91.5	0.8	112.8	2.1	159.8	1.4	1036	51.6	1036	8.9
	spaCy			13	1.4			75.0	7.8			80.5	8.6			152	16.9
WEB	fastapi			1	0.3			5.0	1.5			37.6	8.8			282	49.9
	sanic	1	0.3	98	21.2	32.0	4.5	265.0	56.5	139.6	25.7	220.3	47.3	787	76.7	298	64.2
	tornado			32	3.4			411.5	40.5			410.5	41.9			787	76.7
	overall	1	0.0	1	0.0	53.0	8.9	53.0	8.9	86.6	4.6	86.6	4.6	1036	100.0	1036	100.0

Table 5.4. Tests used in the fault localization experiments with the bugs of Table 5.2. Following the procedure described in Section 5.8, we selected s_b tests out of the t_b BUGSINPY tests for each bug b among the 135 bugs used in our experiments. For each PROJECT, the table reports the MINimum, MEDIAN, MEAN, and MAXimum percentage $100 \cdot s_b / t_b$ % of selected tests among bugs b in the project (columns *P*); similarly, columns # report the same statistics the MINimum, MEDIAN, MEAN, and MAXimum number of selected tests s_b among all bug b in the project. Finally, columns *C* report the same statistics among all bugs in projects of the same CATEGORY; and the bottom row reports the **overall** statistics among all 135 bugs.

5.8.1 RQ1. Effectiveness

To answer RQ1 (fault localization *effectiveness*), we report the $L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$ counts, the average generalized E_{inspect} rank $\widetilde{\mathscr{I}}_B(L)$, the average exam score $\mathscr{E}_B(L)$, and the average location list length $|L_B|$ for each technique L among Section 5.2's seven standalone fault localization *techniques*; as well as the same metrics averaged over each of the four fault localization *families*. These metrics measure the effectiveness of fault localization from different angles. We report these measures for *all* 135 BUGSINPY bugs B selected for our experiments.

To qualitatively summarize the effectiveness comparison between two FL techniques A and B, we consider their counts $A@1\% \le A@3\% \le A@5\% \le A@10\%$ and $B@1\% \le B@3\% \le B@5\% \le B@10\%$ and compare them pairwise: $A@k\%$ vs. $B@k\%$, for the each k among 1, 3, 5, 10. We say that:

$A \gg B$: "*A is much more effective than B*", if $A@k\% > B@k\%$ for all ks, and $A@k\% - B@k\% \ge 10$ for at least three ks out of four;

$A > B$: "*A is more effective than B*", if $A@k\% > B@k\%$ for all ks, and $A@k\% - B@k\% \ge 5$ for at least one k out of four;

$A \ge B$: "*A tends to be more effective than B*", if $A@k\% \ge B@k\%$ for all ks, and $A@k\% > B@k\%$ for at least three ks out of four;

$A \simeq B$: "*A is about as effective as B*", if none of $A \gg B$, $A > B$, $A \ge B$, $B \gg A$, $B > A$, and $B \ge A$ holds.

To *visually compare* the effectiveness of different FL families, we use *scatterplots*—one for each pair F_1, F_2 of families. The scatterplot comparing F_1 to F_2 displays one point at coordinates (x, y) for each bug b analyzed in our experiments. Coordinate $x = \widetilde{\mathscr{I}}_b(F_1)$, that is the average generalized E_{inspect} rank that techniques in family F_1 achieved on b; similarly, $y = \widetilde{\mathscr{I}}_b(F_2)$, that is the average generalized E_{inspect} rank that techniques in family F_2 achieved on b. Thus, points lying below the diagonal line $x = y$ (such that $x > y$) correspond to bugs for which family F_2 performed *better* (remember that a lower E_{inspect} score means more effective fault localization) than family F_1; the opposite holds for points lying above the diagonal line. The location of points in the scatterplot relative to the diagonal gives a clear idea of which family performed better in most cases.

To *analytically compare* the effectiveness of different FL families, we report the estimates and the 95% probability intervals of the coefficients α_F in the fitted regression model (5.9), for each FL family F. If the interval of values lies entirely below zero, it means that family F's effectiveness tends to be *better* than the other families on average; if it lies entirely above zero, it means that family F's effectiveness tends to be *worse* than the other families; and if it includes zero, it means that there is no consistent association (with above- or below-average effectiveness).

5.8.2 RQ2. Efficiency

To answer RQ2 (fault localization *efficiency*), we report the average wall-clock running time $T_B(L)$, for each technique L among Section 5.2's seven standalone fault localization *techniques*, on bugs in B; as well as the same metric averaged over each of the four fault localization *families*. This basic metric measures how long the various FL techniques take to perform their analysis. We report these measures for *all* 135 BUGSINPY bugs B selected for our experiments.

To qualitatively summarize the efficiency comparison between two FL techniques A and B, we compare pairwise their average running times $T(A)$ and $T(B)$, and say that:

$A \gg B$: "*A is much more efficient than B*", if $T(A) > 10 \cdot T(B)$;

$A > B$: "*A is more efficient than B*", if $T(A) > 1.1 \cdot T(B)$;

$A \simeq B$: "*A is about as efficient as B*", if none of $A \gg B$, $A > B$, $B \gg A$, and $B > A$ holds.

To *visually compare* the efficiency of different FL families, we use *scatterplots*—one for each pair F_1, F_2 of families. The scatterplot comparing F_1 to F_2 displays one point at coordinates (x, y) for each bug b analyzed in our experiments. Coordinate $x = T_b(F_1)$, that is the average running time of techniques in family F_1 on b; similarly, $y = T_b(F_2)$, that is the average running time of techniques in family F_2 on b. The interpretation of these scatterplots is as those considered for RQ1.

To *analytically compare* the efficiency of different FL families, we report the estimates and the 95% probability intervals of the coefficients β_F in the fitted regression model (5.9), for each FL family F. The interpretation of the regression coefficients' intervals is similar to those considered for RQ1: β_F's lies entirely above zero when F tends to be *slower* (less efficient) than other families; it lies entirely below zero when F tends to be *faster*; and it includes zero when there is no consistent association with above- or below-average efficiency.

5.8.3 RQ3. Kinds of Faults and Projects

To answer RQ3 (fault localization behavior for different *kinds of faults* and *projects*), we report the same effectiveness metrics considered in RQ1 ($F@_X1\%$, $F@_X3\%$, $F@_X5\%$, and $F@_X10\%$ percentages, average generalized E_{inspect} ranks $\widetilde{\mathscr{I}}_X(F)$, average exam scores $\mathscr{E}_X(F)$, and average location list length $|F_X|$), as well as the same efficiency metrics considered in RQ2 (average wall-clock running

time $T_X(F)$) for each standalone fault localization family F and separately for *i*) bugs X of different *kinds*: crashing bugs, predicate bugs, and mutable bugs (see Figure 5.4); *ii*) bugs X from projects of different *category*: CL, DEV, DS, and WEB (see Section 5.6).

To *visually compare* the effectiveness and efficiency of fault localization families on bugs from projects of different *category*, we color the points in the scatterplots used to answer RQ1 and RQ2 according to the bug's project category.

To *analytically compare* the effectiveness of different FL families on bugs of different *kinds*, we report the estimates and the 95% probability intervals of the coefficients c_F, p_F, and m_F in the fitted regression model (5.10), for each FL family F. The interpretation of the regression coefficients' intervals is similar to those considered for RQ1 and RQ2: c_F, p_F, and m_F characterize the effectiveness of family F respectively on crashing, predicate, and mutable bugs, *relative* to the average effectiveness of the *same family* F on other kinds of bugs.

Finally, to understand whether bugs from projects of certain categories are intrinsically harder or easier to localize, we report the estimates and the 95% probability intervals of the coefficients α_C and β_C in the fitted regression model (5.9), for each project category C. The interpretation of these regression coefficients' intervals is like those considered for RQ1 and RQ2; for example if α_C's interval is entirely below zero, it means that bugs of projects in category C are easier to localize (higher effectiveness) than the average of bugs in any project. This sets a baseline useful to interpret the other data that answer RQ3.

5.8.4 RQ4. Combining Techniques

To answer RQ4 (the effectiveness of *combining* FL techniques), we consider two additional fault localization techniques: CombineFL and AvgFL—both combining the information collected by some of Section 5.2's standalone techniques from different families.

CombineFL was introduced by Zou et al. [145]; it uses a learning-to-rank model to learn how to combine lists of ranked locations given by different FL techniques. After fitting the model on labeled training data,[22] one can use it like any other fault localization technique as follows: *i*) Run any combination of techniques $L_1, \ldots, L_n$ on a bug b; *ii*) Feed the ranked location lists output by each technique into the fitted learning-to-rank model; *iii*) The model's output is a list $\ell_1, \ell_2, \ldots$ of locations, which is taken as the FL output of technique CombineFL. We used Zou et al. [145]'s replication package to run CombineFL on the Python bugs that we analyzed using FAUXPY.

To see whether a simpler combination algorithm can still be effective, we introduced the combined FL technique AvgFL, which works as follows: *i*) Each basic technique L_k returns a list $\langle \ell_1^k, s_1^k \rangle \ldots \langle \ell_{n_k}^k, s_{n_k}^k \rangle$ of locations with *normalized*[23] suspiciousness scores $0 \leq s_j^k \leq 1$; *ii*) AvgFL assigns to location ℓ_x the weighted average $\sum_k w_k s_x^k$, where k ranges over all of FL techniques supported by FAUXPY but Tarantula, and w_k is an integer weight that depends on the FL family of k: 3 for SBFL, 2 for MBFL, and 1 for PS and ST;[24] *iii*) The list of locations ranked by their weighted average suspiciousness is taken as the FL output of technique AvgFL.

Finally, we answer RQ4 by reporting the same effectiveness metrics considered in RQ1 (the $L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$ counts, the average generalized E_{inspect} rank $\widetilde{\mathscr{I}}_B(L)$, the average exam score $\mathscr{E}_B(L)$, and the average location list length $|L_B|$) for techniques CombineFL and AvgFL. Precisely, we consider two variants A and S of CombineFL and of AvgFL, giving a total of four *combined* fault localization techniques: variants A (CombineFL$_A$ and AvgFL$_A$) use the output of all

[22] Since the training time is negligible, we ignore it in all measures of running time—consistently with Zou et al. [145].

[23] We used min-max normalization, also known as feature scaling [56].

[24] These weights roughly reflect the relative effectiveness and applicability of FL techniques suggested by our experimental results.

FL techniques supported by FAUXPY but Tarantula—which was not considered in [145]; variants S (CombineFL$_S$ and AvgFL$_S$) only use the Ochiai, DStar, and ST FL techniques (excluding the more time-consuming MBFL and PS families).

5.8.5 RQ5. Granularity

To answer RQ5 (how fault localization effectiveness changes with granularity), we report the same effectiveness metrics considered in RQ1 (the $L@_B1$, $L@_B3$, $L@_B5$, and $L@_B10$ counts, the average generalized E_{inspect} rank $\widetilde{\mathscr{I}}_B(L)$, the average exam score $\mathscr{E}_B(L)$, and the average location list length $|L_B|$) for all seven standalone techniques, and for all four combined techniques, but targeting *functions* and *modules* as suspicious entities. Similar to Zou et al. [145], for function-level and module-level granularities, we define the suspiciousness score of an entity as the maximum suspiciousness score computed for the statements in them.

5.8.6 RQ6. Comparison to Java

To answer RQ6 (comparison between Python and Java), we quantitatively and qualitatively compare the main findings of Zou et al. [145]—whose empirical study of fault localization in Java was the basis for our Python replication—against our findings for Python.

For the *quantitative* comparison of *effectiveness*, we consider the metrics that are available in both studies: the percentage of all bugs each technique localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@1\%$, $L@3\%$, $L@5\%$, and $L@10\%$); and the average exam score. For Python, we consider all 135 BUGSINPY bugs we selected for our experiments; the data for Java is about Zou et al.'s experiments on 357 bugs in Defects4J [63]. We consider all standalone techniques that feature in both studies: Ochiai and DStar (SBFL), Metallaxis and Muse (MBFL), predicate switching (PS), and stack-trace fault localization (ST).

We also consider the combined techniques CombineFL$_A$ and CombineFL$_S$. The original idea of the CombineFL technique was introduced by Zou et al.; however, the variants used in their experiments combine all eleven FL techniques they consider, some of which we did not include in our replication (see Section 5.1 for details). Therefore, we modified [145]'s replication package to extract from their Java experimental data the rankings obtained by CombineFL$_A$ and CombineFL$_S$ combining the same techniques as in Python (see Section 5.8.4). This way, the quantitative comparison between Python and Java involves exactly the same techniques and combinations thereof.

Since we did not re-run Zou et al.'s experiments on the same machines used for our experiments, we cannot compare efficiency quantitatively. Anyway, a comparison of this kind between Java and Python would be outside the scope of our studies, since any difference would likely merely reflect the different performance of Java and Python—largely independent of fault localization efficiency.

For the *qualitative* comparison between Java and Python, we consider the union of all findings presented in Section 5.9 or in Zou et al. [145]; we discard all findings from one study that are outside the scope of the other study (for example, Java findings about history-based fault localization, a standalone technique that we did not implement for Python; or Python findings about AvgFL, a combined technique that Zou et al. did not implement for Java); for each within-scope finding, we determine whether it is confirmed ✔ (there is evidence corroborating it) or refuted ✖ (there is evidence against it) for Python and for Java.

FAMILY	TECHNIQUE L	$\widetilde{\mathscr{F}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathscr{E}_B(L)$		$\lvert L_B\rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
MBFL	Metallaxis	6 710	6 706	8	10	22	25	27	30	34	37	0.0029	0.0035	113.9	113.9
	Muse		6 714		6		19		25		32		0.0023		113.9
PS		11 945	11 945	3	3	5	5	7	7	7	7	0.0001	0.0001	1.0	1.0
SBFL	DStar	1 584	1 583	12	11	30	30	43	42	54	54	0.0042	0.0042	2 521.3	2 521.3
	Ochiai		1 583		12		30		43		54		0.0042		2 521.3
	Tarantula		1 586		12		30		43		54		0.0042		2 521.3
ST		9 810	9 810	0	0	4	4	6	6	13	13	0.0024	0.0024	42.9	42.9

Table 5.5. Effectiveness of standalone fault localization techniques at the statement-level granularity on all 135 selected bugs B. Each row reports a TECHNIQUE L's average generalized E_{inspect} rank $\widetilde{\mathscr{F}}_B(L)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$); its average exam score $\mathscr{E}_B(L)$; and its average suspicious locations length $\lvert L_B\rvert$. Columns F report the same metrics averaged for all techniques that belong to the same FAMILY. Highlighted numbers denote the best technique according to each metric.

5.9 Experimental Results

This section summarizes the experimental results that answer the research questions detailed in Section 5.8. All results except for Section 5.9.5's refer to experiments with statement-level granularity; results in Sections 5.9.1–5.9.3 only consider standalone techniques. To keep the discussion focused, we mostly comment on the @n% metrics of effectiveness, whereas we only touch upon the exam score, E_{inspect}, and location list length when they complement other results.

5.9.1 RQ1. Effectiveness

Family effectiveness. Among standalone techniques, the SBFL fault localization family achieves the best effectiveness according to several metrics. Table 5.5 shows that all SBFL techniques have better average E_{inspect} rank $\widetilde{\mathscr{F}}$; and higher percentages of faulty locations in the top-1, top-3, top-5, and top-10. The advantage over MBFL—the second most-effective family—is consistent and conspicuous. According to the same metrics, the MBFL fault localization family achieves clearly better effectiveness than PS and ST. Then, PS tends to do better than ST, but only according to some metrics: PS has better @1%, @3%, and @5%, and location list length, whereas ST has better E_{inspect} and @10%.

Finding 1.1: SBFL is the most effective standalone fault localization family.

Finding 1.2: Standalone fault localization families ordered by effectiveness: SBFL $>$ MBFL $\gg$ PS $\simeq$ ST, where $>$ means better, $\gg$ much better, and $\simeq$ about as good.

Contrary to these general trends, PS achieves the best (lowest) exam score and location list length of all families; and ST is second-best according to these metrics. As Section 5.9.3 will discuss in more detail, PS and ST are techniques with a narrower scope than SBFL and MBFL: they can perform very well on a subset of bugs, but they fail spectacularly on several others. They also tend to return shorter lists of suspicious locations, which is also conducive to achieving a better exam score: since the exam score is undefined when a technique fails to localize a bug at all (as explained in Section 5.7.2), the average exam score of ST and, especially, PS is computed over the small set of bugs on which they work fairly well.

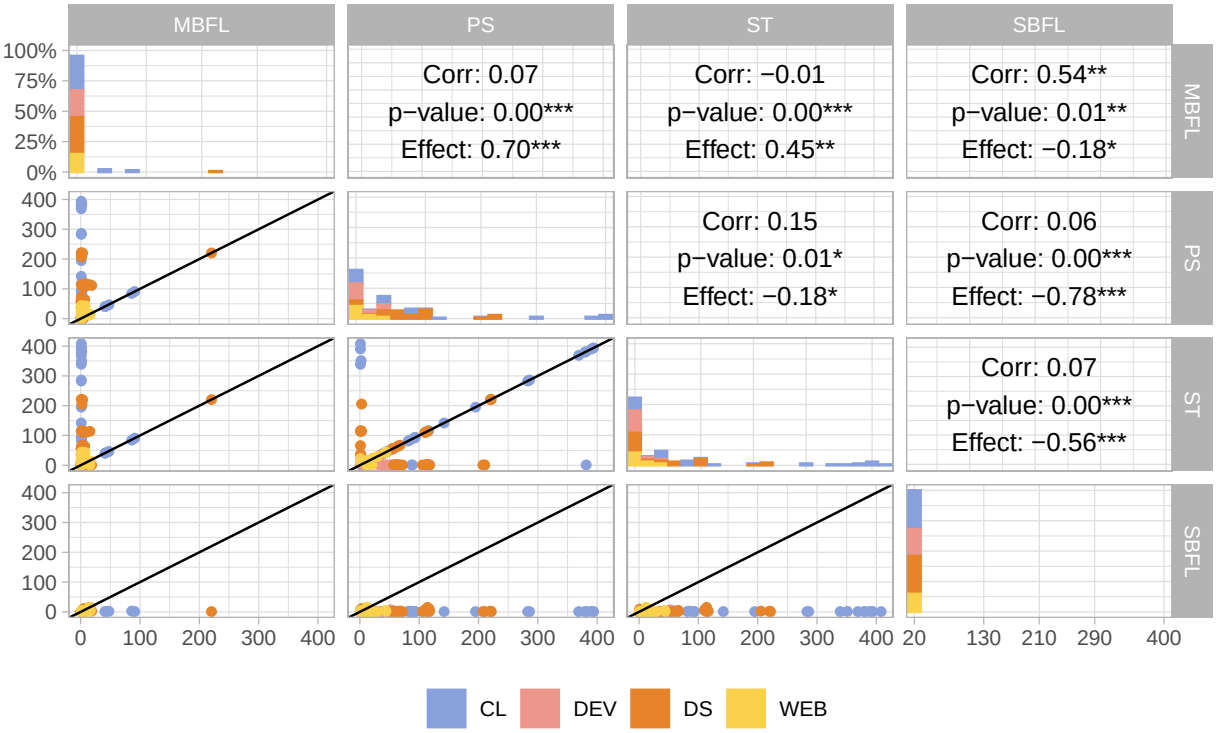


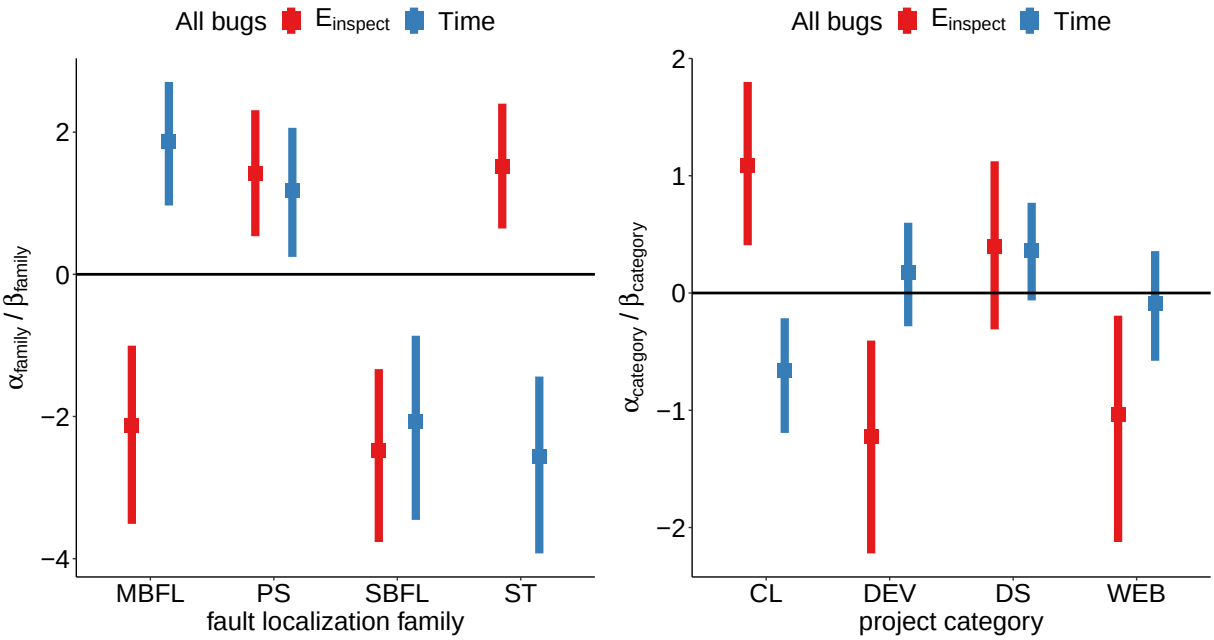
Figure 5.6. Pairwise visual comparison of four FL families for effectiveness. Each point in the scatterplot at row labeled R and column labeled C has coordinates (x, y), where x is the generalized E_{inspect} rank $\widetilde{\mathscr{I}}_b(C)$ of FL techniques in family C and y is the rank $\widetilde{\mathscr{I}}_b(R)$ of FL techniques in family R on the same bug b. Thus, points below (resp. above) the diagonal line denote bugs on which R had better (resp. worse) E_{inspect} ranks. Points are colored according to the bug's project category. The opposite box at row labeled C and column labeled R displays three statistics (correlation, p-value, and effect size, see Section 5.7.3) quantitatively comparing the same average generalized E_{inspect} ranks of C and R; negative values of effect size mean that R tends to be better, and positive values mean that C tends to be better. Each bar plot on the diagonal at row F, column F is a histogram of the distribution of $\widetilde{\mathscr{I}}_b(F)$ for all bugs. Horizontal axes of all diagonal plots have the same E_{inspect} scale as the bottom-right plot's (SBFL); their vertical axes have the same 0–100% scale as the top-left plot (MBFL).

Finding 1.3: PS and ST are specialized fault localization techniques, which may work well only on a small subset of bugs, and thus often return short lists of suspicious locations.

Figure 5.6's scatterplots confirm SBFL's general advantage: in each scatterplot involving SBFL, all points are on a straight line corresponding to low ranks for SBFL but increasingly high ranks for the other family. The plots also indicate that MBFL is often better than PS and ST, although there are a few hard bugs for which the latter are just as effective (points on the diagonal line). The PS-vs-ST scatterplot suggests that these two techniques are largely complementary: on several bugs, PS and ST are as effective (points on the diagonal); on several others, PS is more effective (points above the diagonal); and on others still, ST is more effective (points below the diagonal).

Figure 5.7a confirms these results based on the statistical model (5.9): the intervals of coefficients α_{SBFL} and α_{MBFL} are clearly below zero, indicating that SBFL and MBFL have better-than-average effectiveness; conversely, those of coefficients α_{PS} and α_{ST} are clearly above zero, indicating that PS and ST have worse-than-average effectiveness.

Figure 5.7a's estimate of α_{SBFL} is below that of α_{MBFL}, confirming that SBFL is the most effective



(a) Estimates and 95% probability intervals for the coefficients α_{family} and β_{family} in model (5.9) fitted on all bugs, for each FL family MBFL, PS, SBFL, and ST.

(b) Estimates and 95% probability intervals for the coefficients $\alpha_{category}$ and $\beta_{category}$ in model (5.9) fitted on all bugs, for each project category CL, DEV, DS, and WEB.

Figure 5.7. Point estimates (boxes) and 95% probability intervals (lines) for the regression coefficients of model (5.9). The scale of the vertical axes is over standard deviation log-units.

family overall. The bottom-left plot in Figure 5.6 confirms that SBFL's advantage can be conspicuous but is observed only on a minority of bugs—whereas SBFL and MBFL achieve similar effectiveness on the majority of bugs. In fact, the effect size comparing SBFL and MBFL is −0.18—weakly in favor of SBFL.

Finding 1.4: SBFL and MBFL often achieve similar effectiveness; however, SBFL is strictly better than MBFL on a minority of bugs.

Technique effectiveness. FL techniques of the same family achieve very similar effectiveness. Table 5.5 shows nearly identical results for the 3 SBFL techniques Tarantula, Ochiai, and DStar. The plots and statistics in Figure 5.8 confirm this: points lie along the diagonal lines in the scatterplots, and E_{inspect} ranks for the same bugs are strongly correlated and differ by a vanishing effect size.

Finding 1.5: All techniques in the SBFL family achieve very similar effectiveness.

The 2 MBFL techniques also behave similarly, but not quite as closely as the SBFL ones. Metallaxis has a not huge but consistent advantage over Muse according to Table 5.5. Figure 5.9 corroborates this observation: the cloud of points in the scatterplot is centered slightly above the diagonal line; the correlation between Muse's and Metallaxis's data is medium (not strong); and the effect size suggests that Metallaxis is more effective on around 11% of subjects.

Muse's lower effectiveness can be traced back to its stricter definition of "mutant killing", which requires that a failing test becomes passing when run on a mutant (see Section 5.2.2). As observed elsewhere [94], this requirement may be too demanding for fault localization of real-world bugs, where it is essentially tantamount to generating a mutant that is similar to a patch.

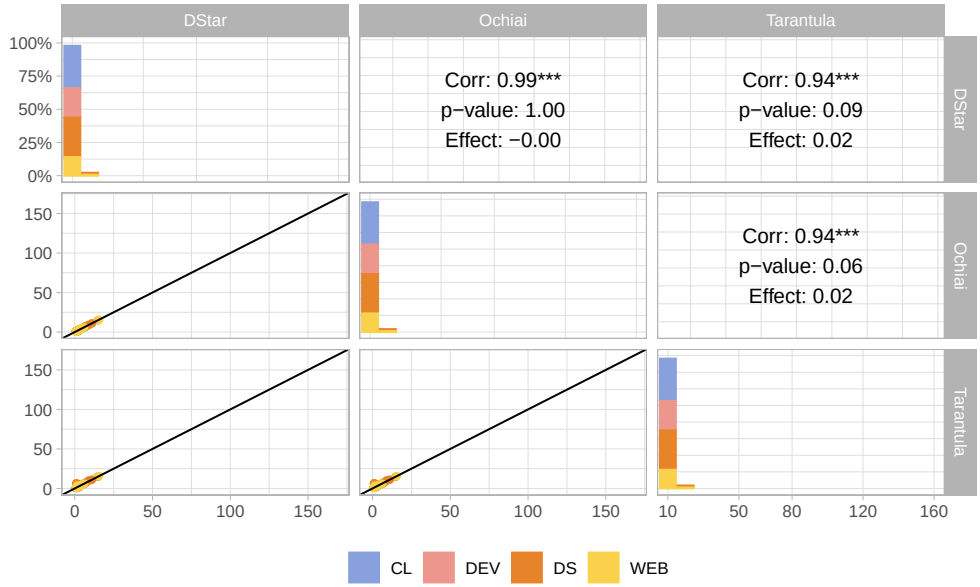


Figure 5.8. Pairwise visual comparison of 3 SBFL techniques for effectiveness. The interpretation of the plots is the same as in Figure 5.6.

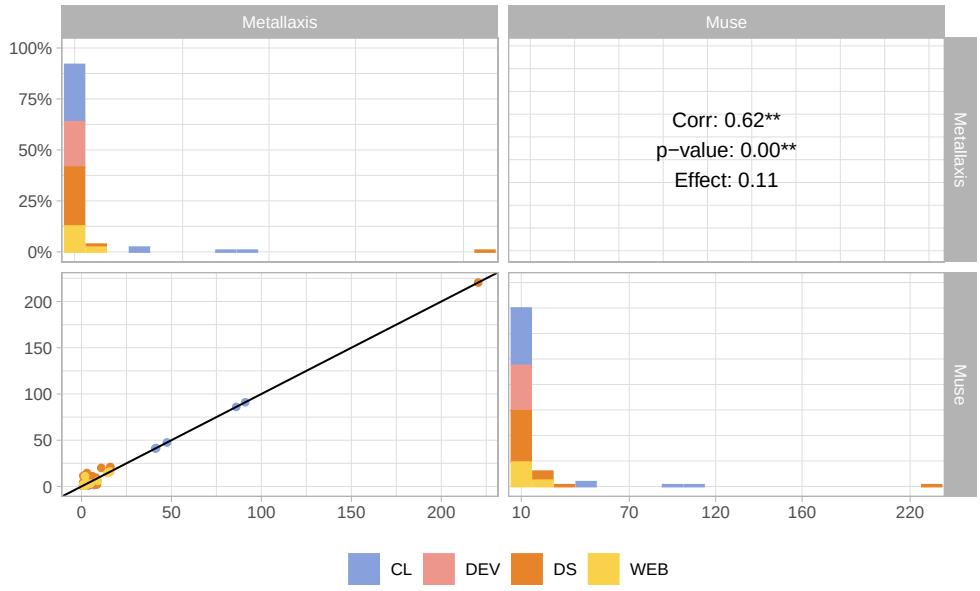


Figure 5.9. Pairwise visual comparison of 2 MBFL techniques for effectiveness. The interpretation of the plots is the same as in Figure 5.6.

Finding 1.6: The techniques in the MBFL family achieve generally similar effectiveness, but Metallaxis tends to be better than Muse.

5.9.2 RQ2. Efficiency

As demonstrated in Table 5.6, the four FL families differ greatly in their efficiency—measured as their wall-clock running time. ST is by far the fastest, taking a mere 2 seconds per bug on average; SBFL is second-fastest, taking around 10 *minutes* on average; PS is one order of magnitude slower,

FAMILY	TECHNIQUE L	ALL	CRASHING	PREDICATE	MUTABLE	CL	DEV	DS	WEB
MBFL	Metallaxis Muse	15 774	18 278	19 671	17 744	3 770	18 694	29 799	7 753
PS		9 751	11 419	17 287	12 932	528	20 210	15 972	828
SBFL	DStar Ochiai Tarantula	589	890	1 284	521	30	38	1 726	231
ST		2	2	2	2	2	1	2	1

Table 5.6. Efficiency of fault localization techniques at the statement-level granularity. Each row reports a TECHNIQUE L's per-bug average wall-clock running time $T_X(L)$ in seconds on: ALL 135 bugs selected for the experiments ($X = B$); CRASHING, PREDICATE-related, and MUTABLE bugs; bugs in projects of category CL, DEV, DS, and WEB (see Section 5.6). The running time is the same for all techniques of the same FAMILY. Highlighted numbers denote the fastest technique for bugs in each group.

taking approximately 2.7 *hours* on average; and MBFL is slower still, taking over 4 hours per bug on average.

Finding 2.1: Standalone fault localization families ordered by efficiency: ST $\gg$ SBFL $\gg$ PS $>$ MBFL, where $>$ means faster, and $\gg$ much faster.[a]

[a] As we discuss at the end of Section 5.9.2, these results are largely expected given how the different fault localization techniques work algorithmically.

Figure 5.10's scatterplots confirm that ST outperforms all other techniques, and that SBFL is generally second-fastest. It also shows that MBFL and PS have similar overall performance but can be slower or faster on different bugs: a narrow majority of points lies below the diagonal line in the scatterplot (meaning PS is faster than MBFL), but there are also several points that are on the opposite side of the diagonal—and their effect size (0.34) is medium, lower than all other pairwise effect sizes in the comparison of efficiency.

Finding 2.2: PS is more efficient than MBFL on average; however, the two families tend to be faster or slower on different bugs.

Based on the statistical model (5.9), Figure 5.7a clearly confirms the differences of efficiency: the intervals of coefficients β_{ST} and β_{SBFL} are well below zero, indicating that ST and SBFL are faster than average (with ST the fastest, as its estimated β_{ST} is lower); conversely, the intervals of coefficients β_{MBFL} and β_{PS} are entirely above zero, indicating that MBFL and PS stand out as slower than average compared to the other families.

These major differences in efficiency are unsurprising if one remembers that the various FL families differ in what kind of information they collect for localization. ST only needs the stack-trace information, which only requires to run once the failing tests; SBFL compares the traces of passing and failing runs, which involves running *all* tests once. PS dynamically tries out a large number of different branch changes in a program, each of which runs the failing tests; in our experiments, PS tried 4 588 different "switches" on average for each bug—up to a whopping 101 454 switches for project black's bug #6. MBFL generates hundreds of different mutations of the program under analysis, each of which has to be run against *all* tests; in our experiments, MBFL generated 461 mutants on average for each bug—up to 2 718 mutants for project black's bug #6. After collecting this information, the additional running time to compute suspiciousness scores (using the formulas presented in Section 5.2) is negligible for all techniques—which explains why the running times of

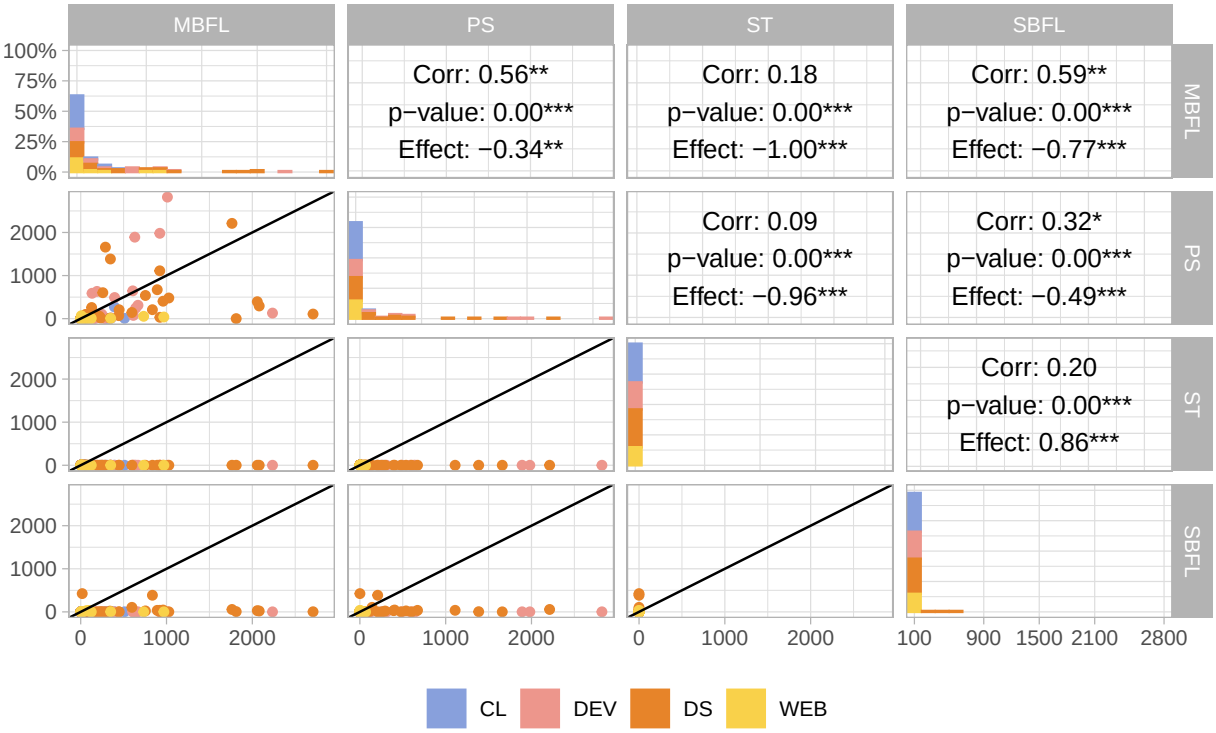


Figure 5.10. Pairwise visual comparison of four FL families for efficiency. Each point in the scatterplot at row labeled R and column labeled C has coordinates (x, y), where x is the average per-bug wall-clock running time of FL techniques in family C and y average per-bug wall-clock running time of FL techniques in family R. Points are colored according to the bug's project category. The opposite box at row labeled C and column labeled R displays three statistics (correlation, p-value, and effect size, see Section 5.7.3) quantitatively comparing the same per-bug average running times of C and R; negative values of effect size mean that R tends to be better, and positive values that C tends to be better.

techniques of the same family are practically indistinguishable.

5.9.3 RQ3. Kinds of Faults and Projects

Project category: effectiveness. Figure 5.7's intervals of coefficients $\alpha_{category}$ in model (5.9) indicate that fault localization tends to be more accurate on projects in categories DEV and WEB, and less accurate on projects in categories CL and DS.

This finding is consistent with the observations that data science programs, their bugs, and their fixes are often different compared to traditional programs [57, 58]. For instance, bug #38 in project keras is an example of what Islam et al. call "structural data flow" bugs [57]: its root cause is passing an incorrect input shape setting to a neural network layer. These characteristics also determine long spectra (i.e., execution traces) that span several functions—which are required to construct the various layer objects; as a result, SBFL techniques struggle to effectively localize this bug. Bugs #68 and #137 in project pandas are instead examples of API bugs, whose root causes are incorrect import statements. While such bugs may occur in any kind of project, they are common in data science programs [57] due to their complex dependencies. In Python, import statements are usually top-level declarations; therefore, FL techniques that can only target locations inside functions end up being ineffective at localizing these API bugs. As yet another example, the overall mutability of bugs in DS projects is 0.7%, whereas it is 1.3% for bugs in other categories of projects. This indicates that

BUGS X	FAMILY F	$\widetilde{\mathscr{F}}_X(F)$	$F@_X1\%$	$F@_X3\%$	$F@_X5\%$	$F@_X10\%$	$\mathscr{E}_X(F)$	$\lvert F_X\rvert$
ALL	MBFL	6 710	8	22	27	34	0.0029	113.9
	PS	11 945	3	5	7	7	0.0001	1.0
	SBFL	1 584	12	30	43	54	0.0042	2 521.3
	ST	9 810	0	4	6	13	0.0024	42.9
CRASHING	MBFL	7 806	7	21	27	34	0.0018	104.4
	PS	15 607	0	0	0	0	–	0.3
	SBFL	897	14	31	43	53	0.0025	3 147.5
	ST	5 273	0	10	16	37	0.0024	118.1
PREDICATE	MBFL	1 891	11	33	40	52	0.0031	146.5
	PS	8 425	8	13	17	17	0.0001	1.3
	SBFL	374	12	23	38	50	0.0065	3 041.5
	ST	9 194	0	2	6	17	0.0007	47.2
MUTABLE	MBFL	489	14	41	50	63	0.0029	138.7
	PS	10 081	5	9	12	12	0.0001	1.1
	SBFL	524	12	35	50	57	0.0042	2 396.2
	ST	9 304	0	4	5	19	0.0007	35.3
CL	MBFL	2 910	9	33	38	45	0.0032	34.3
	PS	8 667	2	5	5	5	0.0002	0.3
	SBFL	2 356	9	42	60	74	0.0056	687.1
	ST	9 124	0	9	9	14	0.0084	19.9
DEV	MBFL	4 720	12	25	28	40	0.0045	160.6
	PS	7 768	3	7	10	10	0.0001	2.1
	SBFL	2 081	20	33	37	47	0.0053	1 431.5
	ST	8 279	0	0	10	13	0.0028	12.4
DS	MBFL	14 519	4	12	19	24	0.0006	169.3
	PS	22 847	2	5	7	7	0.0000	1.0
	SBFL	827	6	23	30	43	0.0018	5 775.4
	ST	15 174	0	0	0	12	0.0003	97.7
WEB	MBFL	1 465	8	18	20	25	0.0042	98.7
	PS	2 362	5	5	5	5	0.0002	1.1
	SBFL	770	15	15	40	45	0.0049	1 266.4
	ST	2 319	0	5	5	15	0.0014	22.7

Table 5.7. Effectiveness of fault localization families at the statement-level granularity on different *kinds* of bugs and *categories* of projects. Each row reports a FAMILY F's average generalized E_{inspect} rank $\widetilde{\mathscr{F}}_X(F)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($F@_X1\%$, $F@_X3\%$, $F@_X5\%$, and $F@_X10\%$); its average exam score $\mathscr{E}_X(F)$ and the length $\lvert F_X\rvert$ of the output list of locations on different groups X of bugs: ALL bugs selected for the experiments (same results as in Table 5.5); bugs of different *kinds* (CRASHING, PREDICATE-related, and MUTABLE bugs); and bugs from projects of different *categories* (CL, DEV, DS, and WEB). Highlighted numbers denote the best family on each group of bugs according to each metric.

the standard mutation operators, used by MBFL, are a poor fit for the kinds of bugs that are most commonly found in data science projects.

Finding 3.1: Bugs in data science projects challenge fault localization's effectiveness (that is, they are harder to localize correctly) more than bugs in other categories of projects.

The data in Table 5.7's bottom section confirm that SBFL remains the most effective FL family, largely independent of the category of projects it analyzes. MBFL ranks second for effectiveness in every project category; it is not that far from SBFL for projects in categories DEV and CL (for example, MBFL and SBFL both localize 9% of CL bugs in the first position; and both localize over 40% of DEV bugs in the top-10 positions). In contrast, SBFL's advantage over MBFL is more conspicuous for projects in categories DS and WEB. Given that bugs in categories CL are generally harder to localize, this suggests that the characteristics of bugs in these projects seem to be a good fit for MBFL. As we have seen in Section 5.9.2, MBFL is the slowest FL family by far; since it reruns the available tests hundreds, or even thousands, of times, projects with a large number of tests are near impossible to analyze efficiently with MBFL. As we'll discuss below, MBFL is considerably faster on projects in category CL than on projects in other categories; this is probably the main reason why MBFL is also more effective on these projects: it simply generates a more manageable number of mutants, which sharpen the dynamic analysis.

Finding 3.2: SBFL remains the most effective standalone fault localization family on all categories of projects.

Figure 5.6's plots confirm some of these trends. In most plots, we see that the points positioned far apart from the diagonal line correspond to projects in the CL and DS categories, confirming that these "harder" bugs exacerbate the different effectiveness of the various FL families.

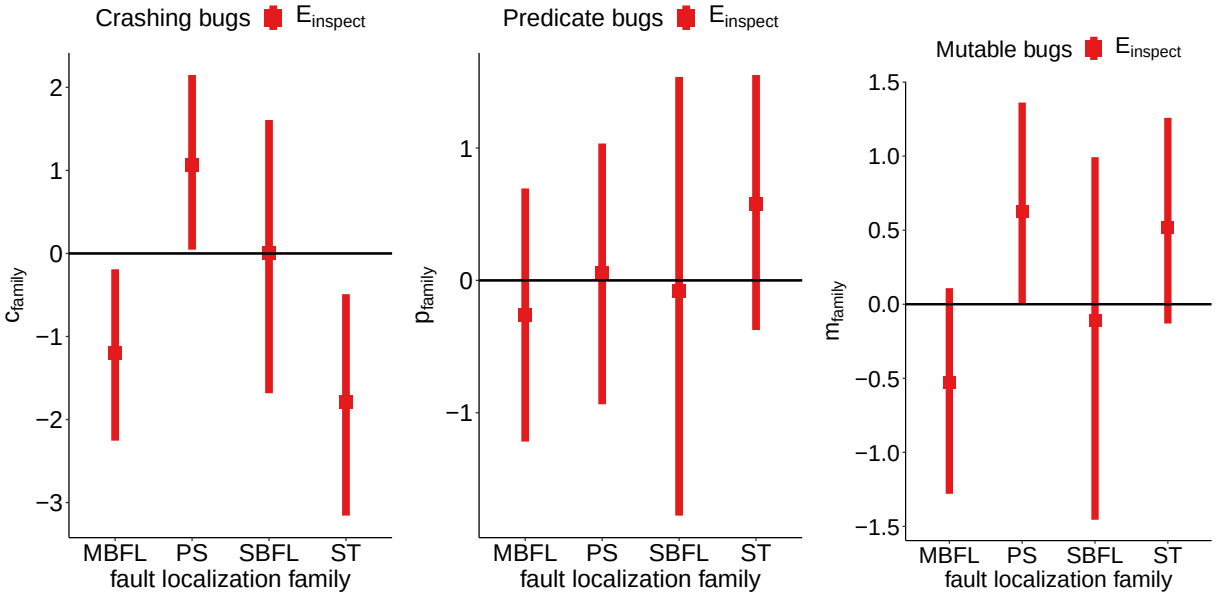
Project category: efficiency. Figure 5.7's intervals of coefficients $\beta_{category}$ in model (5.9) indicate that fault localization tends to be more efficient (i.e., faster) on projects in category CL, and less efficient (i.e., slower) on projects in category DS (β_{DS} barely touches zero). In contrast, projects in categories DEV and WEB do not have a consistent association with faster or slower fault localization. Table 5.2 shows that projects in category DS have the largest number of tests by far (mostly because of outlier project pandas); furthermore, some of their tests involve training and testing different machine learning models, or other kinds of time-consuming tasks. Since FL invariably requires to run tests, this explains why bugs in DS projects tend to take longer to localize.

Finding 3.3: Bugs in data science projects challenge fault localization's efficiency (that is, they take longer to localize) more than bugs in other categories of projects.

The data in Table 5.6's right-hand side generally confirm the same rankings of efficiency among FL families, largely regardless of what category of projects we consider: ST is by far the most efficient, followed by SBFL, and then—at a distance—PS and MBFL. The difference of performance between SBFL and ST is largest for projects in category DS (three orders of magnitude), large for projects in category WEB (two orders of magniture), and more moderate for projects in categories CL and DEV (one order of magnitude). PS is slower than MBFL only for projects in category DEV, although their absolute difference of running times is not very big (around 7.5%); in contrast, it is one order of magnitude faster for projects in categories CL and WEB.

Finding 3.4: The difference in efficiency between MBFL and SBFL is largest for data science projects.

In most of Figure 5.10's plots, we see that the points most frequently positioned far apart from the diagonal line correspond to projects in category DS, confirming that these bugs take longer to analyze and aggravate performance differences among techniques. In the scatterplot comparing MBFL to PS, points corresponding to projects in categories WEB and CL are mostly below the diagonal line, which corroborates the advantage of PS over MBFL for bugs of projects in these two categories.



(a) Estimates and 95% probability intervals for the coefficients c_{family} in model (5.10), for each FL family MBFL, PS, SBFL, and ST.

(b) Estimates and 95% probability intervals for the coefficients p_{family} in model (5.10), for each FL family MBFL, PS, SBFL, and ST.

(c) Estimates and 95% probability intervals for the coefficients m_{family} in model (5.10), for each FL family MBFL, PS, SBFL, and ST.

Figure 5.11. Point estimates (boxes) and 95% probability intervals (lines) for the regression coefficients of model (5.10). The scale of the vertical axes is over standard deviation log-units.

Crashing bugs: effectiveness. According to Figure 5.11a, both FL families ST and MBFL are more effective on *crashing* bugs than on other kinds of bugs. Still, their *absolute* effectiveness on crashing bugs remains limited compared to SBFL's, as shown by the results in Table 5.7's middle part; for example, $@_{\text{CRASHING}}10\%$ is 37% for ST, 34% for MBFL, and 53% for SBFL, whereas ST localizes zero (crashing) bugs in the top rank. Remember that ST assigns that same suspiciousness to all statements within the same function (see Section 5.2.4); thus, it cannot be as accurate as SBFL even on the minority of crashing bugs.

Finding 3.5: ST and MBFL are more effective on crashing bugs than on other kinds of bugs (but they remain overall less effective than SBFL even on crashing bugs).

On the other hand, PS is *less* effective on crashing bugs than on other kinds of bugs; in fact, it localizes zero bugs among the top-10 ranks. PS has a chance to work only if it can find a so-called *critical predicate* (see Section 5.2.3); only three of the crashing bugs included critical predicates, and hence PS was a bust.

Finding 3.6: PS is the least effective on crashing bugs.

Predicate-related bugs: effectiveness. Figure 5.11b says that no FL family achieves consistently better or worse effectiveness on predicate-related bugs. Table 5.7 complements this observation; the ranking of families by effectiveness is different for predicate-related bugs than it is for all bugs: MBFL is about as effective as SBFL, whereas PS is clearly more effective than ST.

Finding 3.7: On predicate-related bugs, MBFL is about as effective as SBFL, and PS is more effective than ST.

This outcome is somewhat unexpected for PS: predicate-related bugs are bugs whose ground truth includes at least a branching predicate (see Section 5.6), and yet PS is still clearly less effective than SBFL or MBFL. Indeed, the presence of a faulty predicate is not sufficient for PS to work: the predicate must also be *critical*, which means that flipping its value turns a failing test into a passing one. When a program has no critical predicates, PS simply returns an empty list of locations. In contrast, when a program has a critical predicate, PS is highly effective: PS@$_\chi$1% = 14%, PS@$_\chi$3% = 24%, and PS@$_\chi$5% = 31% for PS on the 29 bugs χ with a critical predicate—even better than SBFL's results for the same bugs (SBFL@$_\chi$1% = 13%, SBFL@$_\chi$3% = 16%, and SBFL@$_\chi$5% = 20%). In all, PS is a highly specialized FL technique, which works quite well for a narrow category of bugs, but is inapplicable in many other cases.

Finding 3.8: On the few bugs that it can analyze successfully, PS is the most effective standalone fault localization technique.

Mutable bugs: effectiveness. According to Figure 5.11c, FL family MBFL tends to be more effective on *mutable* bugs than on other kinds of bugs: m_{MBFL} 95% probability interval is mostly below zero (and the 87% probability interval would be entirely below zero). Furthermore, Table 5.7 shows that MBFL is the most effective technique on mutable bugs, where it tends to outperform even SBFL. Intuitively, a bug is mutable if the syntactic mutation operators used for MBFL "match" the fault in a way that it affects program behavior. Thus, the capabilities of MBFL ultimately depend on the nature of faults it analyzes and on the selection of mutation operators it employs.

Finding 3.9: MBFL is more effective on mutable bugs than on other kinds of bugs; in fact, it is the most effective standalone fault localization family on these bugs.

Figure 5.11c also suggests that PS and ST are less effective on mutable bugs than on other kinds of bugs. Possibly, this is because mutable bugs tend to be more complex, "semantic" bugs, whereas ST works well only for "simple" crashing bugs, and PS is highly specialized to work on a narrow group of bugs.

Finding 3.10: PS and ST are less effective on mutable bugs than on other kinds of bugs.

Bug kind: efficiency. Table 5.6 does not suggest any consistent changes in the efficiency of FL families when they work on crashing, predicate-related, or mutable bugs—as opposed to all bugs. In other words, for every kind of bugs: ST is orders of magnitude faster than SBFL, which is one order of magnitude faster than PS, which is 14–37% faster than MBFL. As discussed above, the kind of information that a FL technique collects is the main determinant of its overall efficiency; in contrast, different kinds of bugs do not seem to have any significant impact.

Finding 3.11: The relative efficiency of each fault localization family does not depend on the kinds of bugs that are analyzed.

5.9.4 RQ4. Combining Techniques

Effectiveness. Table 5.8 clearly indicates that the combined FL techniques AvgFL and CombineFL achieve high effectiveness—especially according to the fundamental @n% metrics. CombineFL$_A$ and AvgFL$_A$, combining the information from all other FL techniques, beat every other technique. For example, AvgFL$_A$ localizes in the top position 18% of all bugs, CombineFL$_A$ localizes 20% of all bugs, whereas the next-best technique is SBFL, which localizes 12% of all bugs (Table 5.5). CombineFL$_S$

TECHNIQUE L		$\widetilde{\mathscr{I}}_B(L)$	$L@_B1\%$	$L@_B3\%$	$L@_B5\%$	$L@_B10\%$	$\mathscr{E}_B(L)$	$\lvert L_B\rvert$	$T_B(L)$
AvgFL	AvgFL_A	1 575	18	36	47	59	0.0033	2 548.4	26 116
	AvgFL_S	1 585	12	33	44	56	0.0040	2 548.4	591
CombineFL	CombineFL_A	1 580	20	39	49	60	0.0033	2 548.4	26 116
	CombineFL_S	1 584	12	32	41	56	0.0039	2 548.4	591

Table 5.8. Effectiveness and efficiency of fault localization techniques AvgFL and CombineFL at the statement-level granularity on all 135 selected bugs B. Each row reports a TECHNIQUE L's average generalized E_{inspect} rank $\widetilde{\mathscr{I}}_B(L)$; the percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output ($L@_B1\%$, $L@_B3\%$, $L@_B5\%$, and $L@_B10\%$); its average exam score $\mathscr{E}_B(L)$; its average suspicious locations length $|L_B|$; and its average per-bug wall-clock running time $T_B(L)$ in seconds. The four rows correspond to two variants AvgFL_A and CombineFL_A that combine the information of all FL techniques but Tarantula, and two variants AvgFL_S and CombineFL_S that combine the information of SBFL and ST techniques but Tarantula. Highlighted numbers denote the best technique according to each metric.

and AvgFL_S, combining the information from only SBFL and ST techniques, do at least as well as every other standalone technique.

Finding 4.1: Combined fault localization techniques AvgFL_A and CombineFL_A, which combine all baseline techniques, achieve better effectiveness than any other techniques.

While CombineFL_A is strictly more effective than AvgFL_A, their difference is usually modest (at most three percentage points). Similarly, the difference between CombineFL_S, AvgFL_S, and SBFL is generally limited; however, SBFL tends to be less effective than AvgFL_S, whereas CombineFL_S is never strictly more effective than AvgFL_S. In all, AvgFL is a simpler approach to combining techniques than CombineFL, but both are quite successful at boosting FL effectiveness.

Finding 4.2: Fault localization families ordered by effectiveness:
$\text{CombineFL}_A \geq \text{AvgFL}_A > \text{CombineFL}_S \simeq \text{AvgFL}_S > \text{SBFL} > \text{MBFL} \gg \text{PS} \simeq \text{ST}$,
where $>$ means better, $\geq$ better or as good, $\gg$ much better, and $\simeq$ about as good.

The suspicious location length is the very same for AvgFL and CombineFL, and higher than for every other technique. This is simply because all variants of AvgFL and CombineFL consider a location as suspicious if and only if any of the techniques they combine considers it so. Therefore, they end up with long location lists—at least as long as any combined technique's.

Efficiency. The running time of AvgFL and CombineFL is essentially just the sum of running times of the FL families they combine, because merging the output list of locations and training CombineFL's machine learning model take negligible time. This makes AvgFL_A and CombineFL_A the least efficient FL techniques in our experiments; and AvgFL_S and CombineFL_S barely slower than SBFL.

Finding 4.3: Combined fault localization techniques AvgFL_A and CombineFL_A, which combine all baseline techniques, achieve worse efficiency than any other techniques.

Combining these results with those about effectiveness, we conclude that AvgFL_A and CombineFL_A exclusively favor effectiveness; whereas AvgFL_S and CombineFL_S promise a modest improvement in effectiveness in exchange for a modest performance loss.

Finding 4.4: Fault localization families ordered by efficiency:
$\text{ST} \gg \text{SBFL} \geq \text{AvgFL}_S \simeq \text{CombineFL}_S \gg \text{PS} > \text{MBFL} > \text{AvgFL}_A \simeq \text{CombineFL}_A$,
where $>$ means faster, $\geq$ faster or as fast, $\gg$ much faster, and $\simeq$ about as fast.

FAMILY	TECHNIQUE L	$\widetilde{\mathscr{I}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathscr{E}_B(L)$		$\lvert L_B\rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
MBFL	AvgFL_A	66	66	53	53	71	71	77	76	84	84	0.0129	0.0130	296.3	296.3
	CombineFL_A		66		53		70		77		84		0.0128		296.3
	AvgFL_S	67	66	44	44	64	64	73	73	79	79	0.0153	0.0153	296.3	296.3
	CombineFL_S		67		44		64		73		79		0.0154		296.3
	Metallaxis	95	93	31	34	51	56	61	64	67	70	0.0150	0.0135	30.7	30.7
	Muse		97		27		46		57		64		0.0166		30.7
PS		618	618	8	8	13	13	13	13	15	15	0.0025	0.0025	0.6	0.6
SBFL	DStar	67	67	37	37	61	61	72	72	79	79	0.0156	0.0156	296.3	296.3
	Ochiai		67		38		61		72		79		0.0156		296.3
	Tarantula		67		36		61		71		78		0.0156		296.3
ST		451	451	21	21	27	27	27	27	29	29	0.0045	0.0045	1.0	1.0

Table 5.9. Effectiveness of fault localization techniques at the *function*-level granularity on all 135 selected bugs B. The table reports the same metrics as Table 5.5 and Table 5.8 but targeting functions as suspicious entities. Highlighted numbers denote the best technique according to each metric.

5.9.5 RQ5. Granularity

Function-level granularity. Table 5.9's data about function-level effectiveness of the various FL techniques and families lead to very similar high-level conclusions as for statement-level effectiveness: combination techniques CombineFL_A and AvgFL_A achieves the best effectiveness, followed by CombineFL_S and AvgFL_S, then SBFL, and finally MBFL; differences among techniques in the same family are modest (often negligible).

ST is the only technique whose relative effectiveness changes considerably from statement-level to function-level: ST is the least effective at the level of statements, but becomes considerably better than PS at the level of functions. This change is no surprise, as ST is precisely geared towards localizing *functions* responsible for crashes—and cannot distinguish among statements belonging to the same function. ST's overall effectiveness remains limited, since the technique is simple and can only work on crashing bugs.

Module-level granularity. Table 5.10 leads to the same conclusions for module-level granularity: the relative effectiveness of the various techniques is very similar as for statement-level granularity, except that ST gains effectiveness simply because it is designed for coarser granularities.

Finding 5.1: ST is more effective than PS both at the function-level and module-level granularity; however, it remains considerably less effective than other fault localization techniques even at these coarser granularities.

Comparisons between granularities. It is apparent that fault localization's absolute effectiveness strictly *increases* as we target coarser granularities—from statements, to functions, to modules. This happens simply because the number of entities at a coarser granularity is considerably less than the number of entities at a finer granularity: each function consists of several statements, and each module consists of several functions. Therefore, it does not make sense to directly compare the same

FAMILY	TECHNIQUE L	$\widetilde{\mathscr{I}}_B(L)$		$L@_B1\%$		$L@_B3\%$		$L@_B5\%$		$L@_B10\%$		$\mathscr{E}_B(L)$		$\lvert L_B\rvert$	
		F	T	F	T	F	T	F	T	F	T	F	T	F	T
	AvgFL$_A$	2	2	70	70	89	89	93	93	99	99	0.0339	0.0338	20.9	20.9
	CombineFL$_A$		2		70		89		93		99		0.0340		20.9
	AvgFL$_S$	2	2	64	64	87	87	93	93	98	98	0.0362	0.0363	20.9	20.9
	CombineFL$_S$		2		64		87		93		98		0.0362		20.9
MBFL	Metallaxis	6	6	52	57	80	82	86	87	90	92	0.0406	0.0366	5.6	5.6
	Muse		7		47		77		85		87		0.0446		5.6
PS		67	67	13	13	17	17	21	21	28	28	0.0234	0.0234	0.4	0.4
SBFL	DStar	2	2	60	61	86	87	92	93	98	98	0.0369	0.0365	20.9	20.9
	Ochiai		2		61		87		93		98		0.0365		20.9
	Tarantula		2		59		84		91		98		0.0375		20.9
ST		61	61	29	29	33	33	36	36	41	41	0.0284	0.0284	0.6	0.6

Table 5.10. Effectiveness of fault localization techniques at the *module*-level granularity on all 135 selected bugs B. The table reports the same metrics as Table 5.5 and Table 5.8 but targeting modules (files in Python) as suspicious entities. Highlighted numbers denote the best technique according to each metric.

effectiveness metric measured at two different granularity levels, since each granularity level refers to different entities—and inspecting different entities involves incomparable effort.

We do not discuss efficiency (i.e., running time) in relation to granularity: the running time of our fault localization techniques does not depend on the chosen level of granularity, which only affects how the collected information is combined (see Section 5.2.5).

5.9.6 RQ6. Comparison to Java

Table 5.11 collects the main quantitative results for Python fault localization effectiveness that we presented in detail in previous parts of the chapter, and displays them next to the corresponding results for Java. The results are selected so that they can be directly compared: they exclude any technique (e.g., Tarantula) or family (e.g., history-based fault localization) that was not experimented within both our study and Zou et al. [145]; and the rows about CombineFL were computed using [145]'s replication package so that they combine exactly the same techniques (DStar, Ochiai, Metallaxis, Muse, PS, and ST for CombineFL$_A$; and DStar, Ochiai, and ST for CombineFL$_S$).

Then, Table 5.12 lists all claims about fault localization made in our study or in [145] that are within the scope of both studies, and shows which were confirmed or refuted for Python and for Java. Most of the findings (25/28) were confirmed consistently for both Python and Java. Thus, the big picture about the effectiveness and efficiency of fault localization is the same for Python programs and bugs as it is for Java programs and bugs.

There are, however, a few interesting discrepancies; let's discuss possible explanations for them. The most marked difference is about the effectiveness of ST, which was mediocre on Python programs but competitive on Java programs (row 3 in Table 5.12). We think the main reason for these differences is that there were more Java experimental subjects that were an ideal target for ST: 20 out of the 357 Defects4J bugs used in [145]'s experiments consisted of short failing methods whose programmer-written fixes entirely replaced or removed the method body.[25] In these cases,

[25] For example, project Chart's bug #17 in Defects4J v1.0.1.

FAMILY	TECHNIQUE L	L@1%		L@3%		L@5%		L@10%		$\mathscr{E}(L)$	
		Python	*Java*	*Python*	*Java*	*Python*	*Java*	*Python*	*Java*	*Python*	*Java*
CombineFL	CombineFL$_A$	20	19	39	33	49	42	60	52	0.0033	0.0186
	CombineFL$_S$	12	10	32	23	41	30	56	40	0.0039	0.0265
MBFL	Metallaxis	10	6	25	22	30	29	37	36	0.0035	0.1180
	Muse	6	7	19	12	25	16	32	19	0.0023	0.3040
PS		3	1	5	4	7	6	7	6	0.0001	0.3310
SBFL	DStar	11	5	30	24	42	31	54	43	0.0042	0.0330
	Ochiai	12	4	30	23	43	31	54	44	0.0042	0.0330
ST		0	6	4	9	6	11	13	11	0.0024	0.3110

Table 5.11. Effectiveness of fault localization techniques in Python and Java. Each row reports a TECHNIQUE L's percentage of all bugs it localized within the top-1, top-3, top-5, and top-10 positions of its output (L@1%, L@3%, L@5%, and L@10%); and its average exam score $\mathscr{E}(L)$. Python's data corresponds to the experiments discussed in the rest of this chapter on the 135 bugs from BugsInPy; Java's data is taken from Zou et al.'s empirical study [145] or computed from its replication package. Highlighted numbers denote each language's best technique according to each metric.

the ground truth consists of all locations within the method; thus, ST easily ranks the fault location at the top by simply reporting all lines of the crashing method with the same suspiciousness. As a result, Table 5.11 shows that ST was consistently more effective than PS in the Java experiments—whereas there was no consistent difference between ST and PS in our Python experiments. For the same reason, the difference between Java and Python is even more evident on crashing bugs: ST outperformed all other techniques on such bugs in Java but not in Python (row 19 in Table 5.12). We still confirmed that ST works better on crashing bugs than on other kinds of bugs in Python as well, but the nature of our experimental subjects did not allow ST to reach an overall competitive effectiveness on crashing bugs.

Other findings about MBFL were different in Python compared to Java, but the differences were more nuanced in this case. In particular, Zou et al. found that the correlation between the effectiveness of SBFL and MBFL techniques is negligible, whereas we found a medium correlation ($\tau = 0.54$). It is plausible that the discrepancy (reflected in Table 5.12's row 23) is simply a result of several details of how this correlation was measured: we use Kendall's τ, they use the coefficient of determination r^2; we use a generalized E_{inspect} measure $\widetilde{\mathscr{I}}$ that applies to all bugs, they exclude experiments where a technique completely fails to localize the bug ($\mathscr{I}$); we compare the average effectiveness of SBFL vs. MBFL techniques, they pairwise compare individual SBFL and MBFL techniques. Even if the correlation patterns were actually different between Python and Java, this would still have limited practical consequences: MBFL and SBFL techniques still have clearly different characteristics, and hence they remain largely complementary. The same analysis applies to the other correlation discrepancy (reflected in Table 5.12's row 25): in Python, we found a medium correlation between the effectiveness of the Metallaxis and Muse MBFL techniques ($\tau = 0.62$); in Java, Zou et al. found negligible correlation.

Finally, a clarification about the finding that "*On predicate-related bugs, MBFL is about as effective as SBFL*", which Table 5.12 reports as confirmed for both Python and Java. This claim hinges on the definition of "about as effective", which we rigorously introduced in Section 5.8.1. To clarify the comparison, Table 5.13 displays the Python and Java data about the effectiveness of MBFL and SBFL on predicate bugs. On Python predicate-related bugs (left part of Table 5.13), MBFL achieves better

	FINDING		PYTHON		JAVA
1	SBFL is the most effective standalone fault localization family.	✔	f 1.1	✔	[145, f 1.1]
2	Standalone fault localization families ordered by effectiveness: SBFL > MBFL ≫ PS, ST	✔	f 1.2	✔	[145, T 3]
3	Regarding effectiveness, PS ≃ ST.	✔	f 1.2	✖	T 5.11
4	All techniques in the SBFL family achieve very similar effectiveness.	✔	f 1.5	✔	[145, T 3]
5	The techniques in the MBFL family achieve generally similar effectiveness.	✔	f 1.6	✔	[145, T 3]
6	Metallaxis tends to be better than Muse.	✔	f 1.6	✔	[145, T 3]
7	Standalone fault localization families ordered by efficiency: ST ≫ SBFL > PS > MBFL	✔	f 2.1	✔	[145, f 4.2]
8	PS is more efficient than MBFL on average.	✔	f 2.2	✔	[145, T 9]
9	ST is more effective on crashing bugs than on other kinds of bugs.	✔	f 3.5	✔	[145, f 1.3]
10	MBFL is more effective on crashing bugs than on other kinds of bugs.	✔	f 3.5	✔	[145, T 3] , [145, T 4]
11	PS is the least effective on crashing bugs.	✔	f 3.6	✔	[145, T 4]
12	On predicate-related bugs, MBFL is about as effective as SBFL.	✔	T 5.13 , f 3.7	✔	T 5.13 , [145, T 5]
13	On predicate-related bugs, PS tends to be more effective than ST.	✔	f 3.7	✔	[145, T 5]
14	Combined fault localization technique CombineFL_A, which combines all baseline techniques, achieves better effectiveness than any other techniques.	✔	f 4.1	✔	T 5.11
15	Fault localization families ordered by effectiveness: CombineFL_A > CombineFL_S > SBFL > MBFL ≫ PS, ST	✔	f 4.2	✔	T 5.11
16	Combined fault localization technique CombineFL_A, which combines all baseline techniques, achieves worse efficiency than any other technique.	✔	f 4.3	✔	[145, T 10]
17	Fault localization families ordered by efficiency: ST ≫ SBFL ≥ CombineFL_S > PS > MBFL > CombineFL_A	✔	f 4.4	✔	[145, T 10]
18	ST is more effective than PS at the function-level granularity; however, it remains considerably less effective than other fault localization techniques even at this coarser granularity.	✔	f 5.1	✔	[145, T 11]
19	ST is the most effective technique for crashing bugs.	✖	T 5.7	✔	[145, f 1.3]
20	PS is not the most effective technique for predicate-related faults.	✔	T 5.7	✔	[145, f 1.4]

Table 5.12. A comparison of findings about fault localization in Python vs. Java. Each row lists a FINDING discussed in this chapter or in Zou et al. [145], whether the finding was confirmed ✔ or refuted ✖ for PYTHON and for JAVA, and the reported evidence that confirms or refutes it (a reference to a numbered finding, Figure, or Table in this chapter or in [145]).

	FINDING	PYTHON		JAVA	
21	Different correlation patterns exist between the effectiveness of different pairs of techniques.	✔	F 5.6 , F 5.8	✔	[145, f 2.1]
22	The effectiveness of most techniques from different families is weakly correlated.	✔	F 5.6	✔	[145, f 2.2]
23	The SBFL family's effectiveness has medium correlation with the MBFL family's.	✔	F 5.6	✘	[145, T 6]
24	The effectiveness of SBFL techniques is strongly correlated.	✔	F 5.8	✔	[145, T 6]
25	The effectiveness of MBFL techniques is weakly correlated.	✘	F 5.9	✔	[145, T 6]
26	Techniques with strongly correlated effectiveness only exist in the same family.	✔	F 5.6 , F 5.8 , F 5.9	✔	[145, f 2.3]
27	Not all techniques in the same family have strongly correlated effectiveness.	✔	F 5.8 , F 5.9	✔	[145, f 2.3]
28	The main findings about the relative effectiveness of fault localization families at statement-level granularity still hold at function-level granularity.	✔	T 5.9	✔	[145, f 5.1]

Table 5.12. Continued

	PYTHON				JAVA			
FAMILY F	F@1%	F@3%	F@5%	F@10%	F@1%	F@3%	F@5%	F@10%
MBFL	11	33	40	52	9	21	29	34
SBFL	12	23	38	50	4	18	26	37

Table 5.13. A comparison of MBFL's and SBFL's effectiveness on Python and Java *predicate-related* bugs. The left part of the table reports a portion of the same data as Table 5.7: each column @k% reports the average percentage of the 52 predicate bugs in BUGSINPY Python projects used in our experiments that techniques in the MBFL or SBFL family ranked within the top-k. The right part of the table averages some of the data in [145, Table 5] by family: each column @k% reports the average percentage of the 115 predicate bugs in Defects4J Java projects used in Zou et al.'s experiments that techniques in the MBFL or SBFL family ranked within the top-k. Highlighted numbers denote each language's best family according to each metric.

@3%, @5%, and @10% than SBFL but a worse @1% (by only one percentage point); similarly, on Java predicate-related bugs (right part of Table 5.13), MBFL achieves better @1%, @3%, and @5% than SBFL but a worse @10% (by three percentage points). In both cases, MBFL is not strictly better than SBFL, but one could argue that a clear tendency exists. Regardless of the definition of "more effective" (which can be arbitrary), the conclusion we can draw remain very similar in Python as in Java.

Finding 6.1: Our experiments confirmed for Python programs most of Zou et al. [145]'s findings about fault localization techniques on Java programs.

5.10 Discussion

In this section, we discuss two aspects. In Section 5.10.1, we outline the differences between our study and another empirical study of fault localization in Python [127], discussing why we did not compare our findings to theirs. In Section 5.10.2, we examine the role of mutation operators in MBFL effectiveness, discussing why it is meaningful to compare experiments with mutation testing techniques.

5.10.1 Python vs. Java Comparison

To our knowledge, Widyasari et al.'s recent empirical study of spectrum-based fault localization [127] is the only currently available large-scale study targeting real-world Python projects. Like our work, they use the bugs in the BUGSINPY curated collection as experimental subjects [128]; and they compare their results to those obtained by others for Java [94]. Besides these high-level similarities, the scopes of our study and Widyasari et al.'s are fundamentally different: *i*) We are especially interested in comparing fault localization techniques in different *families*; they consider exclusively five *spectrum*-based techniques, and drill down into the relative performance of these techniques. *ii*) Accordingly, we consider orthogonal categorization of bugs: we classify bugs (see Section 5.6) according to characteristics that match the capabilities of different fault-localization families (e.g., stack-trace fault localization works for bugs that result in a crash); they classify bugs according to syntactic characteristics (e.g., multi-line vs. single-line patch). *iii*) Most important, even though both our study and Widyasari et al.'s compare Python to Java, the framing of our comparisons is quite different: in Section 5.9.6, we compare our findings about fault localization in Python to Zou et al. [145]'s findings about fault localization in Java; for example, we confirm that SBFL techniques are generally more effective than MBFL techniques in Python, as they were found to be in Java. In contrast, Widyasari et al. directly compare various SBFL effectiveness metrics they collected on Python programs against the same metrics Pearson et al. [94] collected on Java programs; for example, Widyasari et al. report that the percentage of bugs in BUGSINPY that their implementation of the Ochiai SBFL technique correctly localized within the top-5 positions is considerably lower than the percentage of bugs in Defects4J that Pearson et al.'s implementation of the Ochiai SBFL technique correctly localized within the top-5.

It is also important to note that there are several technical differences between ours and Widyasari et al.'s methodology. First, we handle ties between suspiciousness scores by computing the E_{inspect} rank (described in Section 5.7.2); whereas they use average rank (as well as other effectiveness metrics). Even though we also take our subjects from BUGSINPY, we carefully selected a subset of bugs that are fully analyzable on our infrastructure with all fault localization techniques we consider (Section 5.4, Section 5.8); whereas they use all BUGSINPY available bugs. The selection of subjects is likely to impact the value of *some* metrics more than others (see Section 5.7.2); for example, the exam score is undefined for bugs that a fault localization technique cannot localize, whereas the top-k counts are lower the more faults cannot be localized. These and numerous other differences make our results and Widyasari et al.'s incomparable and mostly complementary. A replication of their comparison following our methodology is an interesting direction for future work, but clearly outside the present study's scope. In Section 5.12.1 we present some additional data, and outline a few directions for future work that are directly inspired by Widyasari et al.'s study [127].

5.10.2 Mutation Testing Operators

MBFL techniques rely on mutation testing to generate mutants of a faulty program that may help locate the fault. Therefore, the selection of mutation operators that are used for mutation testing impacts the effectiveness of MBFL techniques.

Research in mutation testing has grown considerably in the last decade, developing a large variety of mutation operators tailored to specific programming languages, applications, and faults [90]. Despite these recent developments, the fundamental set of mutation operators introduced in Offut et al.'s seminal work [86] remains the basis of basically every application to mutation testing. These fundamental operators generate mutants by modifying or removing arithmetic, logical, and relational operators, as well as constants and variables in a program, and hence are widely applicable and domain-agnostic.

Notably, the Cosmic Ray [27] Python mutation testing framework (used in our implementation of FAUXPY), the two other popular Python mutation testing frameworks MutPy [32] and mutmut,[26] as well as the popular Java mutation testing frameworks Pitest[27], MuJava [79] and Major [62] (the latter used in Zou et al.'s MBFL experiments [145]) all offer Offut et al.'s fundamental operators. This helps make experiments with mutation testing techniques meaningfully comparable.

5.11 Threats to Validity

Construct validity refers to whether the experimental metrics adequately operationalize the quantities of interest. Since we generally used widely adopted and well-understood metrics of effectiveness and efficiency, threats of this kind are limited.

The metrics of effectiveness are all based on the assumption that users of a fault localization technique process its output list of program entities in the order in which the technique ranked them. This model has been criticized as unrealistic [92]; nevertheless, the metrics of effectiveness remain the standard for fault localization studies, and hence are at least adequate to compare the capabilities of different techniques and on different programs.

Using BUGSINPY's curated collection of Python bugs helps reduce the risks involved with our selection of subjects; as we detail in Section 5.4, we did not blindly reuse BUGSINPY's bugs but we first verified which bugs we could reliably reproduce on our machines.

Internal validity can be threatened by factors such as implementation bugs or inadequate statistics, which may jeopardize the reliability of our findings. We implemented the tool FAUXPY (detailed in Chapter 6) to enable large-scale experimenting with Python fault localization; we applied the usual best practices of software development (testing, incremental development, refactoring to improve performance and design, and so on) to reduce the chance that it contains fundamental bugs that affect our overall experimental results. To make it a robust and scalable tool, FAUXPY's implementation uses external libraries for tasks, such as coverage collection and mutant generation, for which high-quality open-source implementations are available.

The scripts that we used to process and summarize the experimental results may also include mistakes; we checked the scripts several times, and validated the consistency between different data representations.

We did our best to validate the test-selection process (described in Section 5.8), which was necessary to make feasible the experiments with the largest projects; in particular, we ran fault localization

[26] `https://mutmut.readthedocs.io`
[27] `https://pitest.org`

experiments on about 30 bugs without test selection, and checked that the results did not change after we applied test selection.

Our statistical analysis (Section 5.7.3) follows best practices [42], including validations and comparisons of the chosen statistical models (detailed in the replication package). To further help future replications and internal validity, we make available all our experimental artifacts and data in a detailed replication package.

External validity is about generalizability of our findings. Using bugs from real-world open-source projects substantially mitigates the threat that our findings do not apply to realistic scenarios. Precisely, we analyzed 135 bugs in 13 projects from the curated BUGSINPY collection, which ensures a variety of bugs and project types.

As usual, we cannot make strong claims that our findings generalize to different application scenarios, or to different programming languages. Nevertheless, our study successfully confirmed a number of findings about fault localization in Java [145] (see Section 5.9.6), which further mitigates any major threats to external validity.

Zou et al.'s study used the Defects4J [63] curated collection of real-world Java faults as their experimental subjects; we used the BUGSINPY [128] curated collection of real-world Python faults. This invariably limits the generalizability of our findings to *all* Python programs, and the generalizability of our comparison to all Python vs. Java programs: the two curated collections of bugs may not represent all programs and faults in Python or Java. While there is always a risk that any selection of experimental subjects is not fully representative of the whole population, choosing standard well-known benchmarks such as Defects4J and BUGSINPY helps mitigate this threat. First, BUGSINPY was explicitly inspired by Defects4J, and was built following a very similar approach but applied to real-world open-source Python programs. Second, BUGSINPY projects were "selected as they represent the diverse domains [...] that Python is used for" [128, Sec. 1], which bodes well for generalizability. Third, BUGSINPY and Defects4J are extensible frameworks, which have been and will be extended with new projects and bugs; thus, using them as the basis of FL studies helps to make future research in this area comparable to previous results. While BUGSINPY and Defects4J are only imperfect proxies for a fully general comparison of FL in Java and Python, they are a sensible basis given the current state of the art.

5.12 Conclusions

This chapter described an extensive empirical study of fault localization in Python, based on a differentiated conceptual replication of Zou et al.'s recent Java empirical study [145]. Besides replicating for Python several of their results for Java, we shed light on some nuances, and released detailed experimental data that can support further replications and analyses.

As a concluding discussion, let's highlight a few points relevant for possible follow-up work. Section 5.12.1 discusses a different angle for a comparison with other studies, suggested by Widyasari et al.'s recent work [127]. Section 5.12.2 describes broader ideas to improve the capabilities of fault localization in Python.

5.12.1 Other Fault Localization Studies

As we discussed in Section 5.10.1, Widyasari et al.'s recent work [127] is the only other large-scale study targeting fault localization in real-world Python projects. We also explained how our study's goals and methodology is quite different from theirs; as a result, we cannot directly compare most

of their findings to ours. In the following, we discuss how Widyasari et al.'s methodology suggests future work that complements our own.

Widyasari et al. directly compare FL effectiveness metrics (such as exam score) between their experiments on Python subjects from BUGSINPY and Pearson et al.'s experiments on Java subjects from Defects4J [94]. Table 5.14a displays the key results of their comparison, alongside a roughly similar comparison between our experiments on Python subjects from BUGSINPY and Zou et al.'s experiments on Java subjects from Defects4J [145].

The picture that emerges from these comparisons is somewhat inconclusive: in our comparison, there is a significant difference, with large effect size, between Python and Java with respect to exam scores, but not with respect to the E_{inspect} metric; conversely, in their comparison, there is a significant difference, with large/medium effect size, between Python and Java with respect to the top-k ranks in the best-case debugging scenarios (roughly analogous to the E_{inspect} ranking metric), whereas the differences with respect to exam scores are significant but with small effect sizes. Furthermore, the *sign* of the effect sizes is opposite: in our comparison, fault localization is more effective on Python programs (negative effect sizes); in their comparison, it is more effective on Java programs (positive effect sizes). It is plausible to surmise that these inconsistencies reflect differences between the effectiveness metrics, how they are measured in each study, and—most important—differences between the experimental subjects; the exam score metric, in particular, also depends on the size of the programs under analysis. As we discussed in Section 5.11, even though both benchmarks BUGSINPY and Defects4J are carefully curated and of significant size, there is the risk that they do not necessarily represent *all* Python and Java real-world projects and their faults. This suggests that follow-up studies targeting different projects in Python and Java (or different selections of projects from BUGSINPY and Defects4J) could help validate the generalizability of any results. Conversely, applying stricter project and bug selection criteria could also be useful not to generalize findings, but to strengthen their validity in more specific settings (for example, with projects of certain characteristics). Without provisioning stricter experimental controls, directly comparing, fault localization effectiveness metrics on sundry programs in two different programming languages, as we did in Table 5.14a for the sake of illustration, is unlikely to lead to clear-cut, robust findings.

Even though Widyasari et al.'s study found some statistically significant differences of effectiveness between SBFL techniques, those differences tend to be modest or insignificant. As shown in Table 5.14b, this is largely consistent with our findings: even though we found some weakly statistically significant differences between SBFL techniques (between DStar and Tarantula for $p < 0.1$, and between Ochiai and Tarantula for $p < 0.06$) these have little practical consequence as the effect sizes of the differences are vanishing small.

Our study did not consider two dimensions of analysis that play an important role in Widyasari et al.'s study: different debugging scenarios, and a classification of faults according to their syntactic characteristics. Debugging scenarios determine how we classify a fault as localized when it affects multiple lines. In our study, we only considered the "best-case" scenario: as long as *any* of the ground-truth locations is localized, we consider the fault localized. Widyasari et al. also consider other scenarios such as the worst-case scenario (*all* ground-truth locations must be localized). While they did not find any significant differences in the various findings under different debugging scenarios, investigating the robustness of our empirical findings in different scenarios remains a viable direction for future work.

5.12.2 Future Work

One of the dimensions of analysis that we included in our empirical study was the classification of projects (and their bugs) in categories, which led to the finding that faults in data science projects

		THIS CHAPTER			[127]			
METRIC	TECHNIQUE L	p	EFFECT		p	EFFECT		REFERENCE
$\mathscr{E}(L)$	DStar	0.0000	−0.64	L	0.000000	0.32	S	[127, Tab. 5]
	Ochiai	0.0000	−0.64	L	0.000093	0.15	S	
$\mathscr{I}(L)$	DStar	0.0000	−0.27	S	0.000000	0.54	L	[127, Tab. 3]
	Ochiai	0.0000	−0.28	S	0.000000	0.41	M	

(a) Comparison of SBFL techniques on Python vs. Java programs. Each row compares the same SBFL TECHNIQUE L applied to Python and to Java programs, reporting the p-value of a Wilcoxon rank-sum test, and Cliff's delta EFFECT size; a letter gives a qualitative assessment of the effect size: N for negligible, S for small, M for medium, and L for large. The data for THIS CHAPTER is each technique L's exam score $\mathscr{E}(L)$ and E_{inspect} rank $\mathscr{I}(L)$ for each bug among all 135 Python bugs used in the rest of this chapter's experiments, and for each Java bug in Zou et al.'s replication package data [145]; to reflect the behavior on all bugs in these statistics, bugs that were not localized are assigned an $\mathscr{I}$ rank and an exam score of -1 (unlike the rest of this chapter where this value is undefined). The statistics of [127] (in the four rightmost columns) are taken from its Table 5 (exam score, which they compute based on their top-k ranks) and Table 3 (best-case debugging scenario top-k ranks).

		THIS CHAPTER			[127, Tab. 14]	
TECHNIQUE L_1	TECHNIQUE L_2	p	EFFECT		EFFECT	
DStar	Ochiai	0.584	0.00	N	0.14	N
DStar	Tarantula	0.093	−0.01	N	0.19	S
Ochiai	Tarantula	0.056	−0.01	N	0.04	N

(b) Pairwise comparison of SBFL techniques according to exam score. Each row compares the exam scores of two TECHNIQUEs L_1 and L_2 for significant differences, reporting the p-value of a Wilcoxon signed-rank test, and Cliff's delta EFFECT size; a letter gives a qualitative assessment of the effect size: N for negligible, S for small, M for medium, and L for large. The data for THIS CHAPTER is each technique L's exam score $\mathscr{E}(L)$ for each bug among all 135 Python bugs used in the rest of this chapter's experiments; to reflect the behavior on all bugs in these statistics, bugs that were not localized are assigned an exam score of -1 (unlike the rest of this chapter where this value is undefined). The statistics of [127] (in the two rightmost columns) are taken from its Table 14.

Table 5.14. A summary of some data presented in Widyasari et al.'s fault localization study [127] vis-à-vis analogous data presented in this chapter.

tend to be harder and take longer to localize. This is not a surprising finding if we consider the sheer size of some of these projects (and of their test suites). However, it also highlights an important category of projects that are much more popular in Python as opposed to more "traditional" languages like Java. In fact, a lot of the exploding popularity of Python in the last decade has been connected to its many usages for statistics, data analysis, and machine learning. Furthermore, there is growing evidence that these applications have distinctive characteristics—especially when it comes to faults [53, 57, 99]. Thus, investigating how fault localization can be made more effective for certain categories of projects is an interesting direction for related work.

It is remarkable that SBFL techniques, proposed nearly two decades ago [61], still remain formidable in terms of both effectiveness and efficiency. As we discussed in Section 2.3, MBFL was introduced expressly to overcome some limitations of SBFL. In our experiments (similarly to Java projects [145]) MBFL performed generally well but not always on par with SBFL; furthermore, MBFL is much more expensive to run than SBFL, which may put its practical applicability into question. Our empirical analysis of "mutable" bugs (Section 5.9.3) indicated that MBFL loses to SBFL usually when its mutation operators are not applicable to the faulty statements (which happened for nearly half of the bugs we used in our experiments); in these cases, the mutation analysis will not bring relevant information

about the faulty parts of the program. These observations raise the question of whether it is possible to predict the effectiveness of MBFL based on preliminary information about a failure; and whether one can develop new mutation operators that extend the practical capabilities of MBFL to new kinds of bugs. More generally, one could try to relate the various kinds of source-code edits (add, remove, modify) [115] introduced to fix a fault to the effectiveness of different fault localization algorithms. We leave answering these questions to future research in this area.

6

FAUXPY: an Automated Fault Localization Tool For Python

In Chapter 5, we conducted a large-scale empirical study of fault localization (FL) in Python programs. In our experiments, we included seven FL techniques from four families and used bugs from real-world Python projects as subjects. As noted in that chapter, we did not find any multi-family FL tool for Python; thus, we developed FAUXPY to conduct our study. To our knowledge, FAUXPY is the first open-source FL tool for Python that supports multiple FL families.

In this chapter, we provide a detailed explanation of FAUXPY by showcasing how to use it on two illustrative examples and discussing its main features and capabilities from a user's perspective. The experiments in Chapter 5 demonstrate that FAUXPY is applicable to analyze Python projects of realistic size as we ran it on projects from BUGSINPY curated dataset of Python bugs [128]. In this chapter, we present a different summary of the same FL experiments discussed in Chapter 5.

Structure of the Chapter

The current chapter is organized as follows:

Section 6.1 provides the motivation behind developing FAUXPY.

Section 6.2 overviews how to use FAUXPY through two illustrative examples.

Section 6.3 highlights details of FAUXPY's architecture and its implementation.

Section 6.4 presents some experimental results.

Section 6.5 concludes this chapter.

6.1 Introduction

Starting from around the 1990s [131], there has been a growing interest in automated *fault localization* techniques for programs, which spurred the development of increasingly sophisticated and effective techniques. Nowadays, fault localization techniques are widely used both on their own, and as components of more complex (dynamic) program analyses—for example, as ingredients of automated program repair.

Like with every program analysis technique, practical adoption of fault localization critically requires that reusable, flexible *tool* implementations are available, so that trying out new research ideas and applications does not require to re-implement from scratch techniques that are already

known to work. Although there exists several fault localization tools [22, 50, 59, 103] in the literature, they are mostly developed for programming languages such as Java, C/C++, and the .NET languages. To our knowledge, CharmFL [55] is the only publicly available fault localization tool for Python. CharmFL, implemented as a plugin of the PyCharm IDE, only supports spectrum-based fault localization (SBFL) techniques.

To address this deficiency, this chapter describes FAUXPY (read: "foh pie"): a fault localization tool for Python. To our knowledge, FAUXPY is the only available Python fault localization tool that supports multiple fault localization families (spectrum based, mutation based, stack-trace based, and predicate switching). The immediate motivation for implementing FAUXPY was to carry out the large scale empirical study of fault localization in Python, which we described in Chapter 5. Nevertheless, we designed and implemented FAUXPY with the broader goal of making it a flexible, reusable standalone tool for all applications of fault localization in Python.

The current chapter presents, focusing on the user's perspective, the tool FAUXPY, some of its concrete usage scenarios (Section 6.2), and its main features and implementation (Section 6.3). FAUXPY supports seven fault localization techniques, and two localization granularities (statement and function); it can use tests written for the most popular Python unit testing frameworks such as Pytest and Unittest; it can be extended with support for new techniques. To demonstrate that FAUXPY is applicable to real-world projects, we also summarize some of the results of the empirical study presented in Chapter 5 from a different perspective—grouping the data by project.

FAUXPY is available as open source; users can easily install FAUXPY from PyPI[1], using `pip install fauxpy`. FAUXPY's source code is also publicly available.[2] In addition, a companion repository[3] makes available the complete dataset of our related empirical study [101] presented in Chapter 5. A short demo of FAUXPY is also available on Youtube.[4]

6.2 Using FAUXPY

This section overviews using FAUXPY on two simple examples, from the perspective of Moe—a nondescript user.

6.2.1 Spectrum-based and Mutation-based Fault Localization

To practice programming in Python, Moe has implemented function `equilateral_area` in Listing 6.1. The function takes as input the length `side` of an equilateral triangle's side, and returns its area computed using the formula $\texttt{side}^2 \times \sqrt{3}/4$. Unfortunately, Moe inadvertently introduced a bug on line 196, which sums variables `const` and `term` instead of *multiplying* them. Fortunately, the bug does not go unnoticed thanks to the tests that Moe also wrote (see Listing 6.2); in particular, the assertion in test `test_ea_fail` fails, indicating that `equilateral_area` does not work as intended.

To help him debug `equilateral_area`, Moe runs our fault localization tool FAUXPY. All fault localization techniques implemented by FAUXPY are *dynamic* (i.e., based on tests); therefore, Moe points FAUXPY to the location of `equilateral_area`'s implementation, as well as of its *tests*. By default, FAUXPY performs spectrum-based fault localization (SBFL)—a family of widely used fault localization techniques based on the idea of comparing program traces ("spectra") of passing and failing runs of a program (see Section 5.2.1). FAUXPY currently supports three techniques (DStar [130], Ochiai [6], Tarantula [61]) that belong to the SBFL family; since they only differ in the formula used to aggregate

[1] `https://pypi.org/project/fauxpy`
[2] `https://github.com/atom-sw/fauxpy`
[3] `https://github.com/atom-sw/fauxpy-experiments`
[4] A demo of FAUXPY: `https://youtu.be/O4T7w-U8rZE`

```
191 def equilateral_area(side):
192   const = math.sqrt(3) / 4
193   if side == 1:
194     return const
195   term = math.pow(side, 2)
196   area = const + term        # bug
197   return area
```

Listing 6.1. Python function `equilateral_area` computes the area of an equilateral triangle given its `side` length; this implementation has a bug at line 196.

```
198 def test_ea_fail():
199   area = equilateral_area(side=3)
200   assert area == pytest.approx(9 * math.sqrt(3) / 4)
201
202 def test_ea_pass():
203   area = equilateral_area(side=1)
204   assert area == pytest.approx(math.sqrt(3) / 4)
```

Listing 6.2. Tests for function `equilateral_area` in Listing 6.1. Library function `pytest.approx` checks equality of floating points within some tolerance.

the information about traces, FauxPy reports the output for all SBFL techniques with a single analysis run.

SBFL runs quite fast, taking only 0.3 seconds on this example. The output, like for every fault localization technique, is a list of program locations (identified by line numbers) ranked by their *suspiciousness score*; the absolute value of the suspiciousness score does not matter, what matters is the *rank* of a location: the higher its rank, the more likely the location is implicated with the failure triggered by the tests. As shown in Table 6.1, all three SBFL techniques correctly assign the top rank to the fault location (line 196 in Listing 6.1); however, they also assign the top rank to some nearby locations (lines 195 and 197) which tie the faulty location's suspiciousness score. This example highlights a fundamental limitation of SBFL techniques: since they compare traces in different executions, they cannot distinguish between locations that are in the same basic block (a portion of code without branches).

Mutation-based fault localization (MBFL) techniques use a different approach, which is capable of distinguishing between locations in the same basic block (Section 5.2.2). As the name suggests, MBFL techniques are based on mutation testing: given a program to analyze, they generate many different *mutants*—syntactic mutations obtained by systematically applying a number of mutation operators. The intuition is that if mutating the code at a certain program location changes the program behavior (a test passes on the original program and fails on the mutant, or vice versa), then the program location is likely to be implicated with the fault.

To run MBFL with FauxPy, Moe simply adds the option `--family mbfl`. FauxPy currently supports two techniques (Metallaxis [91] and Muse [83]) that belong to the MBFL family; just like for SBFL techniques, a single analysis run of FauxPy computes the output of both MBFL techniques. MBFL is notoriously time consuming; in fact, it takes 15.9 seconds on Listing 6.1's example (over 50 times longer than SBFL). As shown in Table 6.1, the two MBFL techniques achieve quite different results despite using the same 32 mutants for analysis: Muse is very accurate, as it singles out line 196 as the most suspicious location; Metallaxis also ranks it at the top, but together with two other locations

		Listing 6.1		Listing 6.3	
FAMILY	TECHNIQUE	TIME [seconds]	TOP-RANK LOCATIONS	TIME [seconds]	TOP-RANK LOCATIONS
MBFL	Metallaxis	15.9	193 195 196	18.4	207, 208, 210
	Muse		196		207, 208, 210
SBFL	DStar	0.3	195, 196, 197	0.1	206, 207, 208, 210
	Ochiai		195, 196, 197		206, 207, 208, 210
	Tarantula		195, 196, 197		206, 207, 208, 210
PS		1.2	–	0.2	–
ST		0.2	–	0.1	206, 207, 208

Table 6.1. A summary of running FAUXPY on the two examples in Listing 6.1 (`equilateral_area`) and Listing 6.3 (`isosceles_area`). For each fault localization technique (grouped by family), the table reports the running TIME of FAUXPY in seconds, and the program locations (line numbers) with the highest suspiciousness (TOP-RANK). A colored background highlights the actual location of the bug in each example. Since the running time of all techniques in a family is the same, it is only reported once per family.

```
205 def isosceles_area(leg, base):
206   def height():
207     t1, t2 = math.pow(base, 2), math.pow(leg, 2) / 4    # bug
208     return math.sqrt(t1 - t2)
209
210   area = 0.5 * base * height()
211   return area
```

Listing 6.3. Python function `isosceles_area` computes the area of an isosceles triangle given its `leg` and `base` lengths; this implementation has a bug at line 207.

that are not responsible for the fault.

6.2.2 Stack Trace and Predicate Switching Fault Localization

FAUXPY supports two other fault-localization families: stack-trace (ST [145]) fault localization and predicate switching (PS [140]).[5] Moe tries them out on `equilateral_area` but the results are disappointing: both techniques return the empty list of locations, meaning that they could not gather any evidence of suspiciousness. The reason for ST's failure in this case is quite obvious: ST analyzes the stack trace dumped after a program *crash* (usually, an uncaught exception); since all tests in Listing 6.2 terminate without crashing, ST is completely ineffective on this example (Section 5.2.4).

In order to try an example where ST may stand a chance, Moe considers another little Python program he wrote: function `isosceles_area` returns the area of an isosceles triangle computed as $\texttt{base}/2 \times \sqrt{\texttt{leg}^2 - \texttt{base}^2/4}$.[6] The implementation in Listing 6.3 erroneously swaps `base` and `leg`; thus, when executing test `test_ia_crash`, expression `t1 - t2` in Listing 6.3 evaluates to a negative number, which crashes the program with an uncaught `ValueError` exception raised by `math.sqrt`.

Executing FAUXPY with option `--family st` on Listing 6.3's example terminates quickly (around 0.1 seconds) and ranks the three locations 206, 207, 208 as top suspiciousness. Even this simple

[5] Unlike SBFL and MBFL, there is only one implementation of ST and one of PS—hence, ST and PS denote both families and individual techniques.

[6] The *legs* of an isosceles triangle are the two sides of equal length; the third side is called *base*.

```
212 def test_ia_crash():
213   area = isosceles_area(leg=9, base=4)
214   assert area == pytest.approx(2 * math.sqrt(77))
215
216 def test_ia_pass():
217   area = isosceles_area(leg=4, base=4)
218   assert area == pytest.approx(2 * math.sqrt(12))
```

Listing 6.4. Tests for function `isosceles_area` in Listing 6.3. Library function `pytest.approx` checks equality of floating points within some tolerance.

example showcases ST's key features: first, it is usually very fast, since it does not have to collect any information other than the stack trace of crashing tests. Second, it can be quite effective with crashing bugs (after all, the fault location 207 is ranked at the top), but fundamentally operates at the level of whole *functions*: a stack trace reports the list of functions that were active when the program crashed; hence, ST fault localization cannot distinguish between the suspiciousness of locations that belong to the same function (`height` in the example, which consists of three lines). Still, on this example, ST is a bit more accurate than SBFL (which also ranks line 210 in the top position), and arguably somewhat better than MBFL (which also reports three locations at the top rank, but one of them is line 210, which is the call location of `height`, and hence not really responsible for the fault). The running time of ST and SBFL is practically indistinguishable on this simple example; in general, however, SBFL takes more time than ST because the latter only runs failing tests and does not require any *tracing* when executing the tests. As usual, MBFL takes considerably longer (18.4 seconds) to generate several mutants (48 mutants) and to execute all tests on each mutant.

As a last experiment of FAUXPY's capabilities, Moe runs PS fault localization on `isosceles_area`. Just like on `equilateral_area`, PS fails to localize the bug and returns an empty list of locations. Once again, the program features explain why these examples are a poor match for PS's capabilities. As the name suggests, PS is based on the idea of *forcefully changing* the outcome of a program conditional branch dynamically during different test executions; the intuition is that if switching a *predicate* (branch condition) turns a failing test into a passing one, then the predicate may be responsible for the fault. Clearly, if a program has no conditionals (like `isosceles_area`), or its conditionals are unrelated to the locations of failure (like `equilateral_area`), PS is unlikely to be of any help to locate the bug (Section 5.2.3).

In all, this section's simple examples gave a concrete idea of FAUXPY's capabilities, showcasing the variety of fault localization techniques that it supports and how they can be applied.

6.3 FAUXPY's Architecture and Implementation

FAUXPY is an automated fault localization tool for Python. The current version of FAUXPY supports seven fault localization techniques in four families: the spectrum-based (SBFL) techniques DStar [130], Ochiai [6], and Tarantula [61]; the mutation-based (MBFL) techniques Muse [83] and Metallaxis [91]; and the predicate switching (PS) [140] and stack trace (ST) [145] fault localization families/techniques.

FAUXPY can perform fault localization with two granularities: statement-level and function-level. That is, the granularity determines what program *entities* are localized: the locations of individual statements in the source code, or the functions that compose the programs.

FAUXPY is a command-line tool, implemented as a plugin of the popular Pytest testing framework.

As essential input, FAUXPY takes the location of the source code of a Python project where to perform fault localization, as well as the location of a test suite. FAUXPY accepts tests in the formats of Pytest, as well as Unittest (another widely used Python testing framework) and Hypothesis (a property-based Python testing tool, which supports the definition of parametric tests). As output, FAUXPY returns a CSV file listing program entities ranked by their suspiciousness score; the higher an entity's suspiciousness score, the more likely the entity is the location of the fault.

6.3.1 Features and Options

The only mandatory command line argument to use FAUXPY is `--src PACKAGE`, which runs SBFL at the statement granularity on the Python package in directory `PACKAGE`, using any tests discovered by Pytest within the project's source files.

Flags `--family` and `--granularity` respectively select the fault localization family (SBFL, MBFL, ST, and PS) and the granularity (statement and function) at which to perform fault localization. As mentioned in Section 6.2, FAUXPY simultaneously runs all techniques that belong to the selected family, since it is able to reuse the output of the same underlying analysis.

Using Pytest's command line options, users can select specific tests to be used by FAUXPY. For example, you can run a test selection algorithm to identify a subset of the tests, and then feed its output to FAUXPY. The command line option `--failing-list` explicitly asks FAUXPY to only use the given list of *failing* tests. Normally, FAUXPY runs all available tests, and figures out which are passing and which are failing. However, a technique like ST only needs to run failing tests; thus, if those are given to FAUXPY explicitly, ST can run much faster by simply ignoring all passing tests. Another scenario where selecting failing tests is useful is whenever a test suite includes multiple failing tests that trigger *different* bugs; localizing one bug at a time is likely to increase the effectiveness of fault localization techniques—whose heuristics usually assume that all failures refer to the same fault.

6.3.2 Implementation

Figure 6.1 overviews the workflow of FAUXPY. The first step of FAUXPY's dynamic analysis is always running the available tests. Then, different components collect different kind of information required by the selected fault localization technique.

SBFL techniques rely on coverage information; to this end, FAUXPY runs Coverage.py [15], a popular coverage library for Python. MBFL techniques generate several mutants of the input program; to this end, FAUXPY uses state-of-the-art mutation framework Cosmic Ray [27]. By default, FAUXPY applies the framework's default mutation operators, but users can also provide other custom operators. FAUXPY includes a module ps_inst that we developed to generate the kind of instrumentation needed by PS fault localization; our implementation is based on Python's `ast` library. FAUXPY's support of ST fault localization parses the dumped output of all crashing tests, reconstructs the stack trace, and then locates the corresponding functions in the program's source code.

FAUXPY outputs the results of its fault localization analysis in CSV format, encoding a ranked list of program entities and their suspiciousness scores. For performance reason, FAUXPY stores all the intermediate results (the outcome of running the various analyses and tools) of its analysis in an SQLite database file. This SQLite database remains available to the user after FAUXPY terminates executing, which can be useful both for debugging and to perform additional analyses on the large amount of data collected by FAUXPY's dynamic analysis.

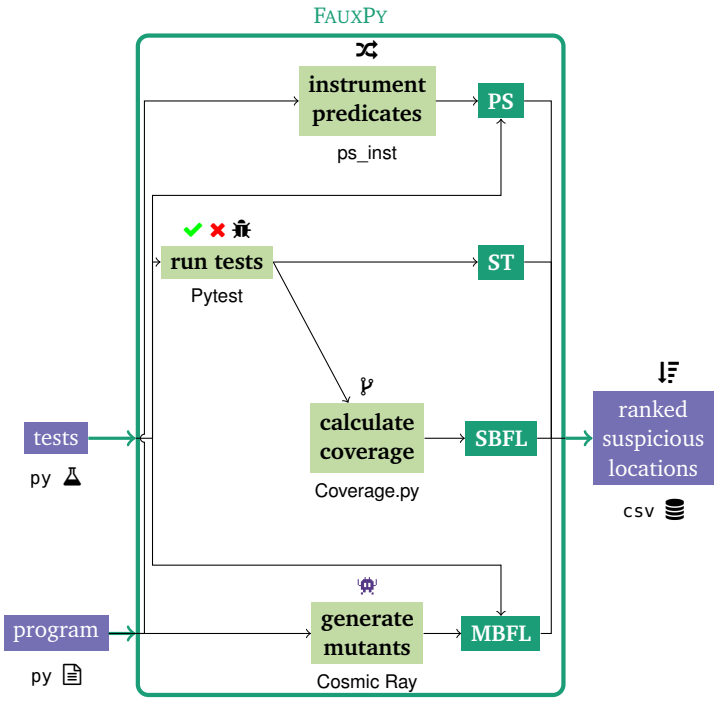


Figure 6.1. An overview of FAUXPY's architecture.

6.4 Experiments

Table 6.2 presents a different summary of the same FL experiments discussed in Chapter 5: in particular, it groups data by project. These experiments involved 135 bugs from 13 open-source Python projects taken from the BUGSINPY curated collection of real-world Python bugs [128]. Each bug b in BUGSINPY consists of two revisions B_b, F_b of a Python project complete with its programmer-written tests; the first revision B_b includes a bug exposed by the tests, and the second revision F_b is the programmer-written fix. Overall, these experiments involve over half a million lines of code and over 15 thousand test functions.

For each bug b, we ran FAUXPY on each buggy revision B_b, and used the fixed revision F_b to determine whether FAUXPY localized the actual bug locations (i.e., where the programmer edited the program to fix it). As key metric of fault localization accuracy (effectiveness), Table 6.2 reports the @5 count for each fault localization technique: the number of bugs that the technique correctly localized within the top-5 ranks of its output. This is a common metric of fault localization effectiveness, which is based on a scenario where the user only inspects a few (i.e., five) locations in the output, and ignores any other locations that are ranked lower. Table 6.2 indicates that SBFL techniques (DStar, Ochiai, Tarantula) are the most effective ones according to this metric, followed by MBFL techniques (Metallaxis, Muse). As we explained intuitively in Section 6.2, PS and ST are more specialized techniques that are only applicable to bugs that involve branching predicates (PS) or that result in a crash (ST); in fact, they are accurate only for a fraction of the bugs in the experiments.

The average running time of FAUXPY on each bug (also reported in Table 6.2) confirms on a much larger scale the same trends that Section 6.2's toy examples demonstrated in the small. Namely, ST is by far the fastest technique, since it just runs failing tests (usually, only a handful of a whole test suite); SBFL is still nimble but has to run *all* tests while collecting coverage information; PS and MBFL take considerably more time, since they have to run all tests on several variants of the programs. We

PROJECT	KLOC	TESTS	FAULTS	@5 COUNT ON PROJECT							AVERAGE TIME/FAULT [sec]			
				MBFL			PS	SBFL			ST	MBFL	PS	SBFL	ST
				Metallaxis	Muse	PS	DStar	Ochiai	Tarantula	ST					
black	93.5	153	13	5	1	2	4	4	4	1	28 936	45 149	62	1
cookiecutter	1.6	218	4	0	0	0	2	2	2	0	51	13	9	1
fastapi	4.7	595	13	3	3	1	5	5	5	1	592	745	7	1
httpie	3.5	217	4	0	3	1	1	1	1	0	646	116	9	1
keras	6.7	616	18	6	4	0	6	7	7	0	31 330	2 977	196	4
luigi	22.0	1 508	13	7	4	1	5	5	5	2	14 188	1 486	22	1
pandas	128.0	12 226	18	2	1	2	3	3	3	0	36 561	29 653	3 810	1
sanic	7.3	466	3	0	0	0	1	1	1	0	11 772	365	209	0
spaCy	97.2	986	6	1	2	1	3	3	3	0	4 920	13 916	60	0
thefuck	4.7	614	16	7	7	0	15	15	15	1	73	49	6	1
tornado	17.9	926	4	1	1	0	2	2	2	0	28 013	1 445	976	1
tqdm	3.3	120	7	1	0	0	4	4	4	2	7 154	192	42	1
youtube-dl	125.0	237	16	7	8	1	6	6	6	1	6 767	1 257	54	4
total	515.4	18 882	135	40	34	9	57	58	58	8	15 774	9 751	589	2

Table 6.2. Overview of FAUXPY's experimental evaluation on an ample selection of bugs from BUGSINPY [128]. For each PROJECT, the table shows its size in KLOC (thousands of non-empty non-comment lines of code, excluding tests), number of TESTS (i.e., test functions), number of FAULTS analyzed with FAUXPY, and how many of them each technique correctly localized within the top-5 positions (@5 COUNT), and the AVERAGE TIME per fault taken by FAUXPY (by fault localization family, since all techniques in a family take the same time). Consistently with what done by the authors of BUGSINPY [128] and similar to Table 5.1, the project statistics reported here refer to the latest version of the projects on 2020-06-19.

refer interested readers to Chapter 5 for many more details about the practical capabilities of different fault localization techniques on Python programs.

6.5 Conclusions

This chapter presented FAUXPY, an automated fault localization tool for Python programs. We explained the motivation behind developing FAUXPY, its implementation details, and simple examples of usage. We presented a summary of Chapter 5's experimental results—from a different perspective—to emphasize that FAUXPY is a flexible tool, usable on projects of considerable size.

Part IV

Epilogue

7

Conclusions and Future Work

In this dissertation, we investigated the capabilities and limitations of state-of-the-art test generation and fault localization techniques on data science programs implemented in Python with the aim of laying the foundation for development of new debugging techniques that are more effective on these programs.

We started the first part of the dissertation with analyzing the effectiveness of general-purpose test generation techniques on neural network (NN) programs, a wide spread class of data science programs. Based on this analysis, we proposed ANNOTEST—a novel test generation technique tailored for NN programs—to address the limitations of general-purpose techniques (Chapter 3). We then presented the ANNOTEST tool—an implementation of the ANNOTEST technique—along with a curated collection of NN bugs to provide tool support for developers and researchers in the field (Chapter 4).

Broadening our focus from data science programs to open-source Python programs, we continued the dissertation in the second part by conducting a large scale multi-family empirical study of fault localization in Python programs (Chapter 5). The empirical knowledge achieved in this study is beneficial for both researchers and developers working on the debugging of Python programs, including data science programs. Finally, we finished the second part of the dissertation by introducing FAUXPY, a multi-family automated fault localization tool for Python programs (Chapter 6). In the rest of this chapter, we provide a summary of our contributions throughout the entire dissertation and highlight possible avenues for further work to continue this research.

Structure of the Chapter

The current chapter is organized as follows:

Section 7.1 highlights the contributions and accomplishments made throughout the dissertation.

Section 7.2 outlines potential avenues for future research.

Section 7.3 concludes this chapter with closing remarks.

7.1 Contributions

The contributions described in this dissertation fit in three groups: *i*) the test generation technique, ANNOTEST, tailored for NN programs written in Python (Chapter 3); *ii*) the multi-family large scale empirical study of fault localization in Python programs (Chapter 5); *iii*) and the tools ANNOTEST and FAUXPY, as well as the curated dataset of NN bugs (Chapters 4 and 6). This section provides a summary of these contributions.

7.1.1 Test Generation Approach for NN Programs

We started Chapter 3 by investigating the effectiveness of general-purpose test generation tools on neural network (NN) programs, and then, we proposed aNNoTest, a novel test generation technique tailored specifically for NN programs written in Python. NN programs usually work with complicated data types while typing information is not available in dynamically typed language such as Python. As a result, general-purpose test generation tools such as Pynguin [78] and Deal [29] are ineffective on NN programs. To address this issue, aNNoTest relies on aN, a simple annotation language that can express the kinds of constraints found in NN programs, concisely and precisely; aNNoTest can generate tests for programs that are annotated with aN.

We experimentally evaluated aNNoTest on 19 open-source programs we curated from Islam et al. [57]'s survey of NN bugs. These programs use some of the popular NN frameworks such as Keras, TensorFlow, and PyTorch. Upon annotating 24 functions in these programs using aN, aNNoTest reproduced 63 known bugs along with 31 previously unknown bugs in them. To also evaluate aNNoTest when used comprehensively, we fully annotated the latest version of two of these 19 open-source programs, adding 330 annotations. In this experiment, aNNoTest found 50 bugs with only 6 false positives. These two experiments indicate that aNNoTest can be used for an entire program, or opportunistically on only a few selected functions that are critical.

While aNNoTest generates tests for programs annotated with aN, annotating remains a manual process. In our experiments, we added 2 annotations per function on average, 96% of which fit a single line. These statistics indicate that the manual effort required by aNNoTest is reasonable compared to the number of bugs it can find in NN programs. Furthermore, aNNoTest generated tests with high coverage, comparable to that of developer written tests. Chapter 3 is based on our following publication [99]:

- M. Rezaalipour and C. A. Furia. **An annotation-based approach for finding bugs in neural network programs.** Journal of Systems and Software, 201:111669, 2023.

7.1.2 Empirical Study of Fault Localization in Python Programs

In Chapter 5, we presented the first multi-family large-scale empirical study of fault localization (FL) in Python programs to better understand the capabilities of fault localization on both Python and data science programs. This study is a differentiated conceptual replication [65] of Zou et al. [145]'s work on Java.

In our empirical study, we considered seven FL techniques from four families as well as their combinations. As subjects, we included 135 bugs across 13 well-known projects from the BugsInPy [128] collection of Python project bugs. We classified these bugs based on both their project categories and bug types; we studied fault localization across three different levels of granularity—statement, function, and module—while measuring both the effectiveness and efficiency of FL techniques in our analysis.

Our empirical study provides several findings, which we presented in Chapter 5. Having compared our findings to those from Zou et al. [145], we confirmed that most of Zou et al. [145]'s findings about FL in Java also apply to Python. Furthermore, our study indicated that bugs in data science projects tend to be harder to localize compared to other categories of projects. Chapter 5 is based on our following publication [101]:

- M. Rezaalipour and C. A. Furia. **An empirical study of fault localization in Python programs.** Empirical Software Engineering, 2024. Accepted in March 2024.

7.1.3 Supporting Tools and Dataset

In order to conduct the studies outlined in Section 7.1.1 and Section 7.1.2, we developed two tools and a curated collection of NN bugs. The first tool we developed was aNNoTest, the implementation of the aNNoTest technique detailed in Chapter 3. We presented the details of this tool in Chapter 4. aNNoTest provides a library of annotations, which users can import into their Python modules to annotate functions and methods in their code. aNNoTest also provides a test generator engine, which takes the path to an annotated program and outputs Hypothesis [80] test templates for the annotated sections of the source code. Users can then run these Hypothesis templates to generate concrete inputs for their projects and find bugs.

Along with the aNNoTest tool, we also presented a curated collection of NN bugs in Chapter 4, which we sourced from Islam et al. [57]'s survey of NN bugs. We spent substantial effort in collecting this dataset to make sure it contains all the required dependencies and the bugs are all reproducible. Chapter 3 is based on our following publication [100]:

- M. Rezaalipour and C. A. Furia. **aNNoTest: An annotation-based test generation tool for neural network programs.** In IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 574–579, 2023.

To conduct our empirical study of fault localization (FL) outlined in Chapter 5, we developed FauxPy, a multi-family FL tool for Python programs. Chapter 6 presents an in-depth exploration of FauxPy's architecture. FauxPy supports seven FL techniques across four families—spectrum-based, mutation-based, predicate switching, and stack-trace based FL—and operates at two levels of granularity: statement and function. It works with tests written for Unittest, Pytest, and Hypothesis, three well-known Python testing frameworks. We designed FauxPy to be extensible. For instance, FauxPy can be extended to support more spectrum-based ranking metrics. Using Cosmic Ray [27]—a well-known mutation testing framework—as backend for mutation-based FL techniques enables FauxPy to support new custom-designed mutation operators. FauxPy is also designed to be efficient. We achieved efficiency by storing each technique's intermediate data in an SQLite database file to reuse them in different FL tasks. This approach also enables further analysis as this stored data provides detailed information about the different steps each FL technique takes.

7.2 Future Work

The current section explores potential future avenues to extend the work presented in this dissertation. In this regard, Section 7.2.1 explores future work on our test generation study, particularly, focusing on the enhancement of aNNoTest; Section 7.2.2 discusses how to further extend our FL study; Section 7.2.3 focuses on opportunities for automated program repair studies built upon our dissertation.

7.2.1 Test Generation

We can think of three potential avenues as future work of our test generation study (i.e., aNNoTest), which we outline below.

Type constraints expansion. The current version of aN supports several type constraints. However, as discussed in Section 3.7.3, we had to write a few custom generators as some of the function parameters in our subjects were of complex types such as NN model objects and tensors. Extending

AN to support such complex type constraints enhances the convenience and applicability of using ANNOTEST.

Postconditions integration. ANNOTEST's main contribution is that it generates valid test inputs, according to the input constraints specified by the annotations. In our experiments, detailed in Chapter 3, we exclusively considered implicit oracles (i.e., the crashing oracle). However, extending ANNOTEST to support other types of oracles is feasible. Since AN is based on annotations—a form of lightweight formal specification—adding *postconditions* can extend ANNOTEST to support custom-made oracles. This feature can be implemented by checking whether the postconditions hold at the end of test functions to determine whether the tests pass or fail.

Regression testing expansion. The primary objective of the ANNOTEST technique proposed in this dissertation is to find bugs, i.e., generating failing tests. However, there exists potential for extending ANNOTEST to also generate regression tests. Test generation tools such as EvoSuite [39] and Pynguin [78] specialize in generating regression tests, using regression oracles. Such tools leverage the post-state of the program under test as regression oracle to formulate assertions for the tests they generate. Regression tests can be run on future versions of the same program to detect regression bugs. Considering ANNOTEST's focus, it could be extended to target regression oracles that capture NN-specific properties [34, 85, 137] and generate regression tests.

7.2.2 Fault Localization

There are three potential directions to extend our work on fault localization. We outline these directions in the following.

Data-science program fault localization. In our Chapter 5's empirical study, we explored fault localization effectiveness and efficiency across different project categories. Our study revealed that localizing faults in data science projects is more challenging compared to projects from other domains, a finding consistent with existing studies [57, 58] that attribute these challenges to the distinctive nature of these programs. It is an interesting direction to investigate strategies for improving fault localization for certain categories of projects such as data science programs.

Enhancements on mutation-based fault localization. As discussed in Section 2.3, MBFL techniques were designed to address the limitations of SBFL techniques. However, our Chapter 5's study, as well as other fault localization studies [94, 145] showed that although MBFL techniques perform generally well, they do not consistently outperform SBFL techniques. This observation puts MBFL's practical applicability into question as it is more expensive than SBFL. However, our empirical analysis of "mutable" bugs (Section 5.9.3) revealed that MBFL's effectiveness diminishes when its mutation operators are not applicable to the faulty statements (which happened for nearly half of the bugs we used in our experiments). In such cases, mutation analysis fails to provide insights about the buggy statements within the program. This observation raises the question of whether devising strategies to predict MBFL effectiveness based on some information about the failure is possible. Employing such strategies would allow users to know in advance whether to spend resources in MBFL or use SBFL, the more efficient family. Another future direction can be to investigate the feasibility of proposing new mutation operators specific to project categories or bug types in order to address this issue.

Exploring debugging scenarios. In addition to our Chapter 5's empirical study of fault localization in Python, Widyasari et al. [127] also conducted an study that targets real-world Python projects. While we considered four FL families in our study, their focus was only on SBFL. Section 5.10.1 outlines all the methodological differences between our study and Widyasari et al.'s; their methodology offers another potential avenue to continue our research. In their study, Widyasari et al. considered two analysis dimensions that we did not include in our study: different debugging scenarios, and a classification of faults according to their syntactic characteristics. Debugging scenarios specify when a bug is considered localized by a FL technique in cases where the bug affects multiple lines, i.e., when the ground-truth (Section 5.5) contains multiple faulty locations. Widyasari et al. included three debugging scenarios in their study: *i*) *best-case* scenario, where a bug is consider localized if the FL technique finds any of its faulty statements; *ii*) *average-case* scenario, where the technique must find half of the faulty statements; *iii*) *worst-case* scenario, where the technique must find all faulty statements. Similar to [145], the best-case was the only debugging scenario we considered in our Chapter 5's study. While Widyasari et al. did not find any significant differences in the various findings across different debugging scenarios, investigating the robustness of our empirical findings under different scenarios is an interesting avenue for future work.

7.2.3 Automated Program Repair

In this dissertation, we introduced ANNoTest (Chapter 3), a test generation technique designed for NN programs. Using ANNoTest, we curated a dataset of NN bugs, presented in Chapter 4. Additionally, we developed FauxPy (Chapter 6)—a multi-family fault localization tool for Python programs—using which, we conducted the empirical study presented in Chapter 5.

An interesting next step to continue our research is to integrate these contributions to investigate automated program repair (APR) [44, 82] of NN bugs. As we repeatedly mentioned [53, 57], the type of bugs found in NN programs are different in nature compared to traditional programs. On the other hand, the fix patterns of these bugs are also different from those found in other programs [58], which makes general-purpose APR techniques ineffective on them.

While some recent studies attempted to provide fix suggestions [124] or even totally fix [141] *structural bugs* occurring at the training phase of NN programs, there exist evidence [53, 57] indicating that NN programs contain other types of bugs along with structural bugs. These other bug types can be the target of a new APR study. We can use FauxPy and the empirical knowledge from Chapter 5 for the fault localization activity and our curated dataset of NN bugs, as well as bugs from other frameworks like BugsInPy, as subjects for APR research. Furthermore, we can use information such as the architecture of models to provide fix suggestions for the targeted bug types.

7.3 Closing Remarks

We started this dissertation with the goal of enhancing our understanding about the capabilities and limitations of standard test generation and fault localization techniques on data science programs to provide foundational knowledge for the development of more effective debugging techniques tailored for data science programs. To this end, we started by investigating how standard test generation techniques work on neural network (NN) programs. Upon identifying the limitations of standard techniques on NN programs, we designed ANNoTest, a novel test generation technique tailored for NN programs in Python.

We also conducted an extensive empirical study of fault localization in Python programs as some of the debugging challenges posed by data science programs are rooted in Python as a dynamically typed language. Additionally, we introduced two open-source tools ANNoTest and FauxPy as well as a

curated collection of NN bugs. Along with listing several potential avenues for researchers to extend this dissertation, we hope our results will be beneficial for both researchers and practitioners in the domain.

Bibliography

[1] Keras adversarial models. https://github.com/bstriner/keras-adversarial/.

[2] *Modeling behind lockdown was an unreliable buggy mess, claim experts*, (accessed October 22, 2020). https://www.wandisco.com/storage/app/media/documents/articles/Sunday_Telegraph_051720.pdf.

[3] *Some Recent Software Failures Caused by Software Bugs!*, (accessed October 22, 2020). http://www.sereferences.com/software-failure-list.php.

[4] *PEP 484 – Type Hints*, (accessed October 8, 2020). https://www.python.org/dev/peps/pep-0484/.

[5] *The Top Programming Languages 2019*, (accessed October 8, 2020). https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019/.

[6] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[7] B. K. Aichernig. Automated black-box testing with abstract vdm oracle. In *Computer Safety, Reliability and Security*, pages 250–259, 1999.

[8] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[9] P. Ammann and J. Offutt. *Introduction to Software Testing Edition 2*. Cambridge University Press, New York, NY, 2017.

[10] V. Amrehin, S. Greenland, and B. McShane. Scientists rise up against statistical significance. *Nature*, 567:305–307, 2019.

[11] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[12] W. Araujo, L. C. Briand, and Y. Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In *International Symposium on Empirical Software Engineering and Measurement*, pages 10–19, 2011.

[13] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177—-188, 2016.

[14] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[15] N. Batchelder. Coverage.py. `https://coverage.readthedocs.io/`, 2023. [Online; accessed 6-April-2023].

[16] G. Bell, T. Hey, and A. Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[17] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 308–318, 2008.

[18] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. ISSTA 2018, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery.

[19] I. BoussaïD, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[20] G. Candea and P. Godefroid. Automated software test generation: Some challenges, solutions, and recent advances. In B. Steffen and G. J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 505–531. Springer, 2019.

[21] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning affordance for direct perception in autonomous driving. In *IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[22] C. Chen and N. Wang. Unitfl: A fault localization tool integrated with unit test. In *Proc. ICCSNT*, pages 136–142, 2016.

[23] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. C. Chu, and B. Xu. Dynamic slicing of Python programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 219–228, 2014.

[24] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[25] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[26] D. Coppit and J. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *29th Annual IEEE/NASA Software Engineering Workshop*, pages 305–314, 2005.

[27] Cosmic Ray: mutation testing for Python. `https://cosmic-ray.readthedocs.io/`, 2019. [Online; accessed 6-April-2023].

[28] W. W. Daniel. *Biostatistics: A Foundation for Analysis in the Health Sciences*. Wiley, 7 edition, 1999.

[29] Deal: A Python library for design by contract. `https://github.com/life4/deal`, 2018.

[30] V. Debroy, W. E. Wong, X. Xu, and B. Choi. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 13–22, 2010.

[31] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[32] A. Derezińska and K. Hałas. Analysis of mutation operators for the Python language. In W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, editors, *Proceedings of the 9th International Conference on Dependability and Complex Systems*, pages 155–164, 2014.

[33] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 245–257. ACM, 2003.

[34] J. Ding, X. Kang, and X.-H. Hu. Validating a deep learning framework by metamorphic testing. In *IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 28–34, 2017.

[35] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[36] T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, and A. Cosar. A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering*, 137:106040, 2019.

[37] H. F. Eniser, S. Gerasimou, and A. Sen. Deepfault: Fault localization for deep neural networks. In R. Hähnle and W. van der Aalst, editors, *Fundamental Approaches to Software Engineering*, pages 171–191, Cham, 2019. Springer International Publishing.

[38] N. Ferguson, D. Laydon, G. Nedjati-Gilani, N. Imai, K. Ainslie, M. Baguelin, S. Bhatia, A. Boonyasiri, Z. Cucunubá, G. Cuomo-Dannenburg, et al. Report 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID19 mortality and healthcare demand. Technical report, Imperial College London, 2020.

[39] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.

[40] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empir. Softw. Eng.*, 20(3):611–639, 2015.

[41] C. A. Furia, R. Feldt, and R. Torkar. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, 47(9):1786–1810, September 2021.

[42] C. A. Furia, R. Torkar, and R. Feldt. Applying Bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality. *ACM Transactions on Software Engineering and Methodology*, 31(3):40:1–40:38, July 2022.

[43] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Transactions on Software Engineering*, 41(10):1019–1037, October 2015.

[44] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[45] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, pages 2672–2680, 2014.

[46] S. Gossett. *12 Data Science Programming Languages to Know*, (accessed May 16, 2024). `https://builtin.com/data-science/data-science-programming-languages`.

[47] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–498, 2020.

[48] C. Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, Nov. 2008.

[49] T. Hey, A. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-intensive Scientific Discovery*. Microsoft Research, 2009.

[50] F. Horváth, A. Beszédes, B. Vancsics, G. Balogh, L. Vidács, and T. Gyimóthy. Experiments with interactive fault localization using simulated and real users. In *Proc. ICSME*, pages 290–300, 2020.

[51] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161, 2019.

[52] G. Huang, Z. Liu, G. Pleiss, L. Van Der Maaten, and K. Weinberger. Convolutional networks with dense connectivity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[53] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.

[54] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994.

[55] Q. Idrees Sarhan, A. Szatmári, R. Tóth, and A. Beszédes. CharmFL: A fault localization tool for Python. In *IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 114–119, 2021.

[56] M. J. Islam, S. Ahmad, F. Haque, M. B. I. Reaz, M. A. S. Bhuiyan, and M. R. Islam. Application of min-max normalization on subject-invariant emg pattern recognition. *IEEE Transactions on Instrumentation and Measurement*, 71:1–12, 2022.

[57] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019.

[58] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1135–1146, 2020.

[59] T. Janssen, R. Abreu, and A. J. van Gemund. Zoltar: a spectrum-based fault localization tool. In *Proc. SINTER*, pages 23–30, 2009.

[60] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[61] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[62] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 433–436, 2014.

[63] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[64] R. Just, D. Jalali, and other Defects4J contributors. Defects4J repository. `https://github.com/rjust/defects4j#export-version-specific-properties`, 2023.

[65] N. J. Juzgado and O. S. Gómez. Replication of software engineering experiments. In *Empirical Software Engineering and Verification – International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, volume 7007 of *Lecture Notes in Computer Science*, pages 60–88. Springer, 2010.

[66] M. Kessel and C. Atkinson. Diversity-driven unit test generation. *Journal of Systems and Software*, 193:111442, 2022.

[67] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 301–310. ACM, 2008.

[68] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011.

[69] T.-D. B. Le, F. Thung, and D. Lo. Theory and practice, do they match? a case with spectrum-based fault localization. In *IEEE International Conference on Software Maintenance*, pages 380–383, 2013.

[70] S. Lee, S. Cha, D. Lee, and H. Oh. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proc. ISSTA*, pages 165–176, 2020.

[71] L. Li, J. Wang, and H. Quan. Scalpel: The Python static analysis framework. *arXiv preprint arXiv:2202.11840*, 2022.

[72] X. Li, W. Li, Y. Zhang, and L. Zhang. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.

[73] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):1–30, 2017. Replication package: `https://github.com/deeprl4fl2021icse/deeprl4fl-2021-icse`.

[74] Y. Li, S. Wang, and T. N. Nguyen. Fault localization with code coverage representation learning. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 661–673, 2021.

[75] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. van der Laak, B. van Ginneken, and C. I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, 2017.

[76] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[77] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.

[78] S. Lukasczyk, F. Kroiß, and G. Fraser. Automated unit test generation for python. In *Proceedings of the 12th Symposium on Search-based Software Engineering*, pages 9–24, 2020.

[79] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[80] D. MacIver, Z. Hatfield-Dodds, and M. Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[81] S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[82] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), 2018.

[83] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014.

[84] S. Mukherjee, A. Almanza, and C. Rubio-González. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 439–451, 2021.

[85] M. Nejadgholi and J. Yang. A study of oracle approximations in testing deep learning libraries. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 785–796, 2019.

[86] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996.

[87] R. A. Oliveira, U. Kanewala, and P. A. Nardi. Chapter three - automated test oracles: State of the art, taxonomies, and trends. volume 95 of *Advances in Computers*, pages 113–199. Elsevier, 2014.

[88] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering Proceedings*, pages 117–132, 2014.

[89] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering*, pages 75–84, 2007.

[90] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. 2019.

[91] M. Papadakis and Y. Le Traon. Metallaxis-fl: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5–7):605–628, 2015.

[92] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In M. B. Dwyer and F. Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 199–209. ACM, 2011.

[93] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[94] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620, 2017.

[95] M. Pezzè and C. Zhang. Chapter one - automated test oracles: A survey. volume 95 of *Advances in Computers*, pages 1–48. Elsevier, 2014.

[96] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 322–331, 2011.

[97] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[98] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[99] M. Rezaalipour and C. A. Furia. An annotation-based approach for finding bugs in neural network programs. *Journal of Systems and Software*, 201:111669, 2023.

[100] M. Rezaalipour and C. A. Furia. aNNoTest: An annotation-based test generation tool for neural network programs. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 574–579, 2023.

[101] M. Rezaalipour and C. A. Furia. An empirical study of fault localization in Python programs. *Empirical Software Engineering*, 2024. Accepted in March 2024.

[102] M. Rezaalipour and C. A. Furia. Fauxpy: A fault localization tool for python, 2024.

[103] H. L. Ribeiro, R. P. A. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon. Jaguar: A spectrum-based fault localization tool for real-world software. In *Proc. ICST*, pages 404–409, 2018.

[104] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella. Testing machine learning based systems: a systematic mapping. *Empir. Softw. Eng.*, 25(6):5193–5254, 2020.

[105] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Should we really be using *t*-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, 2006.

[106] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 224–234. ACM, 2018.

[107] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[108] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos. PyCG: Practical call graph generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 1646–1657, 2021.

[109] Q. I. Sarhan and A. Beszédes. A survey of challenges in spectrum-based software fault localization. *IEEE Access*, 10:10618–10639, 2022.

[110] E. Schoop, F. Huang, and B. Hartmann. Umlaut: Debugging deep learning programs using program structure and model behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021.

[111] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121, 2010.

[112] R. Schwitter. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pages 228–232, 2002.

[113] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *Fourth International Conference on Software Engineering Advances*, pages 140–145, 2009.

[114] W. Shen, J. Wan, and Z. Chen. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 108–115, 2018.

[115] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In R. Oliveto, M. D. Penta, and D. C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 130–140. IEEE Computer Society, 2018.

[116] J. Sohn, S. Kang, and S. Yoo. Search based repair of deep neural networks, 2019.

[117] J. Sohn and S. Yoo. FLUCCS: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2017.

[118] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 314–324, 2013.

[119] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li. An empirical study on real bugs for machine learning programs. In *24th Asia-Pacific Software Engineering Conference*, pages 348–357, 2017.

[120] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore. Structural test coverage criteria for deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 2019.

[121] A. Szatmári, Q. I. Sarhan, and A. Beszédes. Interactive fault localization for Python with CharmFL. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, pages 33–36, 2022.

[122] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280, 2012.

[123] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999.

[124] M. Wardat, B. D. Cruz, W. Le, and H. Rajan. Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *The 44th International Conference on Software Engineering*, 2022.

[125] M. Wardat, W. Le, and H. Rajan. Deeplocalize: Fault localization for deep neural networks. In *ICSE'21: The 43nd International Conference on Software Engineering*, 2021.

[126] R. L. Wasserstein and N. A. Lazar. The ASA statement on *p*-values: Context, process, and purpose. *The American Statistician*, 70(2):129–133, 2016. `https://www.amstat.org/asa/files/pdfs/P-ValueStatement.pdf`.

[127] R. Widyasari, G. A. A. Prana, S. A. Haryono, S. Wang, and D. Lo. Real world projects, real faults: Evaluating spectrum based fault localization techniques on Python projects. *Empirical Softw. Engg.*, 27(6), 2022.

[128] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1556–1560, 2020.

[129] E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *1st International Conference on Software Testing, Verification, and Validation*, pages 42–51, 2008.

[130] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[131] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[132] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006 – Object-Oriented Programming*, pages 380–403, 2006.

[133] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200, 2014.

[134] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, 2012.

[135] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Res. Note RN/14/14, University College London, London, England, 2014.

[136] A. Zeller. *Why Programs Fail – A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[137] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 701–712, 2014.

[138] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1):1–36, 2022.

[139] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1159–1170, 2020.

[140] X. Zhang, R. Gupta, and N. Gupta. Locating faults through automated predicate switching. In *Software Engineering, International Conference on*, pages 272–281, 2006.

[141] X. Zhang, J. Zhai, S. Ma, and C. Shen. Autotrainer: An automatic dnn training problem detection and repair system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 359–371, 2021.

[142] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.

[143] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 826–837, 2020.

[144] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

[145] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2021.

URL References

1. https://github.com/pytorch/vision/issues/5209
2. https://github.com/pytorch/vision/issues/6607
3. DenseNet project page: https://github.com/cmasch/densenet/
4. https://github.com/cmasch/densenet/blob/70ee31d0f6f800324fbe98ea687122395248d39e/densenet.py
5. https://github.com/cmasch/densenet/commit/693d772ae9dcdb4d524b25d7d2f6428de4a524ff#diff-813086a9be01b05b352f0111384c48e74735b009e22f4bab1f3dcaa06e2303c2R68
6. https://github.com/bstriner/keras-adversarial/blob/master/examples/image_utils.py#L34
7. https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/adversarial_utils.py#L10
8. https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/image_grid_callback.py#L7
9. https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/model.py#L21
10. https://github.com/se2p/pynguin/issues/20
11. https://github.com/pytorch/vision/blob/v0.11.2/torchvision/datasets/mnist.py
12. https://github.com/bstriner/keras-adversarial/blob/master/examples/example_rock_paper_scissors.py#L62
13. https://github.com/keras-team/keras/blob/keras-1/keras/layers/core.py#L588
14. https://github.com/keras-team/keras/blob/keras-2/keras/layers/core.py#L823
15. https://github.com/bstriner/keras-adversarial/blob/master/examples/example_aae_cifar10.py#L69-L70
16. https://github.com/bstriner/keras-adversarial/blob/master/examples/example_aae.py#L46-L47
17. https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/h5tool3.py#L500
18. https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/h5tool3.py#L520
19. https://github.com/anastassia-b/neural-algorithm-artistic-style
20. https://github.com/bstriner/keras-adversarial
21. https://github.com/cmasch/densenet
22. https://github.com/csvance/deep-connect-four
23. https://github.com/dhkim0225/keras-image-segmentation
24. https://github.com/dishen12/keras_frcnn
25. https://github.com/heuritech/convnets-keras
26. https://github.com/jamesmf/mnistCRNN
27. https://github.com/javiermzll/Image-Recognition
28. https://github.com/katyprogrammer/regularization-experiment
29. https://github.com/michalgdak/car-recognition
30. https://github.com/naykun/TF_PG_GANS
31. https://github.com/notem/keras-alexnet
32. https://github.com/Spider101/Visual-Semantic-Alignments
33. https://github.com/taashi-s/UNet_Keras

34. https://github.com/yagotome/lstm-ner
35. https://github.com/dennybritz/cnn-text-classification-tf
36. https://github.com/iwyoo/tf_ThinPlateSpline
37. https://github.com/zzsdsgdtc/BiDAF_PyTorch
38. https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL196-L197
39. https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL187
40. https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e03687425040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R51
41. https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e03687425040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R52
42. https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e03687425040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R53
43. https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL35
44. https://github.com/heuritech/convnets-keras/commit/b1b472ccf59bfc3edb7ad033299875c905bf8e37#diff-4a9f068fbd6ab76d347ca7772f3da3f100db338cd6c8fb3900adef38ab9dff20L325
45. https://github.com/notem/keras-alexnet/commit/94638c596ca6f3f474241e8a058fd893e1f5ffaa#diff-23de837fc8b40e270ddb47d0ae913f55e8d31635b80daa5618273535b9d3cd28L198
46. https://github.com/dishen12/keras_frcnn/commit/d91c0adc5ccd34f6e346fdeddc0a2ce7085a4ffb#diff-a3429d56d560ec95c6b119754a121d183b32f8a4b73786f8760d083353914efbL18
47. https://github.com/dhkim0225/keras-image-segmentation/commit/992685cde39c3d53ea881d22b9cb26e84963d4bb#diff-d0ff8417443a18c35cc6c3183197d82f48cee72d735133ff901da033d0e32242L89
48. https://github.com/taashi-s/UNet_Keras/commit/b1b6d938bdd7a3e30f3d1fa58009f4850cbc2958#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800L31
49. https://github.com/taashi-s/UNet_Keras/commit/b1b6d938bdd7a3e30f3d1fa58009f4850cbc2958#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800L35
50. https://github.com/javiermzll/CCN-Whale-Recognition/commit/e2d3ff925460060f0127c894368147b54b5f03c0#diff-1b740140b6c82aacc5a6f6b319be9cf103ee72b424ad475f795ea72d4b267849L46
51. https://github.com/javiermzll/CCN-Whale-Recognition/commit/e2d3ff925460060f0127c894368147b54b5f03c0#diff-1b740140b6c82aacc5a6f6b319be9cf103ee72b424ad475f795ea72d4b267849L46
52. https://github.com/pytorch/vision/pull/5238
53. https://github.com/pytorch/vision/blob/v0.11.2/torchvision/models/detection/backbone_utils.py#L49
54. https://github.com/pytorch/vision/blob/v0.11.2/test/test_backbone_utils.py#L25
55. https://github.com/pytorch/vision/blob/main/torchvision/io/image.py#L127
56. https://github.com/pytorch/vision/issues/6607
57. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/models/detection/backbone_utils.py#L44
58. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/_video_opt.py#L265
59. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/image.py#L160
60. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/test/test_image.py#L382
61. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/image.py#L12
62. https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/_video_opt.py#L14