UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2023-038
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 03 May 2023 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Aayush GARG
Born on 30 September 1988 in Meerut (India)

GUIDING QUALITY ASSURANCE THROUGH CONTEXT AWARE LEARNING

Dissertation defence committee

Dr Yves LE TRAON, Dissertation Supervisor
Professor, Université du Luxembourg

Dr Serge DEMEYER
Professor, Universiteit Antwerpen

Dr Michail PAPADAKIS, Chairman
Associate Professor, Université du Luxembourg

Dr Xavier DEVROEY
Assistant Professor, Université de Namur

Dr Renzo DEGIOVANNI, Vice-Chairman
Research Scientist, Université du Luxembourg

Abstract

Software Testing is a quality control activity that, in addition to finding flaws or bugs, provides confidence in the software's correctness. The quality of the developed software depends on the strength of its test suite. Mutation Testing has shown that it effectively guides in improving the test suite's strength. Mutation is a test adequacy criterion in which test requirements are represented by mutants. Mutants are slight syntactic modifications of the original program that aim to introduce semantic deviations (from the original program) necessitating the testers to design tests to kill these mutants, i.e., to distinguish the observable behavior between a mutant and the original program. This process of designing tests to kill a mutant is iteratively performed for the entire mutant set, which results in augmenting the test suite, hence improving its strength.

Although mutation testing is empirically validated, a key issue is that its application is expensive due to the large number of low-utility mutants that it introduces. Some mutants cannot be even killed as they are functionally equivalent to the original program. To reduce the application cost, it is imperative to limit the number of mutants to those that are actually useful. Since it requires manual analysis and test executions to identify such mutants, there is a lack of an effective solution to the problem. Hence, it remains unclear how to mutate and test a code efficiently.

On the other hand, with the advancement in deep learning, several works in the literature recently focused on using it on source code to automate many non-trivial tasks including bug fixing, producing code comments, code completion, and program repair. The increasing utilization of deep learning is due to a combination of factors. The first is the vast availability of data to learn from, specifically source code in open-source repositories. The second is the availability of inexpensive hardware able to efficiently run deep learning infrastructures. The third and the most compelling is its ability to automatically learn the categorization of data by learning the code context through its hidden layer architecture, making it especially proficient in identifying features. Thus, we explore the possibility of employing deep learning to identify only useful mutants, in order to achieve a good trade-off between the invested effort and test effectiveness.

Hence, as our first contribution, this dissertation proposes Cerebro, a deep learning approach to statically select subsuming mutants based on the mutants' surrounding code context. As subsuming mutants reside at the top of the subsumption hierarchy, test cases designed to only kill this minimal subset of mutants kill all the remaining mutants. Our evaluation of Cerebro demonstrates that it preserves the mutation testing benefits while limiting the application cost, i.e., reducing all cost factors such as equivalent mutants, mutant executions, and the mutants requiring analysis.

Apart from improving test suite strength, mutation testing has been proven useful in inferring software specifications. Software specifications aim at describing the software's intended behavior and can be used to distinguish correct from incorrect software behaviors. Specification inference techniques aim at inferring assertions by generating and filtering candidate assertions through dynamic test executions and mutation testing. Due to the introduction of a large number of mutants during mutation testing such techniques are also computationally expensive, hence establishing a need for the selection of mutants that fit best for assertion inference. We refer to such mutants as Assertion Inferring Mutants. In our analysis, we find that the assertion inferring mutants are significantly different from the subsuming mutants. Thus, we explored the employability of deep learning to identify Assertion Inferring Mutants. Hence, as our second contribution, this dissertation proposes Seeker, a deep learning approach to statically select Assertion Inferring Mutants. Our evaluation demonstrates that Seeker enables an assertion inference capability comparable to the full mutation analysis while significantly limiting the execution cost.

In addition to testing software in general, a few works in the literature attempt to employ mutation testing to tackle security-related issues, due to the fault-based nature of the technique. These works propose mutation operators to convert non-vulnerable code to vulnerable by mimicking common security bugs. However, these pattern-based approaches have two major limitations. Firstly, the design of security-specific mutation operators is not trivial. It requires manual analysis and comprehension of the vulnerability classes. Secondly, these mutation operators can alter the program semantics in a manner that is not convincing for developers and is perceived as unrealistic, thereby hindering the usability of the method.

On the other hand, with the release of powerful language models trained on large code corpus, e.g. CodeBERT, a new family of mutation testing tools has arisen with the promise to generate natural mutants. We study the extent to which the mutants produced by language models can semantically mimic the behavior of vulnerabilities aka Vulnerability-mimicking Mutants. Designed test cases failed by these mutants will also tackle mimicked vulnerabilities. In our analysis, we found that a very small subset of mutants is vulnerability-mimicking. Though,

this set mimics more than half of the vulnerabilities in our dataset. Due to the absence of any defined features to identify vulnerability-mimicking mutants, as our third contribution, this dissertation introduces Mystique, a deep learning approach that automatically extracts features to identify vulnerability-mimicking mutants. Despite the scarcity, Mystique predicts vulnerability-mimicking mutants with a high prediction performance, demonstrating that their features can be automatically learned by deep learning models to statically predict these without the need of investing any effort in defining features.

Since our vulnerability-mimicking mutants cannot mimic all the vulnerabilities, we perceive that these mutants are not a complete representation of all the vulnerabilities and there exists a need for actual vulnerability prediction approaches. Although there exist many such approaches in the literature, their performance is limited due to a few factors. Firstly, vulnerabilities are fewer in comparison to software bugs, limiting the information one can learn from, which affects the prediction performance. Secondly, the existing approaches learn on both, vulnerable, and supposedly non-vulnerable components. This introduces an unavoidable noise in training data, i.e., components with no reported vulnerability are considered non-vulnerable during training, and hence, results in existing approaches performing poorly. We employed deep learning to automatically capture features related to vulnerabilities and explored if we can avoid learning on supposedly non-vulnerable components. Hence, as our final contribution, this dissertation proposes TROVON, a deep learning approach that learns only on components known to be vulnerable, thereby making no assumptions and bypassing the key problem faced by previous techniques. Our comparison of TROVON with existing techniques on security-critical open-source systems with historical vulnerabilities reported in the National Vulnerability Database (NVD) demonstrates that its prediction capability significantly outperforms the existing techniques.

Acknowledgments

I would like to take this opportunity to express my heartfelt gratitude to all those who have supported me throughout my Ph.D. studies. Their guidance, encouragement, and unwavering support have been instrumental in my success.

I am deeply indebted to my supervisor, Prof. Yves Le Traon, for his invaluable guidance, expertise, and support throughout my research. His insights and feedback have been invaluable in shaping my work.

I extend my sincere appreciation to my mentors, Dr. Mike Papadakis, and Dr. Maxime Cordy, for their constant support, encouragement, and expert advice. Their insights and feedback have been instrumental in improving the quality of my research.

My mentor and close colleague, Dr. Renzo Degiovanni, deserves a special mention for his insightful discussions, collaborative efforts, and invaluable support throughout my research.

I would like to acknowledge the contributions of my co-authors, who have played a vital role in the success of my research. Their feedback and suggestions have helped me improve the quality of my work.

I thank the members of the jury for their time, effort, and critical feedback, which helped me improve the quality of my research.

I express my gratitude to my colleagues at SERVAL, SnT for their support, encouragement, and camaraderie throughout my Ph.D. journey.

I would like to dedicate this work to my spouse, Mrs. Aparna Upadhayay, and my son, Ansh Garg, who have been my constant source of inspiration and motivation. Their love, support, and understanding have been instrumental in my success.

Finally, I am thankful to my father, Dr. Vinod Garg, and my mother, Mrs. Latesh Garg, for their unwavering support, encouragement, and belief in me. Their sacrifices, guidance, and love have been pivotal in shaping me into the person I am today.

Aayush Garg
Luxembourg, May 2023

Contents

1

Introduction

This Chapter presents the context, the challenges addressed, and the solutions proposed by us as our contributions in this dissertation. Firstly, the general principles of software testing especially mutation testing are introduced followed by the presentation of challenges faced by mutation testing in practice. Then, we present our proposed solution employing machine learning to address these challenges. Next, we present the applications of mutation testing with their faced challenges followed by our proposed solutions employing machine learning to mitigate these challenges. Next, the limitation of mutation testing in one of its applications, i.e., security testing, in tackling vulnerabilities, is discussed followed by our proposed solution. Finally, an overview of our four contributions in this dissertation is presented followed by the organization of the rest of the dissertation.

Contents

1.1 Context

The focus of this dissertation is on guiding software functional white-box testing through machine learning. White-box testing enables the evaluation of a program's internal behavior and logic due to the accessibility of the internal code and structure of the software being tested. This enables the evaluation through the design of tests that exercise different parts of the code and verify specific functionalities of the program. The tests are written to check for expected behavior and to ensure the correctness of the program under test.

Software systems have become increasingly complex in the past few decades, with larger code size and more interactions between software modules. As a consequence, the cost of software testing, including mutation testing, has risen. Mutation testing, a fault-based testing technique that employs artificial faults or mutants to develop tests, is highly effective. However, the number of mutants increases with larger code, which presents significant challenges for mutation testing and its applications in software engineering and security testing due to the high application cost involved.

On the other hand, the advancement in machine learning continues to improve its ability to learn the categorization of data and make it further proficient in identifying features automatically. This is promising in identifying useful mutants to achieve a good trade-off between the invested effort and the effectiveness gained.

1.1.1 Software Testing

Software testing is an essential process in software development that aims to ensure that software is reliable and meets intended requirements. A key aspect of software testing is designing effective test cases to find faults in the developed software and give confidence in its correctness by validating the developed software's behavior. Researchers have developed various techniques for generating test cases and found that the effectiveness of software testing can be improved, as demonstrated in studies by Li et al. [LZT+19], Padhye et al. [PLS+19], and Fraser et al. [FA11].

White-box testing is a critical testing technique that evaluates the internal workings of software, including its code and logic. It can be used to identify defects that may not be apparent through black-box testing. Researchers have proposed various white-box testing techniques, including code coverage analysis, data-flow analysis, and mutation testing. These techniques have been found to improve the effectiveness of software testing, as demonstrated by many empirical studies [CPT+17; ABL+06; AO08a; Fra00].

Testers typically follow a systematic approach to test the developed software. It involves, among others, the creation of tests (test suite), the execution of the created tests against the developed software, and the observation of the program

behavior during the tests' execution in order to determine its correctness. One key part of software testing is the creation (design) of the test suites. The quality of software testing depends on the quality of the created test suites.

Designing effective test cases is a critical part of the software testing process. The goal is to create a set of test cases that thoroughly exercise the software system, uncover defects, and validate that the system meets its functional and non-functional requirements. Effective test case design is a complex task that requires careful consideration of various factors such as the system requirements, the input and output data, the functionality being tested, and the testing technique used.

The metrics used to evaluate the thoroughness and quality of a set of test cases in software testing are known as Test adequacy criteria [ZHM97] (TAC). The goal of test adequacy criteria is to assess whether the test cases are effective in revealing faults or defects in the software system. Typically, *Coverage metrics* are used to define test adequacy criteria, which measure the degree to which test cases *cover* various aspects of the software system.

Research studies have investigated the effectiveness of different TACs in detecting faults in software systems. For example, Hierons et al. [HLZ12] evaluated the effectiveness of different TACs in detecting faults in a large number of real-world software systems. Their study found that different TACs have varying effectiveness in detecting different types of faults in software systems. Most TACs are based on the structure of the program. A few examples are statement coverage, branch coverage, and path coverage. However, in 1971, Richard Lipton proposed a different TAC based on artificial faults and named it *Mutation* [Lip71]. Many studies in academia [CPT+17; MKK17; JH11; KFZ+19] and in the industry [SW09; BWB+21; PI18a] show that software practitioners find *Mutation* effective to uncover faults in their software.

1.1.2 Mutation Testing

Mutation testing is a fault-based testing technique that has been widely used as a test adequacy criteria (TAC) for evaluating the quality of software testing. This approach involves creating a set of artificial faults, known as *mutants*, in the software program and then checking whether the test cases can detect these faults. Mutation testing has been shown to be an effective approach for evaluating the adequacy of test cases, as it can identify weaknesses in the test suite that other criteria may not detect.

Many studies have investigated the effectiveness of mutation testing in either white-box or black-box settings. Papadakis et al. [PHT14] injected mutants on the input models of programs (used to perform model-based testing) and demonstrated that they provide stronger correlation to actual code-related faults than using other model-based criteria such as combinatorial interaction testing. Smith and

Williams [SW09] found that mutation analysis was useful in identifying additional faults in a software program that were not found using other criteria. In another study, Petrovic et al. [PI18a] reported that mutation testing was effective in uncovering faults in a real-world industrial application, and provided insights on how to further improve the technique. A study by Chekam et al. [CPT+17] found that mutation testing identified significantly more faults than traditional criteria such as branch coverage and statement coverage. Another advantage of mutation testing is that it provides a measure of the quality of test cases by quantifying the percentage of mutants that are killed by the test cases. This measure can be used to improve the overall quality of the test suite.

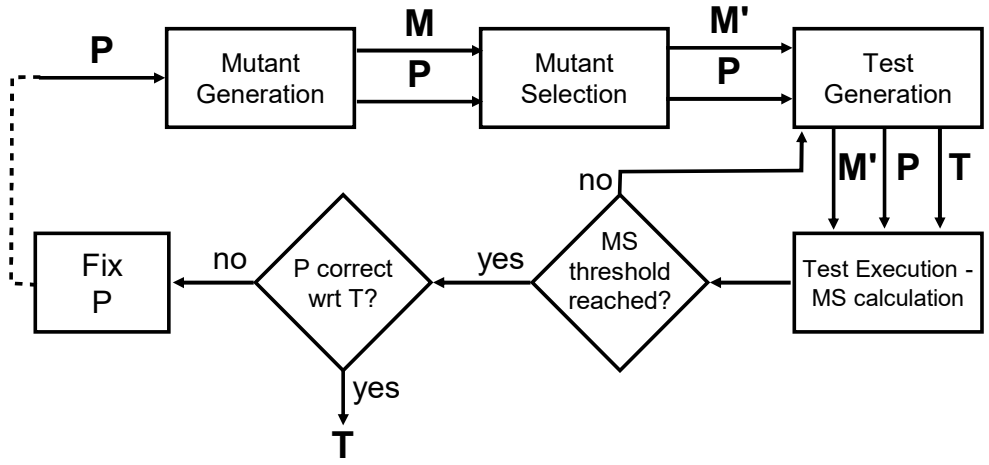
Mutation as a TAC has requirements represented by mutants (a.k.a. artificial bugs). Mutants can be obtained by performing slight syntactic modifications to the original program. For instance, for a program with a statement such as `diff = a - b`, a mutant can be created by replacing the operator '-' with '+'. This is an example of the arithmetic operator replacement mutation operator, which is designed to mimic a common programming mistake. The program mutant created by applying this operator where a single instance of '-' is replaced with '+' is known as called *first order mutation* and the mutants are called *first order mutants*. In contrast to first order mutation, when two or more simple syntactic changes are simultaneously induced into the program under test, the mutation is called *higher order mutation* and the mutants are called *higher order mutants*. The focus of this dissertation is on first order mutants and we use the terms *mutant* and *mutation* referring to first order mutant and first order mutation, respectively.

The next step is to evaluate the generated mutants using the test suite. If the test execution on the original program differs (at the output) from the test execution on the mutant, we say that the mutant is *killed* by the test (the test requirement is fulfilled). Otherwise, the mutant *survives*, indicating that the test suite is not adequate and should be improved.

1.1.3 Mutation Testing in practice

Figure 1.1 presents an overview of how the testing process is performed when it is guided by mutation. We adapted this figure from the one published in [AO08b, Figure 5.2]. Given a program P as input, the mutation testing process starts by creating a set M of mutants forming the test requirements. Test requirements are satisfied when tests kill the mutants. Since the number of mutants is excessive and forms the key cost factor of mutation testing [PKZ+19], testers select a subset M' of mutants from M to focus on their analysis. Then, testers pick a mutant $m \in M'$ and design a test t capable of killing m or judge it as equivalent (to the original code) and discard it. The process is repeated until the design of test is capable of killing a predefined ratio of mutants (threshold). Finally, the designed test suite T is used to check the correctness of program P (w.r.t. test suite T). If test suite T

Figure 1.1: Mutation Testing process. Given a program P and a mutant set M, a practitioner selects from M a subset of mutants M' to be used for test generation. Then, M' is used in Test generation, test execution, and mutation score calculation steps.



detects some bug in program P, then P has to be fixed and the same mutation testing procedure can be employed again.

1.2 Challenges of Mutation Testing

Mutation testing has gained popularity as an effective technique to improve test suite quality, but it has not yet become a widely adopted practice in the industry due to the various challenges it poses. One significant challenge faced by mutation testing is its high computational cost. As the size and complexity of the software system grow, the number of mutants generated increases exponentially, making mutation testing computationally expensive. Many studies have reported that despite its effectiveness, mutation testing is computationally expensive, many times prohibitively expensive [OLR+96; JH11; PKZ+19]. It is also requires substantial resources compared to other coverage techniques [ZHM97]. Li et al. [OL17] also highlighted the issue of high computational cost and its impact on the scalability of mutation testing in large-scale software systems.

Another challenge faced by mutation testing is the issue of redundant and equivalent mutants. The presence of redundant mutants leads to a higher number of generated mutants and longer execution times, without adding any additional value to the test suite. Similarly, equivalent mutants have the same behavior as the original program, making them ineffective in detecting faults. Several studies have addressed this challenge and proposed techniques to reduce the number of redundant and equivalent mutants, such as the use of selective mutation operators [OLR+96]

and test execution optimizations [PM11; PM10b; DPP+16] in order to achieve the common goal of reducing the computational cost of mutation testing.

The major cost factor in mutation testing is the mutants that introduce overheads during both, test generation, and test execution, leading to negligible test effectiveness improvements. Therefore, to reduce the mutation testing effort while preserving its effectiveness, it is essential to focus on those mutants that add value by considerably improving the test suites. One such subset of mutants is Subsuming mutants.

Subsuming mutants, also called disjoint or dominator mutants [KPM10; KAO+16], are the minimum subset of all mutants that when killed, by any possible test suite, results in killing the entire set of killable mutants. Given two mutants M_1 and M_2, it is said that M_1 subsumes M_2 if every test suite T killing M_1 also kills M_2. Unfortunately, identifying subsuming mutants is undecidable as it is not possible to know a mutant's behavior under every possible input. Thus, researchers typically approximate them through test suites [ADO14; PHH+16; KAO+16; PCT18]. Since killing subsuming mutants leads to the killing of all killable mutants, testers should focus mutation analysis on subsuming mutants [KPM10; KPM10; ADO14]. The problem though is that one needs to know the subsumption relations between mutants in advance, before starting to analyze the mutants and designing tests.

Machine learning, on the other hand, allows computers to learn from data and make predictions or decisions without being explicitly programmed [Alp04; GBC16]. Machine learning algorithms are designed to identify patterns, relationships, and anomalies in large and complex datasets by automatically detecting features and extracting meaningful insights. It involves training an algorithm on a labeled dataset that can accurately predict the correct output for new, unseen inputs [Alp04]. Hence, we propose *Cerebro* [GOD+22], an approach based on machine learning to statically predict subsuming mutants from given mutant sets.

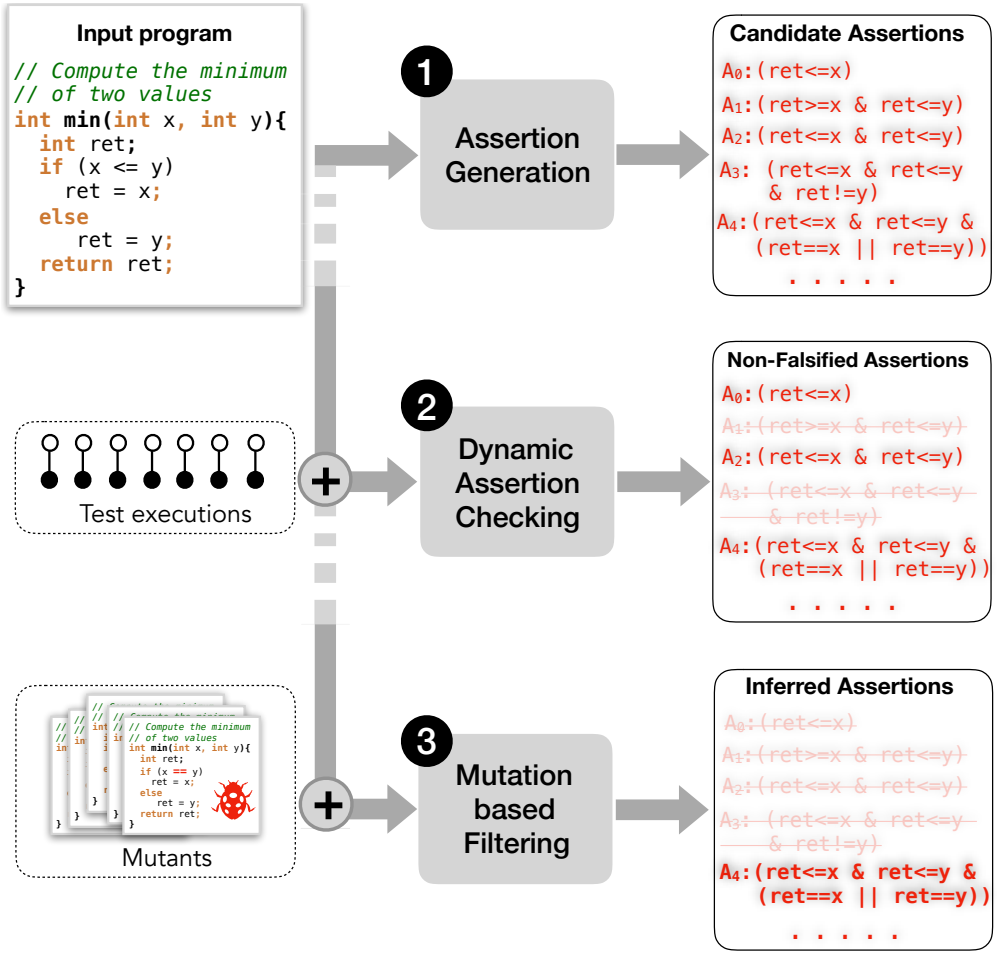
1.3 Applications of Mutation Testing

Mutation testing is a fault-based testing technique that involves introducing small changes (i.e., mutations) into the source code to simulate faults. The goal is to evaluate the effectiveness of the test suite by measuring how many mutants are detected. This fault-based nature of mutation testing enables its application in other software engineering tasks [JH11; PKZ+19], such as specification inference and security testing.

1.3.1 Specification Inference

Software specifications are descriptions of the intended behavior of the software. They are crucial for determining whether software behavior is correct or not. Specifications can be formally expressed as a set of executable constraints/asser-

Figure 1.2: Specification Inference via Dynamic Test Execution and Mutation Analysis.



tions. The specification inference problem consists of automatically generating specifications from existing software artifacts especially source code. Program specifications are composed of a set of (executable) assertions for various program points, such as method preconditions, postconditions, and invariants, that must hold true during the program execution, at the corresponding program points. In this dissertation, we focus on *postcondition* assertions, i.e., assertions that state what are the properties that are expected to hold after a given method is executed.

1.3.2 Role of Mutation in Specification Inference and its associated challenges

Figure 1.2 illustrates the typical process of existing assertion inference techniques [TJT+20; MPA+21; MdA22]. First, the assertion generation step, based in general on a search-based algorithm (e.g. GAssert [TJT+20] and EvoSpex [MPA+21] use evolutionary search algorithms, while SpecFuzzer [MdA22] uses fuzzing), produces a set of candidate assertions for a given program/method. Second, the program's test suite (given as input or automatically generated) is executed to determine which of those assertions are coherent with the behaviors currently exhibited by the program. Lastly, the non-falsified assertions (i.e., those that are coherent with the test suite executions) go through a mutation analysis step for filtering out weak assertions. Here, a non-falsified assertion that is also coherent with all the mutants' execution of a given program, is considered to be weak because it is unable to distinguish between the original and the mutated program behaviors, and is hence discarded. The inferred assertions are the ones that are coherent with the current program behavior but are falsified by the behavior of buggy programs (i.e., they kill at least one mutant).

Despite being effective for discarding weak assertions, mutation analysis suffers from scalability issues due to the large number of mutants that are generated from even a small piece of code. This adversely affects the overall performance and scalability of assertion inference techniques, especially on large subjects.

One way to address the performance and scalability issue faced by assertion inference techniques is to focus mutation analysis on only the mutants that are useful for assertion inference. Though, this information, *i.e.,* which mutants are useful for the task of specification inference, is not known in advance, before starting to employ mutants and executing candidate assertions.

In Figure 1.2, steps 2 and 3 show that the generated *candidate assertions* undergo a two-step filtering process. Assertions that are falsified when running the test suite of a target class C are discarded. Though such filtering alone is not enough as it leaves room for *weak assertions*, i.e, assertions that are trivial to satisfy and would not trigger any error if the target class C had any incorrect behaviour. For instance, a tautology such as $assert(x >= y \mid\mid x <= y)$ is a valid assertion that cannot be falsified, but it is unlikely to be useful.

Such weak assertions are not useful and thus the use of mutation analysis has been proposed to identify and discard them (step 3) [TJT+20; MPA+21; MdA22]. On the contrary, assertions that do not hold for at least one mutant of the target class, are useful because they are capable of distinguishing buggy versions of the code. Given a target class C and a set A of candidate assertions that are consistent with the behavior of C, a mutant C' of C is an *Assertion Inferring Mutant* if at least one assertion in A is able to kill C'.

As specification/assertion inference heavily relies on *Assertion Inferring Mutants*, practitioners should focus mutation analysis on these mutants for the task of specification inference. The problem though is that one needs to know which mutants are assertion inferring before starting to employ mutants and executing candidate assertions. To address this problem, we propose *Seeker* [GDM+23], a machine learning method to predict *Assertion Inferring Mutants* from given mutant sets. *Seeker*'s predictions infer almost all of the total ground truth assertions. Moreover, *Seeker* enables the assertion inference technique *SpecFuzzer* to scale on all our large subjects.

1.3.3 Security Testing

In the field of software development, ensuring security is of utmost importance. To ensure the software system is secure, security testing is essential. It involves evaluating the security controls of the software system and identifying weaknesses that may be exploited by attackers. Security testing can be conducted at different stages of software development, including design, development, testing, and deployment. There are different types of security testing, such as penetration testing, vulnerability scanning, and code review. The primary objective of security testing is to identify and mitigate potential security risks before deploying the software system. By conducting thorough security testing, software developers can minimize the risk of security breaches, data theft, and other cybersecurity incidents.

Common Vulnerability Exposures (CVE) [09] defines a security vulnerability as "*a flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components.*". The inadvertence of a developer or insufficient knowledge of defensive programming usually causes these mistakes. Vulnerabilities in source code are a significant threat to software security, and numerous studies have highlighted their importance. For instance, a study by the software security company, Veracode [Ver20], found that 70% of applications contain at least one security flaw in their source code, which can be exploited by attackers. A study by the Ponemon Institute [Ins20] found that 62% of organizations surveyed had experienced a data breach caused by a vulnerability in their source code.

The consequences of vulnerabilities in source code can be severe, ranging from financial losses to reputational damage. A study by the National Institute of Standards and Technology (NIST) [oST19] found that the average cost of a data breach was 3.86 million USD, with a significant portion of that cost being attributed to the impact on customer trust and reputation.

1.3.4 Role of Mutation in Security Testing and its associated challenges

The issues related to security, especially vulnerabilities have received little attention in the mutation testing literature. As a result, despite its flexibility, mutation testing has not been used as the first line of defense against vulnerabilities. Mutation testing can be used to test the security of a software system by generating mutants that simulate common security vulnerabilities, such as buffer overflows, SQL injection, or cross-site scripting (XSS) [Elb11; dOP17]. For example, a mutant that introduces a SQL injection vulnerability can be created by changing a database query in the source code. If the test suite does not detect this mutant, it suggests that the system is vulnerable to SQL injection attacks and needs to be improved.

Mutation testing can also be used to evaluate the effectiveness of security testing tools, such as vulnerability scanners or penetration testing frameworks [LDP+17; BGV10]. By generating mutants that simulate known vulnerabilities, developers can check if these tools can detect the mutants and identify the corresponding vulnerabilities. This can help to improve the quality of security testing tools and ensure that they are effective in detecting real-world vulnerabilities.

With the release of powerful language models trained on large code corpus, e.g. *CodeBERT*, a new family of mutation testing tools has arisen with the promise to generate natural mutants. One such example is *μBERT* [DP22], a mutation testing tool that uses *CodeBERT* to generate mutants by masking and replacing tokens. *μBERT* takes a Java class and extracts the expressions to mutate. It then masks the token of interest, e.g. a variable name, and invokes CodeBERT to complete the masked sequence (i.e., to predict the missing token). It is interesting to study the extent to which the mutants produced by *μBERT* can semantically mimic the behavior of vulnerabilities aka vulnerability-mimicking mutants. Designed test cases failed by these mutants will also tackle mimicked vulnerabilities.

In the existing literature, there is no clear definition of *Vulnerability-mimicking Mutants*, (i.e., mutants that mimic the vulnerability behavior) to focus on. Therefore, for the purpose of this dissertation, we define a mutant as vulnerability-mimicking [GDP+23] if it fails exactly the same tests that are failed by the vulnerability it mimics, hence having the same observable behavior as the vulnerability.

Since a mutant is a slight syntactic modification to the original program, a large number of mutants are generated during mutation testing. This introduces the problem of identifying *Vulnerability-mimicking Mutants* among a huge pile of mutants. In our dataset, vulnerability-mimicking mutants are *3.9%* of the entire lot. Also, the labeling information, *i.e.,* which mutant is vulnerability mimicking, is not known in advance. Hence, we proposed *Mystique*, a machine learning method to predict *Vulnerability-mimicking Mutants* from a given mutant's code context. Our experiments show that *Mystique* identified *Vulnerability-mimicking Mutants*

with high prediction performance, which indicates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features.

1.3.5 Limitation of Mutation in Security Testing and Key obstacles in Vulnerability Prediction

Vulnerability prediction is the process of identifying software components, such as files or modules, that are likely to contain vulnerabilities. Though mutation testing is powerful, it is challenging for security testing, especially vulnerability prediction. The reason is that artificially generated mutants cannot mimic all the potential vulnerabilities that may exist in a software system. This also happened during our study where we analyze the extent to which the mutants produced by the language models can semantically mimic vulnerabilities. Hence, we perceive that these mutants are not a complete representation of all the vulnerabilities, and there exists a need for actual vulnerability prediction approaches. These techniques can help identify potential security issues that are not easily detectable through manual inspection or traditional testing methods.

There are many vulnerability prediction approaches in the literature but their performance is limited. Firstly, vulnerabilities are fewer in comparison to software bugs, limiting the information one can learn from, which affects the performance of existing techniques. Secondly, the existing approaches learn on both, vulnerable, and supposedly non-vulnerable components. This introduces an unavoidable noise in training data, i.e., components with no reported vulnerability are considered non-vulnerable during training, and hence, results in existing approaches performing poorly. This establishes a need for robust vulnerability prediction techniques.

Hence, we explore if we can avoid learning on supposedly non-vulnerable components, and we propose *TROVON* [GDJ+22], a deep learning based vulnerability prediction approach that learns only on components known to be vulnerable, thereby making no assumptions and bypassing the key problem faced by previous techniques.

1.4 Overview of the Contribution and Organization of the Dissertation

This section presents the contributions of this dissertation followed by the organization of the remaining chapters.

1.4.1 Contributions

Following are the contributions of this dissertation.

- **Cerebro: Static Subsuming Mutant Selection (Chapter 4).** As our first contribution, in chapter 4, we proposed *Cerebro*, a method that learns to

select subsuming mutants (a subset of mutants that subsumes the others, i.e., tests killing them also kill all the mutants of the given mutant set) from given mutant sets. Our experiments showed that *Cerebro* identified subsuming mutants with high precision and recall at an inter-project scenario (trained on different projects than the ones it was evaluated). These predictions enable testers to design test cases capable of killing more than twice the subsuming mutants that they would kill if they were using either randomly selected mutants or another previously proposed machine learning-based mutant selection technique. At the same time *Cerebro* entails the analysis of significantly fewer equivalent mutants and mutant executions, indicating a large reduction in the practical effort/cost of the approach.

- **Seeker: Efficient Class Specification Inference (Chapter 5).** As our second contribution, in chapter 5, we proposed *Seeker*, a method that learns to select *Assertion Inferring Mutants* (a small subset of mutants that is suitable for assertion inference) from given mutant sets. Our experiments show that *Seeker* identified assertion inferring mutants with high prediction performance. These predictions enable many times faster inference with minor effectiveness loss compared to the use of all mutants. Similarly, *Seeker*'s predictions infer almost all of the total ground truth assertions, which is substantially greater than *Subsuming Mutant Selection* and *Random Mutant Selection*. Moreover, *Seeker* enables the assertion inference technique SpecFuzzer to scale on all our large subjects (by inferring assertions where SpecFuzzer failed previously due to timeouts) in comparison to *Random Mutant Selection* which failed to infer any assertion in half of the cases.

- **Mystique: Enabling Security conscious Mutation Testing using Language Models (Chapter 6).** As our third contribution, in chapter 6, we showed that language model based mutation testing tools can produce *Vulnerability-mimicking Mutants*, i.e., mutants that mimic the observable behavior of vulnerabilities. Since these mutants are significantly fewer among the entire mutant set, there is a need for a static approach to identify such mutants. To achieve this, we proposed *Mystique*, a method that learns to select *Vulnerability-mimicking Mutants* from a given mutant's code context. Our experiments show that *Mystique* identified *Vulnerability-mimicking Mutants* with high prediction performance, which indicates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features.

- **Learning from What We Know: How to Perform Vulnerability**

Prediction using Noisy Historical Data (Chapter 7). As our fourth contribution, in chapter 7, we proposed *TROVON*, a machine translation based approach to automatically learn to predict vulnerable components from noisy historical data. Taking advantage of the large amounts of historical data, our predictions can be used to assist developers in code reviews and security testing. The important advantage of *TROVON* is that it is completely automatic as it learns latent features (context, patterns, etc.) linked with vulnerabilities based on information mining from code repositories (in particular by analyzing historical vulnerability fixes and their context). We empirically evaluated the effectiveness of *TROVON* following the methodological guidelines set by Jimenez et al. [JRP+19]. In particular, we demonstrated that *TROVON* can mitigate the problem of real-world noisy data on the releases of the three security-critical open source systems that were used by previous research. Moreover, we showed that *TROVON* significantly outperforms existing techniques.

1.4.2 Organization of the Dissertation

In the remaining of this dissertation, chapter 2 presents the technical background and definitions used in this dissertation. Chapter 3 discusses the existing work related to the contribution of this dissertation, and presents an overview of the existing literature on mutation testing and its applications especially specification inference and security testing. Chapter 4 presents *Cerebro*, our proposed method that learns to select subsuming mutants from given mutant sets. Chapter 5 presents *Seeker*, our proposed method that learns to select assertion-inferring mutants from given mutant sets. Chapter 6 discusses the ability of language model based mutation testing tools to produce vulnerability-mimicking mutants. It also presents *Mystique*, our proposed method that learns to select such mutants from a given mutant's code context. Chapter 7 presents *TROVON*, our proposed machine learning based vulnerability prediction approach that learns only on components known to be vulnerable, and bypasses the key problem faced by previous techniques. Finally, Chapter 8 concludes this dissertation and presents the future research directions.

2

Technical Background and Definitions

This chapter presents the technical background and definitions used in this dissertation.

Contents

2.1 Mutation Testing

2.1.1 Mutant Selection

Mutation testing is computationally expensive due to the large number of mutants that it introduces, all of which require analysis and execution. To reduce its application cost, it is imperative to limit the number of mutants to those that are actually useful, prior to any manual mutant analysis or test execution. This problem is known as the mutant selection problem [PKZ+19] and has been studied in the form of selective mutation [OLR+96; ZGM+13], i.e., restricting the number of transformations to be used, with limited success [KAO+16; CPB+20]. Though the key issue with mutant selection is the simple syntactic-based nature of the selection process. The problem is that the mutants are introduced everywhere with respect to simple language operators (*e.g.* by replacing an operator with another) that completely ignore the program and particular location semantics. This operator matching mutant selection has the unfortunate effect of introducing mutants independent of their context and program semantics. It is desirable to analyze only the mutants that add value and help in improving the test suite.

2.1.2 Equivalent Mutants

Early research on mutation testing has demonstrated that deciding whether a mutant is equivalent (to the original code) is an undecidable problem [BA82]. Mutation testing may produce a mutant that is syntactically different from the original, yet semantically identical, *aka* equivalent mutant [PJH+15; KPJ+18]. The undecidability of equivalence means that it is impossible to automatically discard them all. As a result, the tester may never know whether he or she has failed to find a killing test case because the mutant is particularly hard to kill, yet remains killable (a 'stubborn' mutant [PCT18]), or whether failure to find a killing test case derives from the fact that the mutant is equivalent. The best options we have are effective algorithms that can remove most equivalent mutants, e.g., in C data-set [CPC+21] authors applied Trivial Compiler Equivalence (TCE) [PJH+15; KPJ+18; HSF+19] to filter out equivalent and duplicated mutants. Interestingly, early research on mutation testing [Acr80] has shown that humans also make many mistakes (approximately 20%) when judging mutants as being equivalent or not. This means that it is unrealistic to expect that automated tools (or testers, in the case of manual test case design) kill all the killable mutants.

To make a fair approximation of killable mutants state-of-the-art test generation tools (KLEE [CDE08], SEMu [CPC+21], and EvoSuite [FZ10]) are used, together with mature developer test suites to identify killable mutants. The remaining live mutants (i.e., mutants killed neither by the developers' written nor automatically generated test suites) are assumed as equivalent. Although this assumption may

have some impact on the results, it allows quantifying the effort involved by testers in analyzing low utility mutants when using the current state-of-the-art advances. We analyze the impact of this assumption on our results in Chapter 4 Section 4.7.4.

2.1.3 Subsuming Mutants

Subsuming mutants are the minimum subset of all mutants that when killed, by any possible test suite, results in killing the entire set of killable mutants. Given two mutants M_1 and M_2, it is said that M_1 subsumes M_2 if every test suite T killing M_1 also kills M_2. Unfortunately, identifying subsuming mutants is undecidable as it is not possible to know a mutant's behavior under every possible input. Thus, researchers typically approximate them through test suites [JH09; ADO14; PHH+16; KAO+16; PCT18].

More precisely, let M_1, M_2 and T be two mutants and a test suite, respectively, where $T_1 \subseteq T$ and $T_2 \subseteq T$ are the set of tests from T that kill mutants M_1 and M_2, respectively, and $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$, indicating that both M_1 and M_2 are killable mutants. We say that mutant M_1 subsumes mutant M_2, if and only if, $T_1 \subseteq T_2$. In case $T_1 = T_2$, we say that mutants M_1 and M_2 are indistinguishable for T. The set of mutants which are both killable, and subsumed only by indistinguishable mutants are called *Subsuming mutants*.

For example, if we have a mutant set of 3 mutants (M_1, M_2, and M_3) and a test set $T = \{t_1, t_2, t_3\}$, where M_1 is killed by $T_1 = \{t_1\}$; M_2 is killed by $T_2 = \{t_1, t_2\}$; and M_3 is killed by $T_3 = \{t_3\}$. We can notice that every time that we run a test (t_1) to kill mutant M_1 we will also kill mutant M_2. However, the opposite does not hold. Thus, in this example, we have two subsuming mutants, i.e., M_1 and M_3.

Interestingly, killing subsuming mutants leads to the killing of all killable mutants, thus, testers need to focus mutation analysis on subsuming mutants [JH09; KPM10; KAD+14; ADO14]. The problem though is that one needs to know the subsumption relations between mutants in advance, before starting to analyze the mutants and designing tests. To deal with this issue, in Chapter 4, we propose a *static* technique that predicts subsuming mutants without requiring any dynamic analysis, with the aim to help testers decide on which mutants to use when performing mutation-guided test generation [PM10b; FZ10]. Given the input program, P and the set M of mutants, our proposed technique selects a subset M' of mutants (mutants predicted as subsuming) to be used for mutation testing, *i.e.,* to guide testers and evaluate test effectiveness. Based on M', testers and/or automatic test generation techniques can focus on the few strong mutants and design effective test cases.

2.1.4 Subsuming Mutation Score (MS*)

Subsuming mutation score (MS*) is the ratio between killed subsuming mutants over the total number of subsuming mutants [PHH+16]. It has been pro-

posed [ADO14; PHH+16; KPM10] as a reliable metric to evaluate the effectiveness of testing techniques as it does not consider the presence of subsumed mutants. Subsumed mutants can artificially inflate the mutation score of a testing technique and can mislead its apparent ability to detect faults. For example, if we have a mutant set of 3 mutants (M_1, M_2, and M_3) and a test set $T = \{t_1, t_2, t_3\}$, where M_1 is killed by $T_1 = \{t_1\}$; M_2 is killed by $T_2 = \{t_1, t_2\}$; and M_3 is killed by $T_3 = \{t_3\}$, we have two subsuming mutants, i.e., M_1 and M_3. A test suite $\{t_1, t_2\}$ kills 66.7% of all the mutants (i.e., M_1 and M_2), but 50% of the subsuming ones (M_3 is not killed).

2.2 Machine Learning

Machine learning is a subset of artificial intelligence that focuses on the development of algorithms that allow computers to learn from data and make predictions or decisions without being explicitly programmed [Alp04; GBC16]. Machine learning algorithms are designed to identify patterns, relationships, and anomalies in large and complex datasets by automatically detecting features and extracting meaningful insights.

Machine learning has a wide range of applications in various fields, including natural language processing, computer vision, speech recognition, recommendation systems, fraud detection, medical diagnosis, and autonomous driving [LBH15; Dom12]. With the growing availability of data and computational resources, machine learning is becoming an increasingly important tool for businesses and organizations to extract insights and make data-driven decisions.

2.2.1 Supervised learning

There are several types of machine learning, including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning [Bis07]. Throughout this dissertation, we focus on supervised learning. Supervised learning is a type of machine learning that involves training an algorithm on a labeled dataset, where the correct output is provided for each input. The goal is to develop a model that can accurately predict the correct output for new, unseen inputs [Alp04].

The process of building a supervised learning model typically involves several steps, including data collection, data preprocessing and cleaning, feature engineering, model selection, training and validation, and testing and evaluation. The first step is to collect a labeled dataset, where each input is associated with a known output. The dataset is then preprocessed and cleaned to ensure that it is ready for analysis. Next, the dataset is split into two subsets: a training set and a validation set. The training set is used to train the model, while the validation set is used to evaluate its performance and tune its parameters.

During training, the model is fitted to the training set by adjusting its parameters to minimize a loss function, which measures the difference between the predicted and actual outputs. The validation set is used to prevent overfitting, which occurs when the model becomes too complex and begins to fit the noise in the training set rather than the underlying patterns [Bis07]. Once the model is trained and validated, it is tested on a separate test set to evaluate its performance on new, unseen data. The goal is to develop a model that generalizes well to new inputs and can accurately predict the correct output.

Supervised learning has a wide range of applications including image classification [KSH12], speech recognition [12], fraud detection [PBC+18], and recommendation systems [KBV09]. The key advantage of supervised learning is that it allows for accurate predictions on new, unseen data, making it a powerful tool for many real-world problems.

2.2.2 Machine Translation

Machine Translation is a type of supervised learning. In machine translation, the goal is to develop a model that can automatically translate text from one language to another [SVL14]. This is typically done by training a machine learning model on a large corpus of parallel texts, where each sentence in one language is paired with its translation in the other language. During training, the model is presented with input sentences in one language and the corresponding output sentences in the other language. The goal is to learn a function that can accurately map the input sentences to the correct output sentences. This is a supervised learning task, since the training data is labeled with the correct translations.

Machine Translation is a transformation function $transform(X) = Y$, where the input $X = \{x_1, x_2, \ldots, x_n\}$ is a set of *entities* that represents a component to be transformed, to produce the output $Y = \{y_1, y_2, \ldots, y_n\}$, which is a set of entities that represent a transformed (desired) component. In the training phase, the transformation function learns on the example pairs (X, Y) available in the training dataset. In our context, X contains the source code with an annotation that indicates the location and type of the mutation operator applied, and Y contains the same information, plus a label that indicates whether the mutant is subsuming or not.

The transformation function is trained to append the label to a given mutant by training the function on the example pairs (Code+MutationAnnotation, Code+MutationAnnotation+Label), where Code+MutationAnnotation represents the source code with an annotation in the statement to indicate the mutation operator type applied. This learned transformation is used as our prediction model. Among the several machine translation algorithms that have been suggested over the past years, we use the RNN Encoder-Decoder which is established and is used by many recent studies [TWB+19a; TWB+19b; SVL14].

2.2.3 RNN Encoder-Decoder architecture

The encoder-decoder architecture for recurrent neural networks is the standard neural machine translation method that rivals and in some cases outperforms classical statistical machine translation methods [Bro18a]. We use the RNN Encoder-Decoder that is established and is used by many recent studies [GDJ+22; SVL14; TWB+19a]. The RNN Encoder-Decoder machine translation is composed of two major components: *RNN Encoder* to encode a sequence of terms x into a vector representation, and *RNN Decoder* to decode the representation into another sequence of terms y. The model learns a conditional distribution over an output sequence conditioned on another input sequence of terms: $P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where n and m may differ. For example, given an input sequence $x = Sequence_{in} = (x_1; \ldots; x_n)$ and a target sequence $y = Sequence_{out} = (y_1; \ldots; y_m)$, the model is trained to learn the conditional distribution: $P(Sequence_{out} | Sequence_{in}) = P(y_1; \ldots; y_m | x_1; \ldots; x_n)$, where x_i and y_j are separated tokens. A bi-directional RNN Encoder [BGL+17], formed by a backward RNN and a forward RNN, is considered the most efficient to create representations as it takes into account both past and future inputs while reading a sequence [BCB14].

2.2.4 Prediction performance metrics

Prediction modeling is a binary classification problem where the data is divided into 2 classes,*i.e.,Positive* class and *Negative* class. The positive class represents the data that we are interested in, and the rest is represented by the negative class [HTF09]. *e.g.* in case of subsuming mutant prediction where a model is trained to predict the mutants that are subsuming, the positive class is represented by the subsuming mutants, and the negative class is represented by the rest, *i.e.,*non-subsuming mutants.

Thus prediction modeling results in four types of outputs: Given a component of the positive class, if it is predicted as positive, then it is a true positive (TP); otherwise, it is a false negative (FN). Given a component of the negative class, if it is predicted as negative, then it is a true negative (TN); otherwise, it is a false positive (FP). From these, we can compute the traditional evaluation metrics such as *Precision*, *Recall*, and *F-measure* scores, which quantitatively evaluate the prediction accuracy of prediction models.

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \quad F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Intuitively, *Precision* indicates the ratio of correctly predicted positives over all the considered positives. *Recall* indicates the ratio of correctly predicted positives over all the actual positives. *F-measure* indicates the weighted harmonic mean of Precision and Recall.

Yet, these metrics do not take into account the true negatives and can be misleading, especially in the case of imbalanced data. Hence, these are complemented with the *Matthews Correlation Coefficient (MCC)* [Mat75], a reliable metric of the quality of prediction models [SBH14a]. It is generally regarded as a balanced measure that can be used even when the classes are of very different sizes, e.g. in a case where typically less than 15% components belong to the positive class whereas the remaining 97% belong to the negative class. *MCC* is calculated as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC returns a coefficient between 1 and -1. An MCC value of 1 indicates a perfect prediction, while a value of -1 indicates a perfect inverse prediction *i.e.,*a total disagreement between prediction and reality. MCC value of 0 indicates that the prediction performance is equivalent to random guessing.

2.3 Applications of Mutation Testing

The fault-based nature of mutation testing enables its application in other software engineering tasks [PKZ+19], such as specification inference and security testing. In specification inference, mutation testing can be used to validate the inferred specification by checking if it can distinguish between the original code and the generated mutants. By doing so, it can help to identify weaknesses in the inferred specification, thereby improving its quality. Similarly, in security testing, mutation testing can be used to simulate common security vulnerabilities by generating mutants. By checking if the tests can detect the generated mutants, developers can identify areas where the system is vulnerable to attacks. This can help to improve the security of the system by addressing the identified vulnerabilities.

2.3.1 Specification Inference

Software specifications are descriptions of the intended behavior of the software. They are crucial for determining whether software behavior is correct or not. The provision of software specifications is strongly related to the oracle problem, i.e., the problem, in the context of software testing, of determining whether the results of program executions are coherent with the desired behavior of the program [BHM+15]. Though specifications are typically expressed informally (e.g., via API documentations), when these are expressed more formally as a set of executable constraints/assertions, they have powerful applications in many software engineering tasks such as software design [Mey97], software testing [AO08a; FZ12], and verification [CR06; GRP+10].

The specification inference problem consists of automatically generating specifications from existing software artifacts, e.g., documentation, source code, program

executions, etc. At the source code level, formal program specifications are typically composed of a set of (executable) assertions for various program points, such as method preconditions, postconditions, and invariants, that must hold true during the program execution, at the corresponding program points. In this dissertation, we focus on *postcondition* assertions, i.e., assertions that state what are the properties that are expected to hold after a given method is executed.

2.3.2 Assertion Inferring Mutants

The generated *candidate assertions* undergo a two-step filtering process. Assertions that are falsified when running the test suite of a target class C are discarded, since these are invalid assertions not satisfying the legit program behavior exhibited by the test suite execution. Though important to identify *valid assertions*, such filtering is not enough as it leaves room for *weak assertions*, i.e, assertions that are trivial to satisfy and would not trigger any error if the target class C had any incorrect behaviour. For instance, a tautology such as $assert(x >= y \mid\mid x <= y)$ is a valid assertion that cannot be falsified, but it is unlikely to be useful. In the case of SpecFuzzer [MdA22], the fuzzer reports thousands of constraints (i.e., candidate assertions), and only a few are falsified by the test suite.

Such weak assertions are not useful and thus the use of mutation analysis has been proposed to identify and discard them [TJT+20; MPA+21; MdA22]. The underlying idea is that valid assertions that are also coherent with every mutant's execution of target class C are weak because they represent properties that hold also for buggy versions of C (the mutants). On the contrary, assertions that do not hold for at least one mutant of C, are useful because they are capable of distinguishing buggy versions of the code. Given a target class C and a set A of candidate assertions that are consistent with the behavior of C, a mutant C' of C is called *assertion inferring* if at least one assertion in A is able to kill of C'.

Despite being effective for discarding weak assertions, mutation analysis suffers from scalability issues due to the large number of mutants that are generated from even a small piece of code. This adversely affects the overall performance and scalability of assertion inference techniques, especially on large subjects. Therefore, for the task of specification inference, a practitioner should focus mutation analysis on *Assertion Inferring Mutants*. Though the problem is that this information, *i.e.,* which mutants are assertion inferring, is not known in advance, before starting to employ mutants and executing candidate assertions. To deal with this issue, in Chapter 5, we propose a learning-based approach to effectively identify *Assertion Inferring Mutants* to improve the performance and scalability of assertion inference techniques.

2.3.3 Security Testing and Vulnerabilities

Security testing is an essential component of software development, particularly in today's ever-evolving threat landscape. With cyber attacks becoming increasingly sophisticated and frequent, it is crucial that software systems are designed and tested to be as secure as possible. Security testing involves evaluating the security controls of a software system and identifying weaknesses that could be exploited by attackers. Security testing can be conducted at various stages of the software development lifecycle, including design, development, testing, and deployment. This ensures that security is built into the software system from the very beginning and helps to minimize the risk of security breaches, data theft, and other cybersecurity incidents.

There are different types of security testing that can be used to identify potential vulnerabilities in a software system. These include penetration testing, vulnerability scanning, and code review. Penetration testing involves simulating an attack on the software system to identify potential vulnerabilities that could be exploited by attackers. Vulnerability scanning involves using automated tools to scan the software system for known vulnerabilities. Code review involves manually reviewing the software code to identify potential vulnerabilities.

By conducting thorough security testing, software developers can identify and mitigate potential security risks before deploying the software system. This is particularly important in predicting and preventing vulnerabilities that could be exploited by attackers. The aim of security testing is to ensure that software systems are as secure as possible, and that they are designed and tested with the latest security measures in mind. Overall, security testing plays a critical role in ensuring the security and integrity of software systems in the face of ever-evolving cybersecurity threats.

Common Vulnerability Exposures (CVE) [09] defines a security vulnerability as "*a flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components.*". The inadvertence of a developer or insufficient knowledge of defensive programming usually causes these mistakes. Still, vulnerabilities are of critical importance for software vendors, who often offer bounties to find them and prioritize their resolution over other less harmful bugs, hence reducing a potential business impact.

Vulnerabilities in source code are a significant threat to software security, and numerous studies have highlighted their importance. For instance, a study by the software security company, Veracode [Ver20], found that 70% of applications contain at least one security flaw in their source code, which can be exploited by attackers. A study by the Ponemon Institute [Ins20] found that 62% of organizations surveyed had experienced a data breach caused by a vulnerability in their source code.

The consequences of vulnerabilities in source code can be severe, ranging from financial losses to reputational damage. A study by the National Institute of Standards and Technology (NIST) [oST19] found that the average cost of a data breach was 3.86 million USD, with a significant portion of that cost being attributed to the impact on customer trust and reputation.

In addition to financial losses, source-code vulnerabilities can also pose risks to human life. For example, vulnerabilities in medical devices or critical infrastructure systems can be exploited by attackers to cause harm to patients or disrupt essential services. A study by the Institute of Electrical and Electronics Engineers (IEEE) [IEE17] highlights the importance of addressing vulnerabilities in medical devices, as these devices can be exploited to cause harm to patients. Similarly, vulnerabilities in critical infrastructure systems, such as those that control power grids or transportation systems, can have serious consequences if they are exploited.

Vulnerabilities are usually reported in publicly available databases to promote their disclosure and fix. One such example is National Vulnerability Database, aka NVD [02]. NVD is the U.S. government repository of standards based vulnerability management data. All vulnerabilities in the NVD have been assigned a CVE (Common Vulnerabilities and Exposures) identifier. The Common Vulnerabilities and Exposures (CVE) Program's primary purpose is to uniquely identify vulnerabilities and to associate specific versions of codebases (e.g., software and shared libraries) to those vulnerabilities. The use of CVEs ensures that two or more parties can confidently refer to a CVE identifier (ID) when discussing or sharing information about a unique vulnerability. For every vulnerability, along with the Git commit IDs of the code related to vulnerability-fix commit, NVD also provides related information, i.e., CVE number, vulnerability description, CWE number (if applicable), time of creation, and the list of the impacted releases in the form of reports.

2.3.4 Vulnerability-mimicking mutants

In the existing literature, there is no clear definition of *Vulnerability-mimicking Mutants*, (i.e., mutants that mimic the vulnerability behavior) to focus on, in order to perform mutation testing to guarantee the software under analysis is vulnerability-free. Therefore, for the purpose of this dissertation, we use the following definition:
A mutant is vulnerability-mimicking if it fails exactly the same tests that are failed by the vulnerability it mimics, hence having the same observable behavior as the vulnerability.

Since a mutant is a slight syntactic modification to the original program, a large number of mutants are generated during mutation testing which requires analysis and execution with the related test suites. This introduces the problem of identifying vulnerability-mimicking mutants among a huge pile of mutants. In our

dataset, vulnerability-mimicking mutants are *3.9%* of the entire lot. To deal with the problem of identification of vulnerability-mimicking mutants (which are not known in advance), in Chapter 6, we introduce a deep learning based approach that predicts *Vulnerability-mimicking Mutants* without requiring any dynamic analysis.

2.4 Limitation of Mutation in Security Testing: Vulnerability Prediction

Vulnerability prediction is the process of identifying software components, such as files or modules, that are likely to contain vulnerabilities. This is typically done using various software analysis techniques, such as static analysis or dynamic analysis. In the case of predicting vulnerable files, the goal is to identify files within a software project that are likely to contain vulnerabilities, such as buffer overflow or SQL injection vulnerabilities.

While mutation testing is a powerful software testing technique, it is challenging for security testing, especially vulnerability prediction, because artificially generated mutants cannot mimic all the potential vulnerabilities that may exist in a software system. Hence, it can be perceived that these mutants are not a complete representation of all the vulnerabilities and there exists a need for actual vulnerability prediction approaches. These techniques can help identify potential security issues that are not easily detectable through manual inspection or traditional testing methods.

2.4.1 Prediction Modeling

Vulnerability prediction modeling aims at learning statistical properties of interest based on historical data. While the resulting models are usually suitable only for the project/application on which they have been trained, the learning process is generic and applies to a specific set of features that associate with the property to predict. In the context of vulnerabilities, a prediction model can be used to classify software components, such as files, as likely or unlikely vulnerable. This information can be used to support the code review process. The task is similar to defect prediction, yet due to the sparsity of available examples, it is harder to predict vulnerabilities than defects [SW13; TW20].

2.4.2 Intra vs Inter Predictions

Prediction modeling is usually performed in both intra- and cross-project fashion, i.e., training on data of the same or of other projects. However, vulnerabilities are project-specific, i.e., they are tied to the project context, used libraries, and development process, and thus, inter-project predictions do not work. Scandariato *et al.* [SWH+14] found that the models for 11 apps out of 20 were too specific for cross-project prediction and the link was more pairwise rather than generic. The

results of cross-project vulnerability prediction in the study of Sara *et al.* [MS16] show high recall but comparatively low F2 score. Therefore, we focus our research in this area on intra-project predictions.

2.4.3 Prediction Granularity Level

Prediction models can target various levels of granularity, such as line, function, component/file, module, etc. However, the key target should be actionable for the developers and code reviewers that are envisioned to use the technique. Given this, a commonly accepted tradeoff is the component (file) level granularity as it has been vetted by Microsoft developers in a study by Morrison *et al.* [MHM+15], and is used by most existing approaches. Thus, in this dissertation, we consider a source code file as our component for vulnerability prediction, *i.e.,*file-level granularity, as it is actionable for industrial use [MHM+15], and provides a baseline for comparing our results with those reported in the relevant literature that we elaborate in Section 7.3.5 in Chapter 7.

2.4.4 Clean Training Data Settings

Jimenez et al. [JRP+19] demonstrated that the existing vulnerability prediction approaches have been built under a *"clean"* training data assumption, *i.e.,*all the component's labeling information (vulnerable / non-vulnerable) is always available irrespective of time, which is unrealistic. Jimenez *et al.* showed that under these settings, aka *Clean Training Data Settings*, prediction approaches fail to account for the gradual revelation of vulnerabilities over time. This results in biased prediction models, i.e., models trained on vulnerabilities that have not been discovered at the release time, *e.g.* all vulnerabilities known from time t onwards are available at all times, even before time t.

2.4.5 Realistic Training Data Settings

In contrast to *Clean Training Data Settings*, where the component's labeling information (vulnerable / non-vulnerable) is always available irrespective of time, *Realistic Training Data Settings* necessitate vulnerability labels to be used for training the prediction models to be those that are available at training time. For instance in *Realistic Training Data Settings*, at a given time t, only the vulnerabilities known at time t should be available for training. All vulnerabilities known after time t should *not* be available for training beforehand. Jimenez *et al.* study demonstrated that *Realistic Training Data Settings* introduce noise in the training data, because every component with no reported vulnerability till the training time is considered as non-vulnerable during training, that makes existing approaches perform poorly.

Irrespective of the poor performance of existing approaches, *Realistic Training Data Settings* represents a realist case study, the vulnerabilities are discovered and

fixed long after the release date of the projects. In our release-based experiments, (*i.e.,*one release for training the model and next release for testing the trained model), only those components are considered as vulnerable in the training set whose vulnerabilities have been discovered and fixed before the next release date of the system.

2.4.6 Seen Vulnerable Components

Vulnerabilities can remain in the code and get propagated throughout different releases (one release after another) of a system, without getting fixed. Due to this, in a release-based experiment, (*i.e.,*one release for training the model and next release for testing the trained model), vulnerable components that are present in the training set and *"seen"* by the prediction model during training can also appear in the testing set. Throughout this dissertation, we refer to such components as *Seen* vulnerable components.

2.4.7 Unseen Vulnerable Components

From one release of a system to the next one, many files/components are modified either to introduce new functionality or to modify an existing one. In the case of Linux Kernel, Wireshark, and OpenSSL projects, we analyzed that 29.95%, 72.53%, and 73.58% of the files, on average, are changed between the releases. A component that was non-vulnerable in the previous release can become vulnerable in this release, because of such a modification by a developer. Due to this, in a release-based experiment, any component in a testing set which is vulnerable and is not available in the training set, represents a novel vulnerability. Since this component is *"unseen"* and has not been trained on by the model, we refer to it as *Unseen* vulnerable component.

2.4.8 Overcoming Key Obstacles in Vulnerability Prediction

There are many vulnerability prediction approaches in the literature but their performance is limited due to a few factors that we explained in the sections above. Firstly, vulnerabilities are fewer in comparison to software bugs, limiting the information one can learn from, which affects the performance of existing techniques. Secondly, the existing approaches learn on both, vulnerable, and supposedly non-vulnerable components. This introduces an unavoidable noise in training data, i.e., components with no reported vulnerability are considered non-vulnerable during training, and hence, results in existing approaches performing poorly. Hence, in Chapter 7, we explore if we can avoid learning on supposedly non-vulnerable components and we propose a deep learning based vulnerability prediction approach that learns only on components known to be vulnerable,

thereby making no assumptions and bypassing the key problem faced by previous techniques. The comparison of our proposed approach with existing techniques on security-critical open-source systems with historical vulnerabilities reported in the National Vulnerability Database (NVD) demonstrates that its prediction capability significantly outperforms the existing techniques.

3

Related Work

This chapter discusses the existing work related to the contribution of the dissertation. We present an overview of the existing literature on mutation testing and works targeted to reduce its cost. This chapter also discusses the existing works on the applications of mutation testing especially specification inference and security testing.

Contents

3.1 Mutation Testing

3.1.1 Random Mutant Sampling and Selective Mutation

Mutation testing has been established as one of the strongest test criteria [CPT+17; ADO14]. Despite its potential, mutation is considered to be expensive since it introduces too many mutants. To this end, random mutant sampling [DLS78; PM10a] and selective mutation [OLR+96] (restricting mutant instances according to their types) have been proposed as potential solutions. Unfortunately, these approaches fail to capture relevant program semantics and performing similarly to random mutant sampling [ZGM+13; KAO+16; CPB+20].

3.1.2 Selective Program Locations for Mutation

Other attempts regard the selection of relevant program locations, which should be mutated. Sun et al. [SXL+17] proposed selecting mutants that reside in diverse static control flow graph paths. Gong et al. [GZY+17] identified dominator nodes (using static control flow graph) to select mutants.

3.1.3 Mutant Selection

More recent attempts regard the identification of interesting mutants (pairs of mutant types and related locations). Petrovic and Ivankovic [PI18b] proposed using the code AST in order to identify "useful" mutants. Papadakis et al. [PDT14] used dynamic features to select mutants that have impact on the internal program states but not on the program behaviour to guide the selection of likely killable mutants. Ma et al. [MCP+21] used a combination of Contextual, modifcation and mutant utility features to predict relevant mutants to code commits. A followup study of Ma et al. [MZS+22] used Graph Neural Networks to predict killable mutants. Mirshokraie et al. [MMP15] employed complexity metrics together with test executions to select killable mutants. Similarly, Titcheu et al. [CPB+20] employed static features, including data flow analysis, complexity, and AST information, in order to perform mutant selection, with respect to the mutants linked with real faults.

In Chapter 4, we approximate the performance of the above approaches through the two baselines we adopt and show that our proposed approach significantly outperforms these. Random mutant sampling performs comparably to operator mutant selection [ZGM+13], while the supervised baseline we consider simulates the AST-based and complexity-based approaches.

3.1.4 Weak Mutant Selection

Perhaps the closest work to ours is from Marcozzi et al. [MBK+18], which attempts to identify subsumed mutants using verification techniques (such as

weakest precondition). While Marcozzi et al.'s approach is particularly powerful, it targets weak mutation opposite to what we do. This results in several false positives in the strong mutation case due to failed error propagation [CPT+17]. Moreover, Marcozzi et al.'s approach is time consuming, and requires complex computations and infrastructure while our proposed approach in Chapter 4 is fast and simple. Nevertheless, future research should attempt to combine these methods.

3.1.5 Deep Learning guided mutant generation

Tufano et al. [TWB+19a] proposed using Neural Machine Translation to learn mutations from bug fixes with the aim of introducing mutations that are syntactically similar to real bugs. Our proposed approach in Chapter 4 relies on the same technology, though it targets a different problem; the identification of high utility mutants, among those given by regular mutation testing tools, while Tufano et al. aim at generating mutants regardless of their potential. This indicates that our proposed approach can complement Tufano et al by selecting relevant mutants. Nevertheless, we focus on subsuming mutants, that help in measuring test adequacy and designing test suites, which are unlikely to be supported by Tufano et al. as there is no notion of subsumption in the bug-fixing sets they use. Moreover, we make no assumption about the availability and repetitiveness of historical bugs and their fixes.

3.1.6 Predictive mutation testing (PMT)

Predictive mutation testing (PMT) [ZZH+19] attempts to predict whether a given test can kill a given mutant without performing any mutant execution. The approach relies on a set of both static and dynamic features (relying on coverage and code attributes) and achieves relatively good results (on average with 10% error). PMT mainly targets intra-project predictions, while we target inter-project. Nevertheless, PMT is incomparable to our proposed approach since PMT aims at evaluating test execution results, while we perform mutant selection prior to any test execution. In other words, we aim at identifying the mutants to be used for test design/generation, while PMT is used to verify whether mutants are killed by some tests. Therefore, the two methods target different but complementary problems.

3.1.7 Evolutionary Mutation Testing (EMT)

Evolutionary Mutation Testing (EMT) [DEG+11] utilizes dynamic features (execution traces) in order to identify interesting locations and mutant types. As such, EMT requires tests and user feedback, which make it different but complementary to ours; our proposed approach in Chapter 4 can set a starting point for EMT or integrate its predictions within EMT's fitness function.

3.1.8 Higher-order Mutation

Higher-order mutation [JH09] aims at dynamically optimizing mutants based on given test suites. While higher-order mutation is only applicable after test generation, our proposed approach can be directly applied to support test generation prior to any test generation. More importantly, while higher-order mutation introduces major mutant execution overheads, our proposed approach does not introduce any expensive dynamic mutant execution.

3.2 Specification Inference

3.2.1 Specification Inference Technique

Modern assertion inference techniques take on *Daikon* [EPG+07], a well-known dynamic technique that infers assertions by monitoring test executions. Given a program under analysis and a test suite, Daikon executes the tests, monitors the program states at various points, and then evaluates candidate assertions, obtained by instantiating assertion patterns on the program states. Those assertions that are *never falsified* by any test at a given program point are reported as likely invariants at the program point. As Daikon does not use mutation analysis or any other sophisticated mechanism to detect irrelevant/redundant assertions, it often reports many assertions that can be weak or redundant with respect to other reported program assertions [MdA22].

3.2.2 Evolutionary algorithm guided Specification Inference

GAssert [TJT+20] and *EvoSpex* [MPA+21] are assertion inference techniques based on evolutionary search algorithms. Similar to Daikon, these tools execute a test suite of the program under analysis and observe the execution in order to infer assertions that are consistent with the observations. By favoring shorter assertions during evolution, and also favoring assertions that are able to detect buggy behaviors via mutation analysis, these techniques are able infer shorter and stronger assertions, compared to Daikon. However, as the components of the evolutionary process are specifically designed to handle the assertion languages these tools support, changing or extending these languages implies redefining evolutionary operators and other elements of the process, which is non-trivial.

3.2.3 Fuzzing based Specification Inference

SpecFuzzer [MdA22] is another assertion inference technique which infers assertions through a combination of static analysis, grammar-based fuzzing, and mutation analysis. First, it uses a lightweight static analysis to produce a grammar for the assertion language, which is tuned to the program under analysis. Second, it uses a grammar-based fuzzer to generate candidate assertions from the grammar.

Then, a dynamic detector determines which of those assertions are consistent with the behavior exhibited by a provided test suite. Finally, SpecFuzzer eliminates redundant and irrelevant assertions using a selection mechanism based on mutation analysis. A salient feature of SpecFuzzer is that developers can adjust the produced specifications by tuning the grammar, as opposed to making changes to a search algorithm, as GAssert and EvoSpex would require.

It is worth remarking that all of the above described techniques infer assertions from the current program behavior, which may not necessarily be the intended program behavior if the program is incorrect. Inferred assertions are useful for many tasks, including regression and differential analyses, as well as for program understanding.

3.3 Security Testing

3.3.1 Proof of vulnerability

There exist several vulnerability datasets for many programming languages [BNM21; FLW+20; GDJ+22]. However, they do not contain bug-witnessing test cases to reproduce vulnerabilities, i.e., Proof of Vulnerability (Pov). Such test cases are essential for this study in order to determine whether generated mutants are *Vulnerability-mimicking Mutants*, as explained in the section above. In general, it is hard to reproduce exploitation (i.e., PoV) for vulnerabilities. *Vul4J* [BSF22] is a dataset of real vulnerabilities, with the corresponding fixes and the PoV test cases, that we utilized for our study on *Vulnerability-mimicking Mutants* in Chapter 6. Although, due to a few test cases failing even after applying the provided vulnerability-fixes, we had to exclude a few vulnerabilities. In total, we conducted our study on 45 vulnerabilities. In table 6.1 of Chapter 6, we mention the details of considered vulnerabilities that include CVE ID, CWE ID and description, Severity level (that ranges from 0 to 10, provided by National Vulnerability Database [02]), number of Files and Methods that were modified during the vulnerability fix, and number of Tests that are failed by the vulnerability a.k.a. Proof of Vulnerability (PoV).

3.3.2 Pre-trained Language Model guided Mutation

µBERT [DP22] is a mutation testing tool that uses a pre-trained language model *CodeBERT* to generate mutants by masking and replacing tokens. *µBERT* takes a Java class and extracts the expressions to mutate. It then masks the token of interest, e.g. a variable name, and invokes CodeBERT to complete the masked sequence (i.e., to predict the missing token). This approach has been proven efficient in increasing the fault detection of test suites [DP22] and improving the accuracy of learning-based bug-detectors [RW22] and thus, we consider it as a representative of pre-trained language-model-based techniques.

For instance, consider the sequence `int total = out.length;`, *µBERT* mutates the object field access expression `length` by feeding CodeBERT with the masked sequence `int total = out.<mask>;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `total`, `length`, `size`, `count` and `value` for the given masked sequence. *µBERT* takes these predictions and generates mutants by replacing the masked token with the predicted ones (per masked token creates five mutants). *µBERT* discards non-compilable mutants and those syntactically the same as the original program (cases in which CodeBERT predicts the original masked token).

3.3.3 Designing vulnerability-targeted mutation operators

The unlikelihood of standard PIT [CLH+16] operators to produce security-aware mutants was observed by Loise et al. [LDP+17] where the authors designed pattern based operators to target specific vulnerabilities. They relied on static analysis for validation of generated mutants to have similarities with their targeted vulnerabilities.

Fault modeling related to security policies was explored by Mouehli et al. [MLB07] where they designed new mutation operators corresponding to fault models for access control security policies. Their designed operators targeted modifying user roles and deleting policy rules to modify application context, specifically targeting the implementation of access control policies.

Mutating high-level security protocol language (HLPSL) models to generate abstract test cases was explored by Dadeau et al. [DHK+15] where their proposed mutations targeted to introduce leaks in the security protocols. They relied on the automated validation of Internet security protocols and applications tool set to declare the mutated protocol unsafe and capable of exploiting the security flaws.

Targeting black box testing by mutating web applications' abstract models was explored by Buchler et al. [BOP12] where they produced model mutants by removing authorization checks and introducing noisy (non-sanitized) data. They relied on model-checkers to generate execution traces of their mutated models for the creation of intermediate test cases. Their work was focused on guiding penetration testers to find attacks exploiting implementation-based vulnerabilities (e.g., a missing check in a RBAC system, non-sanitized data leading to XSS attacks).

Similar to Loise et al., Nanavati et al. [NWH+15] also show that traditional mutation operators only simulate some simple syntactic errors. Hence, they designed memory mutation operators to target memory faults and control flow deviation. They focused on programs in C language and rely on memory allocation primitives in specific to C. Similarly, Shahriar and Zulkernine [SZ08] and Ghosh et al. [GOM98] also defined mutation operators related to the memory faults. Their designed operators also exploited memory manipulation in C programs (such as buffer

overflows, uninitialized memory allocations, etc.), which security attacks may exploit. These works also focused on programs in C language.

Unlike the above-mentioned related works, our work in Chapter 6 does not target a specific vulnerability pattern/type. Also, since we rely on a pre-trained language model (employed by *μBERT*), we do not require to design specific mutation operators to target specific security issues. Additionally, our validation of *Vulnerability-mimicking Mutants* is not based on a static analysis, but rather a dynamic proof as our produced/predicted vulnerability-mimicking mutants fail tests that were failed by respective vulnerabilities, a.k.a., Proof-of-vulnerability (PoV).

3.3.4 Manual Feature Definition for Vulnerability Prediction

Early work in the area of vulnerability prediction has focused on defining features that could be linked to vulnerabilities and thus to be used to train learners. The first such work can be traced back to the study of Neuhaus *et al.* [NZH+07], which investigated the use of libraries and function calls. Later, Shin *et al.* [SW13; SMW+11] and Zulkernine *et al.* [CZ11] investigated the use of code metrics such as complexity, code churn, and object oriented metrics. Theisen and Williams [TW20] showed that a combination of these features can slightly improve the F-score and recommend identifying new features.

These approaches, although promising, were all using features designed based on human intuition. Scandariato *et al.* [SWH+14] advocated that the learners should find their features without human intervention. To achieve this, they suggested the *Text Mining* approach where code is treated as text and the learner learns from *Bag of Words* (BoW). The results of their exploratory study demonstrated that *Text Mining*'s prediction power was superior to the state of the art vulnerability prediction models with good performance for both precision and recall in intra-project predictions.

3.3.5 Deep Learning guided Prediction

Recently, deep learning techniques have been explored to automatically learn the required features to predict vulnerabilities. Li et. al [LZX+18] used Bidirectional LSTMs to train a vulnerability prediction model on *code gadgets*, which are semantically related lines of code. Under *Clean Training Data Settings*, this technique was shown to be effective for analyzing two particular weaknesses, namely, buffer error vulnerabilities (CWE-119) and management error vulnerabilities (CWE-399). In contrast, our proposed approach in Chapter 7 trains the translation model on sequences extracted from the source code and does not target specific weaknesses.

Machine learning has also been used in other software engineering prediction

tasks. For instance, several works [DLR12; HBB+12; YLX+15; WLT16] used machine learning models for defect prediction. Particularly, RNN models have been used for automatically fixing errors in C programs [GPK+17], for generating API usage sequences [GZZ+16], and for fault localization [HLZ16]. Closer to our work, machine translation-based approaches have been successfully applied to automatically learn code features for detecting code clones [WTV+16], and interesting mutants [GOD+22], for learning how to mutate source code from bugs [TWB+19a], and to produce bug-fixing repairs [TWB+19b]. To our knowledge, our proposed approach in Chapter 7 is the first that proposes and evaluates a machine translation-based vulnerability prediction.

4

Cerebro: Static Subsuming Mutant Selection

Mutation testing research has indicated that a major part of its application cost is due to the large number of low utility mutants that it introduces. Although previous research has identified this issue, no previous study has proposed any effective solution to the problem. Thus, it remains unclear how to mutate and test a given piece of code in a best effort way, i.e., achieving a good trade-off between invested effort and test effectiveness. To achieve this, we propose Cerebro*, a machine learning approach that* statically *selects subsuming mutants, i.e., the set of mutants that resides on the top of the subsumption hierarchy, based on the mutants' surrounding code context. We evaluate* Cerebro *using 48 and 10 programs written in C and Java, respectively, and demonstrate that it preserves the mutation testing benefits while limiting application cost, i.e., reduces all cost application factors such as equivalent mutants, mutant executions, and the mutants requiring analysis. We demonstrate that* Cerebro *has strong inter-project prediction ability, which is significantly higher than two baseline methods, i.e., supervised learning on features proposed by state-of-the-art, and random mutant selection. More importantly, our results show that* Cerebro*'s selected mutants lead to strong tests that are respectively capable of killing 2 times higher than the number of subsuming mutants killed by the baselines when selecting the same number of mutants. At the same time,* Cerebro *reduces the cost-related factors, as it selects, on average, 68% fewer equivalent mutants, while requiring 90% fewer test executions than the baselines.*

Contents

4.1 Introduction

Research and practice with mutation testing has shown that it can effectively guide developers in improving their test suite strengths [CPT+17; ADO14], and can be used to reliably compare test techniques [ABL+06; PSY+18]. A key issue though, is that it is expensive, as a large number of mutants are involved, the majority of which are of low utility, i.e., they do not contribute to the testing process [JH09; KPM10; ADO14]. This means that mutation testers should filter their mutant sets using manual analysis to identify equivalent mutants [BA82], and perform numerous test executions to discard mutants that do not provide testing value, i.e., mutants that are detected by the tests designed to detect other mutants [JH09; KPM10; ADO14].

Working with large real-world systems makes the problem almost intractable due to the vast numbers of mutants involved. Test execution overheads alone can limit the scalability of the technique. For instance, in our experiments, we needed around 48 hours to execute the mutants for a single component of the systems we examined. At the same time the manual effort required by testers is escalated with larger programs as the number of mutants grows proportionally to program size.

To reduce application cost, it is imperative to limit the number of mutants to those that are actually useful, prior to any manual mutant analysis or test execution. Thus, we need to identify which mutants are killable in order to limit the manual effort involved in their identification, and also to identify the mutants that are subsuming (disjoint)[1], in order to reduce unnecessary computations, and to provide accurate adequacy measurements [PHH+16].

This problem is known as the mutant selection problem [PKZ+19] and has been studied in the form of selective mutation [OLR+96; ZGM+13], i.e., restricting the number of transformations to be used, with limited success [KAO+16; CPB+20]. Though, the key issue with mutant selection is the simple syntactic-based nature of the selection process. The issue is that mutants are introduced everywhere with respect to simple language operators, e.g., by replacing an operator with another, that completely ignore the program and particular location semantics. This operator matching mutant selection has the unfortunate effect of introducing mutants independent of their context and program semantics.

We propose *Cerebro*[2], a machine learning technique that learns to identify interesting mutants given their context. In particular we learn the associations between mutants and their surrounding code. Our learning scope is a relatively small area around the mutation point that differentiates locally, the mutants that

[1]The term disjoint mutants refers to a minimal subset of mutants that need to be killed in order to reciprocally kill the original set [KPM10; PCT18].

[2]Cerebro is a fictional device appearing in Marvel comics used by the X-Men to detect human mutants. More details in `https://en.wikipedia.org/wiki/Cerebro`.

are useful from those that are not. This allows mutating the program elements to fit best to their context, instead of mutating entire codebases with every possible transformation, enabling inter-project predictions.

Cerebro operates at lexical level, with a simple code preprocessing. In particular, a mutant and its surrounding code is represented as a vector of tokens where all literals and identifiers, i.e., user defined variables, types, and method calls, are replaced with predefined, hence predictable, identifier names. This allows restricting the related vocabulary and learning scope to a relatively small fixed size of tokens around the mutation points. Learning is performed using a powerful and language-agnostic machine translation technique [BGL+17] that we train on related code fragments and their labels.

We consider useful, the subset of mutants that resides on top of the subsumption hierarchy and subsumes the others [KAD+14], *aka subsuming mutants* [JH09], for the set of all possible mutant instances produced by a given set of mutation operators. Mutant M_1 subsumes mutant M_2 if every test case detecting M_1 also detects M_2. This implies that the tests detecting the subsuming mutant will also detect the subsumed ones thereby making subsumed mutants redundant.

We implemented *Cerebro* and evaluated its ability to predict (inter-project predictions) subsuming mutants on a large set of programs, composed of 48 C programs (CoreUtils) and 10 Java projects (Apache Commons, Joda-Time, and Jsoup). Our results demonstrate that *Cerebro* significantly outperforms both, random mutant selection and a supervised machine learning approach (used by previous research) on both, C and Java benchmarks.

In particular, our results show that *Cerebro* significantly outperforms the baselines. In Java projects, *Cerebro* obtained 2.81 times higher MCC[3] values, an improvement of 82% in F-measure, 68.88% in Precision, and 85.71% in Recall over the state-of-the-art supervised machine learning. In C programs, *Cerebro* obtained 2.76 times higher MCC values, 3.72 times higher precision, and slightly increased Recall value (4% higher). The improvement measured in F-measure is approximately 65%.

To put the predictions into a context and understand its influence on mutation testing, we also validated *Cerebro* in a controlled simulation of the envisioned use case. In particular, we simulate a scenario where testers are guided by mutation testing, i.e., they design test cases based on mutants. Therefore, fewer mutants imply less effort, while stronger mutants imply stronger tests. Our analysis shows that *Cerebro* achieved more than twice the subsuming mutation scores[4] in both, C

[3]The *Matthews Correlation Coefficient* (MCC) [Mat75] is a reliable metric of the quality of prediction models [SBH14a], relevant when the classes are of very different sizes, e.g. in case of C programs, 10.2% subsuming mutants (positives) over 89.8% non-subsuming mutants (negatives).

[4]*Subsuming mutation score* (MS*) is the ratio of the killed and the total number of subsuming mutants.

and Java programs that we use. At the same time *Cerebro* required significantly less effort in terms of both, analyzed equivalent mutants and test executions. In C programs, 3.70% of the mutants analyzed by *Cerebro* are equivalent, while 55.56% and 53.33% analyzed by random mutant selection and supervised learning, respectively are equivalent; *Cerebro* also required 91% fewer test executions than random selection and supervised learning, respectively. In Java programs, *Cerebro* required the analysis of 41% and 36% fewer equivalent mutants, and 92% and 87% fewer test executions than random mutant selection and supervised learning, respectively.

All-in-all this chapter makes the following contributions:

1. We present *Cerebro*, a powerful static subsuming mutant selection technique.
2. We provide evidence suggesting that *Cerebro* successfully predicts subsuming mutants with 0.85 Precision, 0.33 Recall and 0.46 MCC.
3. We show that *Cerebro* significantly outperforms the current state-of-the-art, i.e., random mutant selection and previously proposed machine learning technique, by revealing 2 times the subsuming mutants, while analyzing 64% to 67% fewer equivalent mutants and requiring 89% to 92% fewer test executions.

The remainder of the chapter is organized as follows.Section 4.2 elaborates on a particular motivating example for *Cerebro*. Section 4.3 describes the approach in detail. Section 4.4 introduces the research questions and Section 4.5 details the experimental setup. The results of our experimental evaluation are summarized in Section 4.6. We discuss threats to validity in Section 4.8. In Section 4.7 we also discuss the impact of the abstraction process and mutants' context size on *Cerebro*'s prediction performance. Finally, we present our conclusion in section 4.10.

4.2 Motivating Example

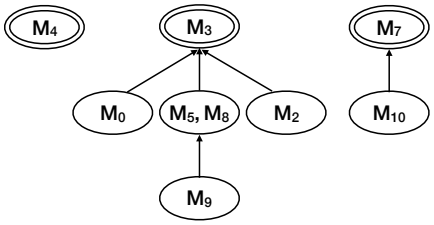
Let us consider the code snippet of function `max` of Figure 4.1a, which takes three integers as input and returns the maximum number among them. Also, consider (for simplicity) that we have the 11 mutants shown in the figure. For instance, mutant M_0 mutates sub-expression `a >= b` of line 2 into `a < b`. Similar mutations on relational operations were applied to produce mutants M_1, M_3, M_5, M_6 and M_8. Mutants M_2 and M_7 replace the conjunction (`&&`) by the disjunction (`||`). While mutants M_4, M_9 and M_{10} replace the returned variable name by other variable name or constant (M_{10} replaces variable name `c` by constant `0`).

For the sake of the thorough demonstration, we observed scenarios under the following testing conditions: A test case invoking `max(1,2,0)` and expecting `2` as a result, kills mutant M_3, as well as, mutants M_0, M_2, M_5, M_8, and M_9. But

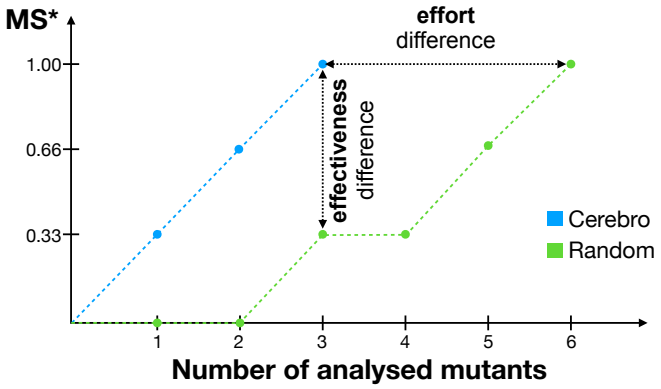
Figure 4.1: The example shows that by analyzing only the three subsuming mutants M_3, M_4 and M_7 is enough for covering all 9 killable mutants. Particularly, mutants M_1 and M_6 are equivalents.

```
1        int max(int a, int b, int c){
2          if (a >= b && a >= c) //M0: (a < b && a >= c)
                                 //M1: (a >= b && a > c)
                                 //M2: (a >= b || a >= c)
                                 //M3: (true && a >= c)
3            return a;           //M4: return b;
4          else if (b >= a && b >= c)//M5: (b < a && b >= c)
                                 //M6: (b >= a && b > c)
                                 //M7: (b >= a || b >= c)
                                 //M8: (false && b >= c)
5            return b; //M9: return a;
6          else
7            return c; //M10: return 0;
8        }
```

(a) The code and mutants for the function `max`.



(b) Mutant subsumption hierarchy graph for the function `max`. Subsuming mutants are the ones at the top of hierarchy.



(c) The mutants selected by *Cerebro* lead to stronger test suites than those designed to kill randomly selected mutants, when equal number of mutants is analyzed.

tests invoking `max(2,0,1)`, `max(1,0,2)`, and `max(0,2,1)` will kill mutants M_0, M_2, M_5, M_8, and M_9, except M_3. Figure 4.1b shows a graph representation of the subsumption relation between the 9 killable mutants. Moreover, Figure 4.1b shows that M_3 subsumes M_0, M_5, M_8 and M_2. Particularly notice that mutants M_5 and M_8 are indistinguishable, since they are killed by the same tests, and subsume mutant M_9. Although, mutants M_1 and M_6 are equivalent.

In summary, mutants M_3, M_4 and M_7 are subsuming, indicating that in order to kill every killable mutant it is sufficient to kill only these 3 subsuming mutants.

Cerebro will take as input the program `max` and the set of mutants, and it will point to those that are most likely subsuming. In an ideal scenario, *Cerebro* would point only to M_3, M_4 and M_7, but it is possible, as in every machine learning based technique, that it does some mistakes, i.e., incorrect predictions of subsuming mutants, pointing to some non-subsuming (subsumed or equivalent mutants) as subsuming.

For instance, consider the case in which *Cerebro* predicts M_3 and M_4 and M_{10} as subsuming mutants. Therefore, a tester will incrementally design test cases to kill all the predicted mutants. Assume that the tester starts by analyzing mutant M_3 and designs a test to kill it, e.g., by invoking `max(1,2,0)`. This test does not kill the rest of the selected mutants. The tester then proceeds to analyze the surviving mutant M_4, for which he/she designs a test that invokes `max(2,0,1)` to kill it. Finally, the tester designs a test by invoking `max(0,1,2)`, which kills mutant M_{10} and also (non selected) subsuming mutant M_7. Notice that this test suite designed to kill all mutants selected by *Cerebro* progressively increments the MS*: first test kills subsuming mutant M_3 leading to a MS* of 33.33%; second test kills subsuming mutant M_4, obtaining 66.66% of MS*; and finally, third test kills collaterally subsuming mutant M_7 leading to a MS* of 100%.

Consider a scenario in which mutants are selected *randomly*. For instance, assume that M_9 is the first one to be selected for analysis for which a test case invoking `max(0,2,1)` is designed to kill it. This test collaterally kills mutants M_5 and M_8, but it does not kill any subsuming mutant. Then, assume that equivalent mutant M_1 is randomly selected, adding no value to the testing process, but requiring analysis anyway. Afterwards mutant M_0 is randomly selected for which a test case invoking `max(2,0,1)` is designed to kill it, that fortunately also kills subsuming mutant M_4. Then, mutant M_2 is randomly selected for which the tester designs a test to kill it by invoking `max(1,0,2)`. This test also kills mutant M_{10}, but no subsuming mutant is killed. After that, tester randomly selects mutant M_3 for analysis and designs a test by invoking `max(1,2,0)` to kill it. This test kills subsuming mutant M_3 and also mutant M_2. Finally, mutant M_4 is randomly selected for which the tester designs a test to kill it, by invoking `max(2,0,2)`. Hence, all subsuming mutants are killed.

In this particular scenario we can observe that MS* remains at 0% after analyzing the first 2 mutants randomly selected, and reaches a MS* of 33.33% after analyzing the third randomly selected mutant. The analysis of the fourth selected mutant (non-subsuming) did not add value (MS* remains the same). Finally, fifth and sixth analyzed mutants were subsuming, leading to a test suite that obtains MS* of 100% after analyzing 6 mutants.

Figure 4.1c depicts the progress of MS* obtained by the test suites when guided by *Cerebro* and random mutant selection in the previously described scenarios. Through this example we demonstrate a case where two approaches analyze the same number of mutants (same effort) with *Cerebro* having higher effectiveness (MS*) than the random mutant selection baseline. At the same time, in order to reach the same MS* as *Cerebro*, random mutant selection needs more effort, i.e., it will require the analysis of many more mutants than *Cerebro* (in the example random baseline analyzed two times more mutants than *Cerebro*).

There are several points we want to highlight about the particular scenarios just described. First, it is essential to notice that mutants selected by *Cerebro* will be as close as possible to subsuming in the subsumption relation. Killing these (almost subsuming) mutants can help in killing subsuming mutants predicted as non-subsuming by *Cerebro*, for instance, the test that kills subsumed mutant M_{10}, also kills subsuming mutant M_7 that was incorrectly predicted as non-subsuming by *Cerebro*. Second, it is also important to notice that *Cerebro* selects the least possible number of equivalent mutants, saving the time of analysis to the tester (in the example, *Cerebro* did not predict any equivalent mutant as subsuming). Third, notice that the prediction performance obtained by *Cerebro* does not necessarily reflect its effectiveness in practice, since mutant kills are not independent of one another. While *Cerebro* reached 66.66% of Precision and 66.66% of Recall in the example, in practice, the test suite designed to kill all selected mutants obtains 100% of subsuming mutation score (MS*). And fourth, it is worth to study the trade-off between the effectiveness and effort of the different mutant selection techniques. We consider all these points in our empirical evaluation to assess the prediction performance, effectiveness, and effort required by *Cerebro* and the related mutant selection techniques.

4.3 Approach

The main objective of *Cerebro* is to automatically learn the silent features/-patterns of the context surrounding subsuming mutants without requiring any features definition and/or selection by human intervention, that we can use later to predict if mutants on an unseen source code are likely to be subsuming or not. Thus, we train a machine translator (viz. an encoder-decoder model) to identify subsuming mutants, by feeding it with source code where the statement (to mutate)

is annotated with the mutant type and its label (subsuming or not). Machine translators have been successfully used to translate text from one language to another, as they automatically recognize (i) the features of the language (to be translated) and (ii) the required translation (to the desired language). In our case, it is used to automatically identify the features of subsuming mutants without any investment of time and/or resources to define features.

After training, one can input to the translator, an unseen mutant (source code where the statement to mutate is annotated with the mutation annotation). The translator will append the label to the mutant given as input, to predict whether it is subsuming or not.

Figure 4.2 shows an overview of the implementation. For training, *Cerebro* takes a set of mutants and their corresponding label. In each mutant source code, the statement (to mutate) is annotated with the mutation annotation, and the model learns the label to be appended to this annotation, that indicates whether the mutant is subsuming or non-subsuming. We can summarize *Cerebro*'s pre-processing, training and testing steps as follows:

1. *Abstraction:* Producing abstracted code of the actual source code by removing irrelevant information (e.g. comments) and replacing user-defined identifiers and literals (e.g. variable names) by predictable tokens;

2. *Pairs Generation:* Generating the pairs (input-expected output) to be used for training, by adding the corresponding label into the mutation annotations;

3. *Training:* Training the machine translator to learn which label is to be appended to the mutation annotations;

4. *Testing:* Utilizing the trained translator to predict and append labels to the mutation annotations present in unseen mutant source code.

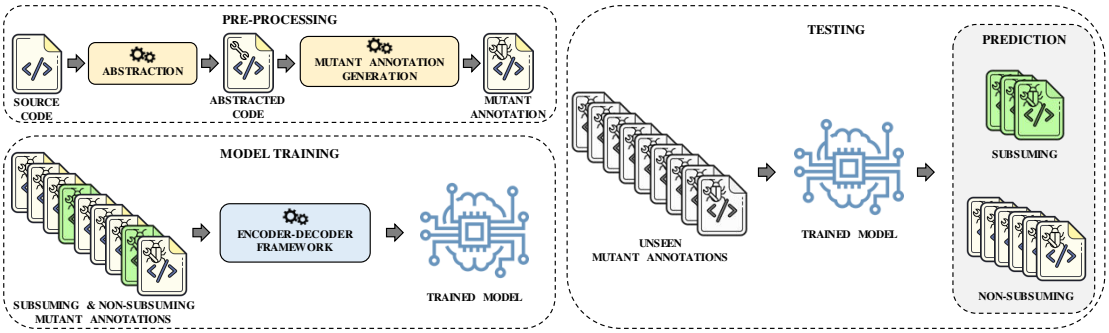
In the remainder of this section we describe each of the aforementioned phases of our approach, in detail.

4.3.1 Abstracting the Irrelevant Information

A major challenge in dealing with raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code. On such a large scale, vocabulary may hinder the goal of learning features surrounding the subsuming mutants. Thus, to reduce vocabulary size, we abstract source code by replacing user-defined entities with re-usable identifiers.

Figure 4.3 shows an actual code snippet (Figure 4.3a) converted into its abstract representation (Figure 4.3b). The purpose of this abstraction is to replace any reference to user-defined entities (function names, types, goto labels, variable

Figure 4.2: Implementation: Source code is abstracted and attached with mutation annotation to produce mutant annotations. Model is trained on mutant annotations to further append the label (subsuming/non-subsuming). Trained model is provided with an unseen mutant annotation to append the label. The appended label acts as the prediction for the unseen mutant annotation.



names and string literals) by identifiers that can be reused across source code file, hence reducing the vocabulary size. Thus, our abstraction approach first detects user-defined entities before replacing them with unique identifiers (new IDs).

New IDs follow the regular expression `(fn|tp|lb|vr|lr)_(num)`$^+$, where `num` stands for numbers $1, 2, 3, \ldots$ assigned in a sequential and positional fashion based on the occurrence of that entity. All the user-defined *Function* names, *Type* names, *Variable* names, *Labels*, and *String Literals* are replaced with `fn_num`, `tp_num`, `lb_num`, `vr_num`, and `lr_num`, respectively. Thus, the first function name found receives the ID `fn_1`, the second receives the ID `fn_2`, and so on. If any of these entities appear multiple times in a source code file, it is replaced with the same ID.

Additionally, we remove code comments and add mutation annotations to encode the mutation operator and the corresponding label (to be learned by the translator). Our mutation annotations have the general shape `MST[MutationOperator]MSP[]`, where MST and MSP denote mutation annotation start and stop, respectively, and MutationOperator indicates the applied mutation operation (in green in Figure 4.3c). Between the last brackets `[]`, our trained model adds one of the labels `S` or `N`, indicating that the mutant obtained by applying the mutation operation, is predicted as subsuming or non-subsuming, respectively.

4.3.2 Pairs Generation

The mutation operation (`ReturnValsMutator`[5]) shown in Figure 4.3c represents a mutant in which the sentence `return null` is replaced by `throw new`

[5] `https://pitest.org/quickstart/mutators/#RETURN_VALS`

Figure 4.3: Abstraction: Actual Source Code (4.3a) is abstracted by replacing user-defined entities (Function names, Type names, Variable names) with tokens (fn_num, tp_num, vr_num) to achieve the Abstracted Code (4.3b). Mutant annotation (4.3c) is generated by adding the Mutation annotation with its corresponding label, *i.e.*,Subsuming (S) or Non-Subsuming (N). The trained model is used for prediction of unseen mutant annotations.

(a) Actual Source Code

```
1  .
2  .
3  public String getOptionValue (
4   final Option option ) {
5   if ( option == null ) {
6    return null ;
7   }
8   final String [] values =
9    getOptionValues ( option ) ;
10  return ( values == null ) ?
11   null : values [ 0 ] ;
12 }
13 .
14 .
```

(b) Abstracted Code

```
1     .
2     .
3     public String fn_3 (
4      final tp_1 vr_3 ) {
5      if ( vr_3 == null ) {
6       return null ;
7      }
8      final String [] vr_5 =
9       fn_4 ( vr_3 ) ;
10     return ( vr_5 == null ) ?
11      null : vr_5 [ 0 ] ;
12    }
13    .
14    .
```

(c) Mutant Annotation

```
1              .
2              .
3              public String fn_3 (
4               final tp_1 vr_3 ) {
5               if ( vr_3 == null ) {
6                return null ; MST[ReturnValsMutator]MSP[  ]
7               }
8               final String [] vr_5 =
9                fn_4 ( vr_3 ) ;
10              return ( vr_5 == null ) ?
11               null : vr_5 [ 0 ] ;
12             }
13             .
14             .
```

Label
S / N

`java.lang.RuntimeException()` exception. Notice that this mutant is labeled as subsuming in our dataset, since there is only one test that can kill it, when the input `option` is null. Hence for training we consider `S` as the label to be learned by the translator to predict this mutant as subsuming.

To do so, we train in pairs (MutantAnnotation, MutantAnnotation+Label), where the first component is the annotated code shown in Figure 4.3c, and the second component is the same code with the predicted label, i.e., `MST[ReturnValsMutator-]MSP[S]` in our case, to indicate that the mutant is subsuming. The resulting text is arranged in a single sentence to represent a sequence of space-separated entities (the representation supported by the machine translator). The only difference between the input sequence given to the translator and the expected output sequence produced by it, is the predicted label `S` or `N`. Using these sequences, we intend to capture as much code as possible around the mutant without incurring the exponential increase in training time.

4.3.3 Building the Machine Translator

To build our machine translator, we train an encoder-decoder model that can transform an input sequence to a desired output sequence. In our representation, a sequence consists of tokens separated by spaces that ends with a newline character. Thus, we train the encoder-decoder by feeding it with pairs of sequences, produced in the previous step. The translator learns to replicate the abstracted source code with the mutation annotation and to append the label (S/N) that will be used as a prediction for the mutant.

We found that training the translator on sequences of maximum *100* tokens in length is computationally feasible, but expensive (740 training hours required on a Tesla V100 GPU). Hence, we also experiment with sequences of *50* tokens in length and demonstrate that the computation cost of training the translator can be further contained (360 training hours required). We name *Cerebro* trained on sequences of 100 tokens in length as Cerebro-*100*. Following our naming convention, we name *Cerebro* trained on sequences of 50 tokens in length as Cerebro-*50*.

4.3.4 Predicting from appended labels

To predict whether or not a certain mutation at a particular position in an unseen code is subsuming, we abstract the unseen code followed by sequence generation which results in abstracted code sequence attached with mutation annotation as depicted in Figure 4.2. We feed this sequence into the trained machine translator to yield an output sequence with an appended label. The appended label acts as a prediction (subsuming/non-subsuming) for this specific mutation. If the translator produces an output sequence with a change other than appending the predicted label, the input sequence is predicted as non-subsuming, by default. In our experiments reported in Section 4.6, this happened in 4.2% and

0.1% of the sequences for C and Java programs, respectively.

4.4 Research Questions

We start by checking the prediction ability of *Cerebro* and ask:

RQ1 *Prediction Performance:* How effective is *Cerebro* in predicting subsuming mutants?

We leverage two datasets, made of C and Java programs, for which extensive mutation analysis has been performed to identify subsuming mutants. We reimplemented 2 techniques that we use as baselines in our analysis. The first baseline is a *Random* mutant sampling, while the second is a supervised machine learning method based on manually designed features that were used by previous work [CPB+20] (e.g., data flow, control flow, etc.). These features are used to train a binary classifier in order to predict whether a mutant is subsuming or not. Further details about the baselines can be found in Section 4.5.2.

After analyzing the predictions, we turn our attention to the envisioned application scenario; measuring test effectiveness of the predicted mutants. It is important to check the application case because a) predictions may select weak mutants [CPB+20] (weak subsuming mutants result in lower test effectiveness than the strong ones), b) selected mutants may not be diverse as they may include mutually subsuming mutants [KAD+14], and c) tester benefits are unclear. Thus, we ask:

RQ2 *Effectiveness Evaluation:* How does *Cerebro* compare with the baselines in terms of subsuming mutation score?

We perform a simulation of a mutation testing scenario where a tester analyzes the selected mutants in order to generate tests [ABL+06; KAO+16; CPB+20]. For test effectiveness, we measure the subsuming mutation score (MS*) achieved by the tests that kill the selected mutants. In essence, we evaluate the guidance offered by the mutants when testers design tests to kill the selected mutants. It is worth noticing that in this part of the experiment we control the number of mutants, i.e., all techniques analyze the same number of mutants. Such simulation is typical in mutation testing literature [ABL+06; KAO+16; CPB+20] and aims at quantifying the benefit of an approach over the other.

Complementary to the previous question, we compare the effort required by each technique to obtain the same level of test effectiveness. Hence, we first investigate the human effort measured in terms of the number of mutants analyzed by the tester, to reach the same subsuming mutation score using *Cerebro* and the baselines. Hence, we ask:

RQ3 *Manual Effort:* How many mutants require manual analysis in order to reach a given level of subsuming mutation score?

We perform a similar simulation of a testing scenario in which we measure how many mutants the tester needs to analyze (generate a test case to kill or judge equivalence), until he/she obtains a determined subsuming mutation score. This allows us to quantify the human effort required by each approach to obtain the same benefit.

Related to the previous question, we also investigate the number of test executions necessary to reach the same subsuming mutation score, by following the incremental process of mutation analysis, *i.e.,*a tester picking a mutant and analyzing it. If the picked mutant is killable, he/she generates a test case that kills it, and then checks if the remaining alive (not analyzed and not killed) mutants are collaterally killed by the same test (by executing the generated test on alive mutants). The killed mutants are removed from the set of alive mutants. Then, we ask:

RQ4 *Computational Effort:* How many test executions are required in order to reach a given level of subsuming mutation score?

We perform a simulation as before, but in this case, every time that a test is generated, we count the number of test executions and measure the attained subsuming mutation score, until we reach a given subsuming mutation score.

4.5 Experimental Setup

4.5.1 Benchmarks and Ground Truth

In order to show that our approach is language agnostic, we make our evaluation on a set of C and Java programs.

C-Benchmark: To perform our study that requires strong test suites, we used an independently built dataset from related work [CPC+21]. It includes C programs from the GNU Coreutils[6], that consist of file, text and shell utility programs widely used in Unix systems. The data-set is composed of 48 GNU Coreutils (v8.22) programs *aka* subjects (mentioned in Table 4.1), each packaged with an accompanying system test suite, generated by developers. The size of these programs ranges from 1,000 to 14,000 lines of code (LOC), with a median size of 3,500 LOC. For each subject, the data-set includes a mutant-test killing matrix that records, for each mutant, a set of tests that kill it.

The mutant-test killing matrices were obtained by generating mutants using the *Mart* mutant generation tool [CPL19] and executing them against large test pools.

[6]https://www.gnu.org/software/coreutils/

The test pools were built by considering developer tests and adding automatically generated tests using a 24 hours run of KLEE [CDE08]. Additionally, mutation-based test suites were automatically generated using 128 different configurations of *SEMu* [CPC+21], each running for 2 hours, and an additional 'seeded' test generation of KLEE. To reduce the total execution cost, for each program, the 3 functions that were covered by the largest number of developer tests were selected for mutation analysis, *i.e.,*mutants were generated only for these functions.

We use these mutant-test killing matrices to compute the mutant subsumption, following the definition given in Section 2.1.3, and label each mutant as either subsuming or non-subsuming. To make the problem as balanced as possible (to assist in machine learning), we mark as subsuming all mutants in the top of the hierarchies, including mutually subsumed mutants.

Needless to say, it is possible to have some noise in our labeling process in the sense that mutants labeled as subsuming may be non-subsuming. The data-set reduced this noise by augmenting the test suites with multiple large and diverse test suites generated by different state-of-the-art tools. Please refer to the threat in Section 4.8 for a related discussion.

Java-Benchmark: For Java we select a set of well-tested open source projects from GitHub. We select projects from the Apache Commons Proper[7] repository of reusable Java components, Joda-Time[8] - a date and time library, and Jsoup[9] - an HTML manipulation library. The set counts 10 projects: `commons-cli`, `commons-codec`, `commons-collections`, `commons-csv`, `commons-io`, `commons-lang`, `commons-net`, `commons-text`, `jsoup`, `joda-time`. These projects contain up to 284 classes. Table 4.1 reports the version/commit of each project we used for our study. Following a similar procedure done for C in [CPC+21], we also build test pools by using developer tests and adding automatically generated tests by running EvoSuite[FZ10] for each project with the default running time, but with multiple coverage metrics[10]. The mutant-test killing matrices were obtained using PIT [CLH+16]. For each project, we run the mutants on the test pools for 48 hours. To reduce execution time, we select the classes processed during that time lapse.

Table 4.2 records the total number of mutants, number (and percentage) of killable and subsuming mutants, and number of test cases conforming to the mutant-test killing matrices. Please note that the difference on the ratio of subsuming mutants with previous research [PHH+16; ADO14; KAD+14] is due to the inclusion of all mutually subsuming mutants. As already explained, we include all subsuming mutants to avoid misleading our learner.

[7]https://commons.apache.org
[8]https://github.com/JodaOrg/joda-time/
[9]https://github.com/jhy/jsoup
[10]LINE:BRANCH:MUTATION:OUTPUT:METHOD:CBRANCH

4.5.2 Baselines

We consider 2 baselines. The first one is the *Random* mutant sampling that samples uniformly from the entire set of mutants. The second baseline is a Decision Tree classification based on the features proposed by related work [CPB+20].

Previous works showed a strong connection between mutant utility and surrounding code (utility captured through CFG, data flows, AST, etc. features). Thus, we use the mutant features to predict subsuming mutants in both C and Java. Features belong to 4 categories: Mutant Type related features, Control-Flow graph related features, Control and Data dependency related features, and AST related features. In total we used the 28 features, used by the related work [CPB+20], for the C programs, and implemented 16 of those features for Java[11]. We excluded features such as AstChildHasIdentifier and AstChildHasLiteral that we found unfeasible to implement in the employed tools, i.e., PIT works at byte-code level making it difficult to identify the original source code expression. Nevertheless, the excluded features were approximated by mutant type.

After extracting the features, following the related work [CPB+20], we trained a stochastic gradient boosted Decision Tree model by using the same configuration as the related work [CPB+20]. We followed the same validation setup for *Cerebro*.

4.5.3 Implementation and Model Configuration

We rely on the *srcML* tool [CM16] to convert source code into an XML format to tag literals, keywords, identifiers, comments, and our mutation annotations. This helps in separating user-defined identifiers and string literals (the largest part of the vocabulary) from language keywords as srcML supports C, Java and other languages. Then, we implement the ID replacement to generate the abstracted code.

We follow the sequence pair generation procedure mentioned in Section 4.3.2 to generate sequences from the abstracted code. These sequences serve as training input for our encoder-decoder model, which we build using *tf-seq2seq* [MAP+15], a general-purpose encoder-decoder framework. Following previous works [TWB+19a; TWB+19b], we configure our model with bidirectional encoder. We use a Gated Recurrent Units (GRU) network [CvMG+14] to act as the Recurrent Neural Network (RNN) cell, which was shown to perform better than possible alternatives (simple RNNs or gated recurrent units) in related prediction tasks [SNL19]. To achieve good performance with acceptable model training time, we utilize AttentionLayerBahdanau [BCS+16] as our attention class, configured with 2 layered

[11] statementComplexity, expressionComplexity, MutantType, BlockDepth, CfgDepth, CfgPredNum, CfgSuccNum, NumInBlock, NumOutDataDeps, NumInDataDeps, NumOutCtrlDeps, NumInCtrlDeps, AstNodeParentType, NumberOfAstParents, AstNodeType, NumberOfAstChildren

AttentionDecoder and 1 layered BidirectionalRNNEncoder, both with 256 units.

To determine an appropriate number of training epochs, we conducted a preliminary study involving a validation set, independent of both, training and test sets that we use in our evaluation. Here we incrementally train the model, with checks after every epoch to monitor model training accuracy. We pursue training the model till the training performance on the validation set does not improve anymore. We found 15 epochs to be a good default for our validation sets. Once model training is complete, we follow the procedure explained in Section 4.3.4 to predict whether an unseen mutant annotation sequence is subsuming or not.

The codebase of C and Java programs with mutant information, abstracted code, and mutant annotation sequences that the encoder-decoder model trains on and predict, with mapping to the original code, are publicly available at `https://github.com/garghub/Cerebro`. In addition to our dataset, we have made available our source code and trained models as well.

4.5.4 Experimental Procedure

In the first experimental part, we evaluate the prediction ability of our approach, answering RQ1, while in the second part, we evaluate cost-effectiveness of *Cerebro*, answering RQs2-4.

First Experimental Part

We start by evaluating the prediction performance of *Cerebro*, and the baselines, using four typical metrics, namely, *Precision*, *Recall*, *F-measure*, and *Matthews Correlation Coefficient* (MCC)[Mat75]. Given a subsuming mutant, if it is predicted as subsuming, then it is a true positive (TP); otherwise, it is a false negative (FN). Given a non-subsuming mutant, if it is predicted as non-subsuming, then it is a true negative (TN); otherwise, it is a false positive (FP). Here, MCC is more reliable to access the quality of prediction models in contrast to others as the classes are of very different sizes, e.g. in case of C programs, 10.2% subsuming mutants (Positives) over 89.8% non-subsuming mutants (Negatives).

The mutants selected by *Cerebro* are the ones predicted as subsuming. For *Decision Trees* baseline, as it computes a probability of a mutant being subsuming, we followed the probability margin convention and considered those mutants whose predicted probability was higher than 0.5 [CPB+20].

To assess the performance we perform a inter-project evaluations. We use 5-folds cross validation, where we evenly split each benchmark in 5 parts (10 programs and 2 projects per fold for C and Java benchmark, respectively). Then, for each benchmark, we repetitively use 1 fold for testing and 4 folds for training (1 part out of 4, is used for validation).

Second Experimental Part

To study the cost and test effectiveness of our approach and the baselines, we simulate a testing scenario where a tester selects a subset of mutants, to use for mutation analysis, and designs tests to kill them. Algorithm 1 provides the pseudo-code of the simulation process we follow in our experiments. It takes as input a set `M` of mutants to analyze, the test pool `P` and a target subsuming mutation score `tMS*`, and returns a test suite `T` that kills every mutant from `M` (or reaches the pre-specified subsuming mutation score). Additionally, it returns the subsuming mutation score obtained by the test suite `T` (`currMS*`), number of analyzed mutants (`analyzedMut`), number of equivalent mutants analyzed (`equivMut`), and number of test executions (`tExec`) required to generate test suite `T` during the simulated mutation testing scenario.

Algorithm 1 Pseudo-code of the simulation procedure to answer RQ2-4.

Input: set of mutants `M`
Input: test pool `P`
Input: target subsuming mutation score `tMS*`
Output: test suite `T` covering mutants in `M`
Output: subsuming mutation score `currMS*` obtained by `T`
Output: `analyzedMut` number of analyzed mutants
Output: `equivMut` number of equivalent mutants analyzed
Output: `tExec` number of test executions

```
1:  T ← ∅
2:  C ← M                                          ▷ set of survived mutants
3:  currMS* ← 0
4:  while currMS* < tMS* and ¬isEmpty(C) do
5:      m ← pickNextMutant(C)
6:      analyzedMut++
7:      if the test pool P can kill mutant m then
8:          t ← randomlyPickTestKilling(m, P)
9:          T ← T ∪ {t}                             ▷ add test t to the suite
10:         tExec += size(C)                        ▷ run t on mutants from C
11:         remove from set C all mutants killed by t
12:     else
13:         equivMut++                              ▷ m is judged as equivalent
14:     end if
15:     currMS* ← calculateMS*(M,T)
16: end while
17: return T, currMS*, analyzedMut, equivMut, tExec
```

The simulation starts by picking (`pickNextMutant`) the top mutant `m`, according to the technique used (*Cerebro*, *Decision Trees*, and *Random*), among survived mutants from set `C` (initialized with all mutants from `M`). It then checks if there exists some test in the test pool `P` that kill `m` (this process simulates a tester picking, analyzing, and designing a test to kill a mutant). If no test kills mutant `m`, we judge it as equivalent and remove it from `C`. Otherwise, we randomly pick one test `t` from the pool that kills `m`. Then, we run the test `t` on every mutant from `C` to check if the same test consequently kills other mutants (killed mutants are then removed from `C`). This process continues by taking the next survived mutant and finding a test to kill it until every mutant in `C` has been killed or until the desired subsuming mutation score is reached. We run this simulation with the set of mutants selected by *Cerebro*, *Decision Trees*, and *Random*, respectively, and use the reported values to compare their cost-benefit performance for answering RQ2-4. Since Algorithm 1 includes some random decisions, we repeat this process 1,000 times for all the approaches.

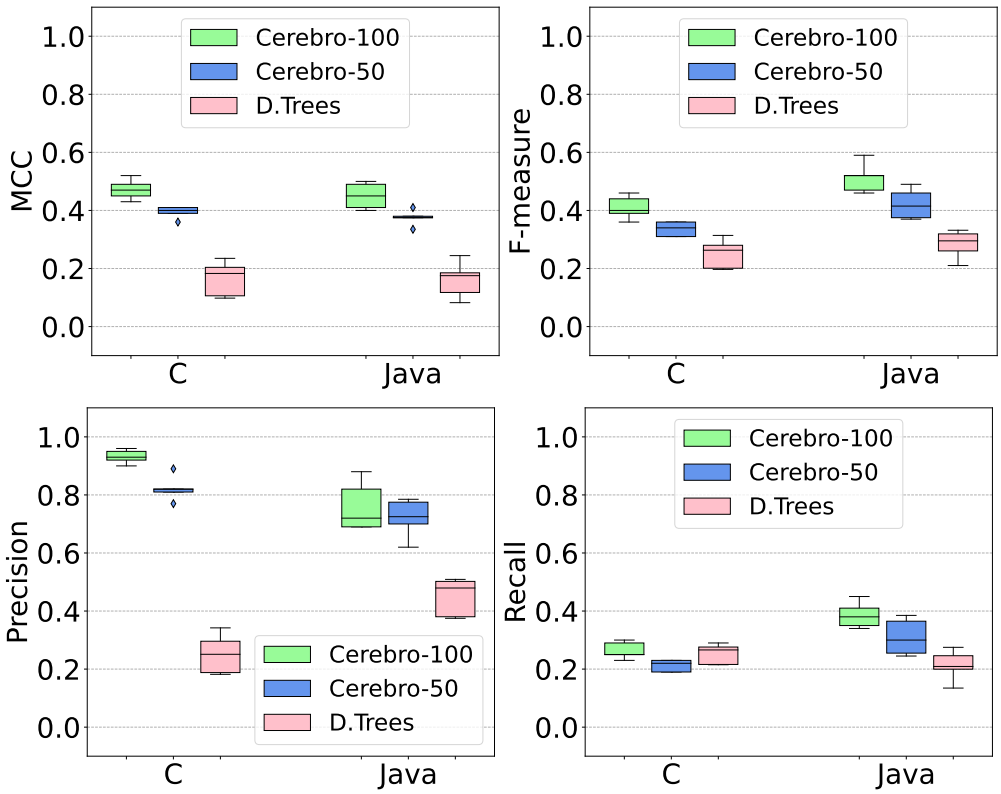
To answer RQ2, we measure the effectiveness (benefit) of the approaches in terms of the *subsuming mutation score* (MS*), *i.e.,*the ratio between killed and total number of subsuming mutants, achieved by the generated test suites when analyzing the selected mutants. The subsuming mutation score reduces the influence of redundant mutants [PHH+16; KAD+14].

For assessing the effectiveness of the approaches, we aim at controlling the number of mutants selected by each tool. In the case of *Cerebro*, the mutants selected are the ones predicted as subsuming by our model. For *Decision Trees* baseline, we rank (in descending order) the mutants according to the predicted probability of being subsuming, and follow the ranking to pick mutants (from highest probability to lowest) for analysis. *Random* baseline randomly ranks the mutants to be selected. Initially, we consider the same number of selected mutants for the 3 approaches, defined as the number of mutants predicted as subsuming by *Cerebro*. For instance, if *Cerebro* predicts 20 mutants as subsuming, then *Decision Trees* and *Random* baselines will also select the top 20 ranked mutants. Our intention is to compare the effectiveness reached by each approach, when the number of selected mutants is equal.

Additionally, we study the number of *equivalent mutants* selected by each approach (as these are an important source of redundancy during mutation testing), as well as, the required number of mutants selected by the baselines in order to reach the same subsuming mutation score as *Cerebro*.

To answer RQ3 and RQ4, we study the effort (cost) required by each approach in two ways. We measure the human effort in terms of the number of *analyzed mutants*, killable or not, that are presented to testers for analysis (*i.e.,*either designing a test to kill these or judging these as equivalent), when applying mutation

Figure 4.4: (RQ1) Prediction Performance Comparison: On average, *Cerebro*-100 outperforms *Decision Trees* by 2.76 times, and 2.81 times higher MCC in C, and Java Benchmark. Moreover, *Cerebro*-50 outperforms *Decision Trees* by 2.29 times, and 2.38 times higher MCC in C, and Java Benchmark. Overall, *Cerebro* outperforms by 2.78 times higher MCC than *Decision Trees*.



testing. Intuitively, for a given set of mutants, the number of analyzed mutants can be considerably smaller than the entire set's size because a test designed by analyzing one mutant can kill other mutants as well. Hence, we also measure the computational effort in terms of the number of *test executions* performed, during the mutation analysis procedure, *i.e.,*we count the test executions required at every step where a new test is created. As for RQ2, here we also study the number of test executions and the number of mutants that require analysis by the baselines, to reach the same subsuming mutation score as *Cerebro*.

4.6 Experimental Results

4.6.1 Prediction Performance (RQ1)

Table 4.3 records the average (and median) performance metrics. Figure 4.4 shows the performance comparison in box plot format showing the distribution of performance indicators (MCC, F-measure, Precision, and Recall) for both approaches in C, and Java Benchmarks.

On average, *Cerebro* obtains a high Precision, i.e., 0.93 and 0.76 (*Cerebro*-100), and 0.82 and 0.72 (*Cerebro*-50) in C and Java benchmarks, respectively. Testers focusing on mutants selected by *Cerebro* can be confident that these are very likely to be subsuming, providing high utility to the testing process. On the other hand, Recall achieved is low, i.e., 0.26 and 0.39 (*Cerebro*-100), and 0.21 and 0.31 (*Cerebro*-50) in C and Java benchmarks, respectively. This indicates that many subsuming mutants are mistakenly predicted as non-subsuming by *Cerebro*. In practice these mutants can still be collaterally killed by other (mutually subsumed) subsuming mutants correctly predicted as subsuming by *Cerebro* (which is often the case, as we will show when answering RQ2 in the following section). Needless to say, any complementary mutation testing and mutant selection technique can be employed to analyze the remaining mutants that are not killed by test suites designed to kill mutants selected by *Cerebro*.

On comparison with baselines, we observe that *Cerebro* clearly achieves much higher prediction performance in comparison to *Decision Trees* in both benchmarks. The differences are statistically significant.[12]

In C-Benchmark, on average, *Cerebro* with its MCC of 0.47 (*Cerebro*-100), and 0.39 (*Cerebro*-50) outperforms *Random* (0.0 MCC). *Cerebro* also outperforms *Decision Trees*, on average, with 2.76 times higher MCC and 64% improvement in F-measure. It is worth mentioning that while *Cerebro* achieves 3.72 times higher precision than *Decision Trees*, *Cerebro* also offers an improvement of 4% in Recall over *Decision Trees*.

In Java-Benchmark, on average, *Cerebro* with its MCC of 0.45 (*Cerebro*-100), and 0.38 (*Cerebro*-50) outperforms *Random* (0.0 MCC). *Cerebro* also outperforms *Decision Trees*, on average, with 2.81 times higher MCC, and an improvement of 82% in F-measure, 68.88% in Precision, and 85.71% in Recall.

In summary, *Cerebro* offers an improvement in prediction capability (MCC) of 2.78 times higher than *Decision Trees*.

[12]We compared the MCC values using *Wilcoxon signed-rank test* and obtained a $p - value < 5.07e{-}3$ in comparison to *Decision Trees*. We also compared the MCC values with the *Vargha-Delaney A measure* [VD00] and observed that in all (100%) cases, *Cerebro* significantly outperforms baseline techniques.

4.6.2 Effectiveness Evaluation (RQ2)

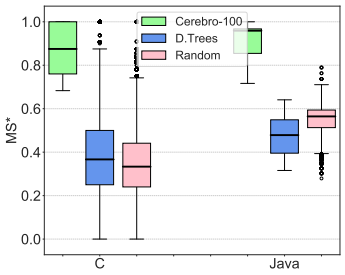
Figure 4.5a and 4.5d show the average subsuming mutation score (MS*) obtained when selecting the same number of mutants (by all techniques). In C-Benchmark, on average, *Cerebro*-100 obtains an MS* of 87.50%, which is 2.39 and 2.63 times higher MS* than *Decision Trees* and *Random*, respectively. Moreover, *Cerebro*-50 obtains an MS* of 71.43%, which is 2.02 and 2.17 times higher MS* than *Decision Trees* and *Random*, respectively.

In Java-Benchmark, on average, *Cerebro*-100 obtains an MS* of 95.90%, which is twice higher than *Decision Trees*, and 69.53% improvement over *Random*. Moreover, *Cerebro*-50 obtains an MS* of 95.66%, which is 2.20 times higher than *Decision Trees*, and 83.33% improvement over *Random*. The differences are statistically significant, according to the computed $p - value$. We also compared them with the *Vargha-Delaney A measure* ($\hat{A}_{12}$) [VD00], showing that *Cerebro* achieves better MS* than *Decision Trees*, and *Random*, in 92.4%, and 95.7% of the cases.

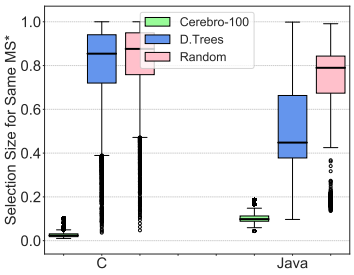
We also study the selection size needed by *Decision Trees* and *Random* to achieve the same MS* obtained by *Cerebro*. For C-Benchmark, Figure 4.5b shows that while *Cerebro*-100 selects only 2.35% of the mutants, *Decision Trees*, and *Random* need to select 85.42% (36.35 times higher), and 87.61% (37.28 times) of the mutants to achieve same MS* as *Cerebro*. Also, Figure 4.5e shows that while *Cerebro*-50 selects only 2.52% of the mutants, *Decision Trees*, and *Random* need to select 34.23% (13.57 times higher), and 42.37% (16.79 times) of the mutants, to achieve same MS* as *Cerebro*. For Java-Benchmark, while *Cerebro*-100 selects 9.85% of the mutants, *Decision Trees*, and *Random* need to select 44.80% (4.55 times higher), and 78.97% (8.02 times) of the mutants, to achieve same MS* as *Cerebro*-100. Also, while *Cerebro*-50 selects 11.60% of the mutants, *Decision Trees*, and *Random* need to select 41.77% (3.60 times higher), and 75.09% (6.48 times) of the mutants, to achieve same MS* as *Cerebro*-50. We obtained a statistically significant $p - value$ and $\hat{A}_{12}$ when compared these values, evidencing that *Cerebro* in more than 98.5%, and 99.1% of the cases, selects fewer mutants than *Decision Trees*, and *Random*.

We also measure the percentage of equivalent mutants selected. For C-Benchmark, Figure 4.5c shows that 1.10% of mutants selected by *Cerebro*-100 are equivalent, whereas 24.44%, and 26.09%, of the mutants selected by *Decision Trees*, and *Random*, are equivalent. Also, Figure 4.5f shows that 4.37% of mutants selected by *Cerebro*-50 are equivalent, whereas 24%, and 26.23%, of the mutants selected by *Decision Trees*, and *Random*, are equivalent. In Java-Benchmark, 9.95% of the mutants selected by *Cerebro*-100 are equivalent whereas for *Decision Trees*, and *Random*, 15.11% (51.86% more), and 19.33% (94.27% more) selected mutants are equivalent. Also, 5.45% of the mutants selected by *Cerebro*-50 are equivalent whereas for *Decision Trees*, and *Random*, 15.86% (2.91 times higher), and 19.26%

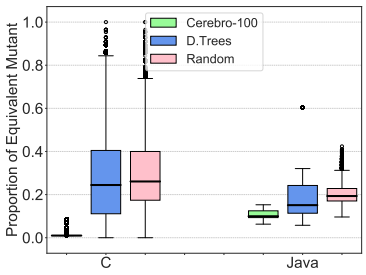
Figure 4.5: (RQ2) Results of the Simulation - Trade off between mutant selection size and MS*.



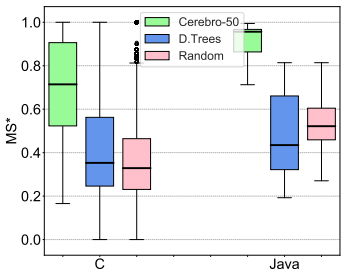
(a) C-Benchmark: For the same mutant selection size, *Cerebro*-100 obtains an MS* of 87.50%, while *Decision Trees*, and *Random* obtains 36.67%, and 33.33%.
Java-Benchmark: *Cerebro* obtains, on average, an MS* of 95.90%, while *Decision Trees*, and *Random* obtains 47.85%, and 56.45%.



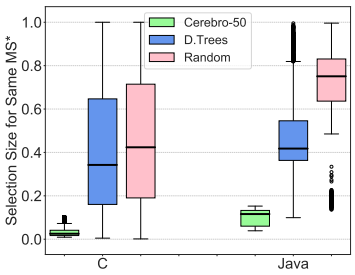
(b) C-Benchmark: to reach the same MS*, *Cerebro*-100 uses 2.35% of the mutants, while *Decision Trees*, and *Random* 85.42%, and 87.61%.
Java-Benchmark: *Cerebro*-100 uses 9.85% of the mutants, while *Decision Trees*, and *Random* use 44.80%, and 78.97%.



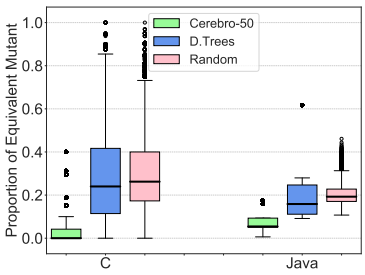
(c) C-Benchmark: *Cerebro*-100 selects 1.10% equivalent mutants, while *Decision Trees*, and *Random* select 24.44%, and 26.09%.
Java-Benchmark: 9.95% of mutants selected by *Cerebro*-100 are equivalent, whereas 15.11%, and 19.33% of mutants selected by *Decision Trees*, and *Random* are equivalent.



(d) C-Benchmark: For the same mutant selection size, *Cerebro*-50 obtains an MS* of 71.43%, while *Decision Trees*, and *Random* obtains 34.23%, and 32.88%.
Java-Benchmark: *Cerebro*-50 obtains, on average, an MS* of 95.65%, while *Decision Trees*, and *Random* obtains 43.48%, and 52.17%.



(e) C-Benchmark: to reach the same MS*, *Cerebro*-50 uses 2.52% of the mutants, while *Decision Trees*, and *Random* 34.24%, and 42.37%.
Java-Benchmark: *Cerebro*-50 uses 11.60% of the mutants, while *Decision Trees*, and *Random* use 41.77%, and 75.09%.



(f) C-Benchmark: *Cerebro*-50 selects 4.37% equivalent mutants, while *Decision Trees*, and *Random* select 24%, and 26.23%.
Java-Benchmark: 5.45% of mutants selected by *Cerebro*-50 are equivalent, whereas 15.86%, and 19.26% of mutants selected by *Decision Trees*, and *Random* are equivalent.

(3.53 times higher) selected mutants are equivalent. The differences are statistically significant. $\hat{A}_{12}$ shows that *Cerebro* in more than 90%, and 98.4% of the cases selects fewer equivalent mutants than *Decision Trees*, and *Random*. These results provide evidence that our approach can reduce significantly this long-standing problem of mutation analysis.

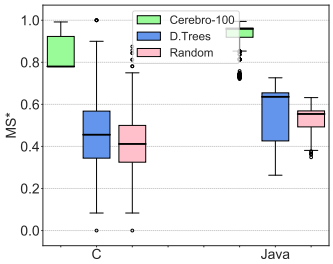
4.6.3 Number of Analyzed Mutants (RQ3)

Figures 4.6a and 4.6d show the average subsuming mutation score (MS*) obtained by each technique for the same number of analyzed mutants. In C-Benchmark, on average, *Cerebro*-100 achieved an MS* of 78%, which is an improvement of 89.41%, and 71.20% over the MS* of *Random*, and *Decision Trees*, respectively. Moreover, *Cerebro*-50 achieved an MS* of 65.75%, which is 2.14 times higher than *Random* and an improvement of 97% over *Decision Trees* In Java-Benchmark, on average, *Cerebro*-100 achieved an MS* of 94.90%, an improvement of 49.24% and 71.21% over *Decision Trees* and *Random*, respectively. Moreover, *Cerebro*-50 achieved an MS* of 95.65%, an improvement of 78.65% and 91.94% over *Decision Trees* and *Random*, respectively. The differences are statistically significant, according to the computed $p - value$ and $\hat{A}_{12}$. We observed that *Cerebro* in more than 96.2%, and 98.4%, of the cases is better than *Decision Trees*, and *Random*.

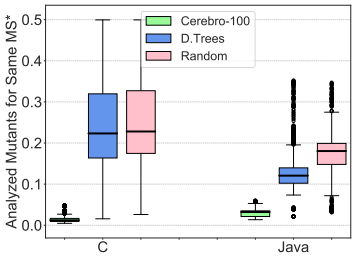
We also study what should be the percentage of mutants to be analyzed by *Decision Trees* and *Random* to achieve the same MS* as *Cerebro*. For C-Benchmark, Figure 4.6b shows that while *Cerebro*-100 analyzes 1.21% mutants, *Decision Trees*, and *Random* need to analyze 22.33% (18.45 times higher), and 22.80% (18.84 times higher) of mutants to reach same MS* as *Cerebro*-100. Also, Figure 4.6e shows that while *Cerebro*-50 analyzes 1.02% mutants, *Decision Trees*, and *Random* need to analyze 11.92% (11.58 times higher), and 13.17% (12.78 times higher) of mutants to reach same MS* as *Cerebro*-50. In Java-Benchmark, while *Cerebro*-100 analyzes 3.22% mutants, *Decision Trees*, and *Random* need to analyze 12.07% (3.75 times higher), and 18.05% (5.61 times higher) of mutants to reach same MS* as *Cerebro*-100. Moreover, while *Cerebro*-50 analyzes 2.52% mutants, *Decision Trees*, and *Random* need to analyze 12.00% (4.76 times higher), and 17.19% (6.82 times) of mutants to reach same MS* as *Cerebro*-50. We obtained a statistically significant $p - value$ and $\hat{A}_{12}$, showing that *Cerebro* in more than 99% of the cases analyzes less mutants than *Decision Trees* and *Random*.

We also measure the percentage of equivalent mutants analyzed by each technique. For C-Benchmark, Figure 4.6c shows that, on average, *Cerebro*-100 analyzes 3.70% equivalent mutants, while 53.33% (14.41 times higher), and 55.56% (15.02 times higher) of the mutants analyzed by *Decision Trees*, and *Random* are equivalent. Also, 4.6f shows that *Cerebro*-50 analyzes 11.31% equivalent mutants, while 50% (4.42 times higher) of the mutants analyzed by *Decision Trees* and *Random* are equivalent. For Java-Benchmark, on average, 33.48% of the mutants analyzed

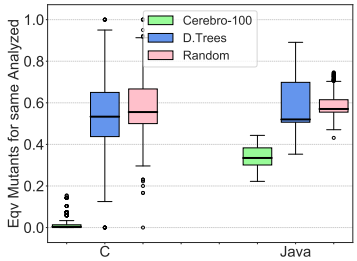
Figure 4.6: (RQ3) Results of the Simulation - Trade off between percentage of mutants analyzed and MS*.



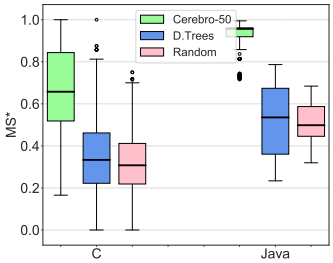
(a) C-Benchmark: Same number of mutants lead to MS* of 78%, 45.56%, and 41.18% for *Cerebro*-100, *Decision Trees*, and *Random*.
Java-Benchmark: *Cerebro*-100 reaches MS* of 94.90%, whereas *Decision Trees*, and *Random* reach 63.59%, and 55.43%.



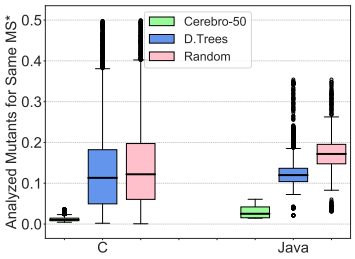
(b) C-Benchmark: *Cerebro*-100 analyzes 1.21% mutants, whereas *Decision Trees*, and *Random* analyze 22.33%, and 22.80% to reach the same MS* as *Cerebro*-100.
Java-Benchmark: *Cerebro*-100 analyze 3.22% mutants, whereas *Decision Trees*, and *Random* analyze 12.07% and 18.05% to reach same MS*.



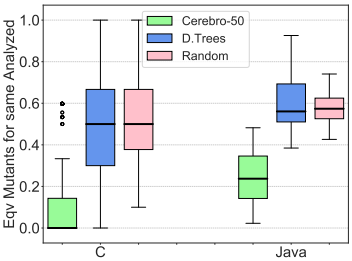
(c) C-Benchmark: 3.70%, 53.33%, and 55.56% of the mutants selected by *Cerebro*-100, *Decision Trees*, and *Random* are equivalent.
Java-Benchmark: 33.48%, 52%, and 57.04% of the mutants selected by *Cerebro*-100, *Decision Trees*, and *Random* are equivalent.



(d) C-Benchmark: Same number of mutants lead to MS* of 65.75%, 33.33%, and 30.77% for *Cerebro*-50, *Decision Trees*, and *Random*.
Java-Benchmark: *Cerebro*-50 reaches MS* of 95.65%, whereas *Decision Trees*, and *Random* reach 53.54%, and 49.83%.

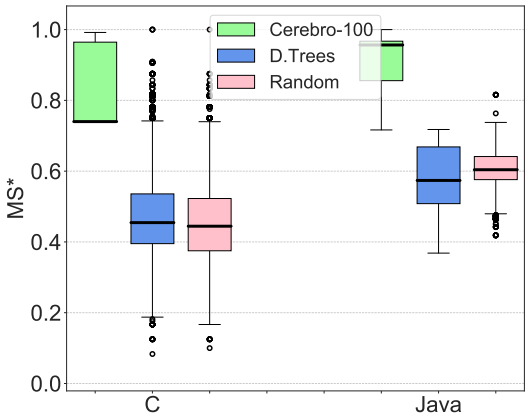


(e) C-Benchmark: *Cerebro*-50 analyzes 1.02% mutants, whereas *Decision Trees*, and *Random* analyze 11.92%, and 13.17% to reach the same MS* as *Cerebro*-50.
Java-Benchmark: *Cerebro*-50 analyze 2.52% mutants, whereas *Decision Trees*, and *Random* analyze 12% and 17.19% to reach same MS*.

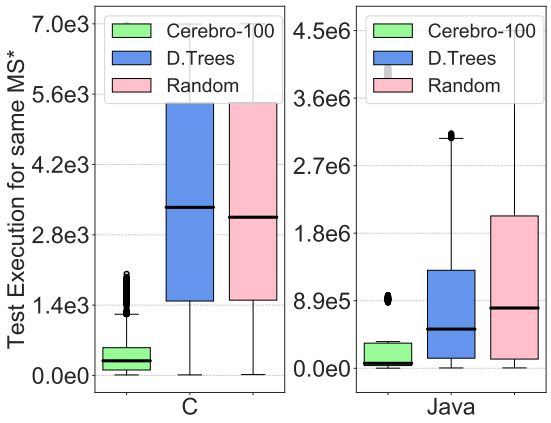


(f) C-Benchmark: 11.31%, 50%, and 50% of the mutants selected by *Cerebro*-50, *Decision Trees*, and *Random* are equivalent.
Java-Benchmark: 23.72%, 56.08%, and 57.38% of the mutants selected by *Cerebro*-50, *Decision Trees*, and *Random* are equivalent.

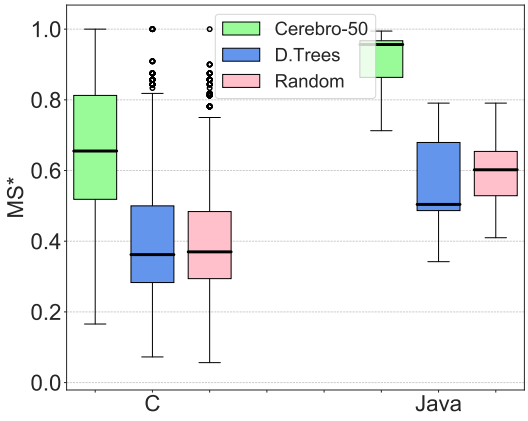
Figure 4.7: (RQ4) Results of the Simulation - Trade off between number of test executions and MS*.



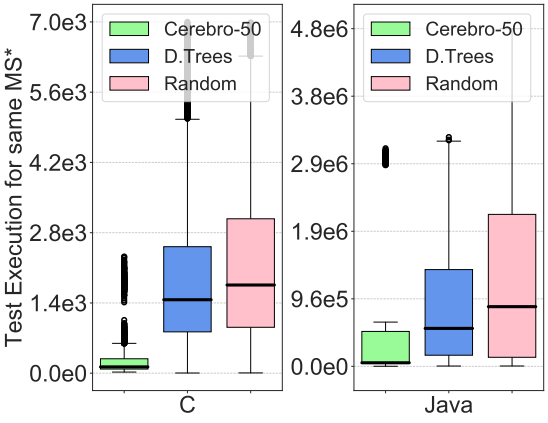
(a) C-Benchmark: For same number of test executions, *Cerebro*-100 obtains an MS* of 74%, while *Decision Trees*, and *Random* obtain 45.45%, and 44.44%.
Java-Benchmark: *Cerebro*-100 obtains MS* of 95.65%, while *Decision Trees*, and *Random* obtain 57.38%, and 60.40%.



(b) C-Benchmark: *Cerebro*-100 requires 291 test executions, while *Decision Trees*, and *Random* require 3,345, and 3,149 to reach the same MS* as *Cerebro*-100.
Java-Benchmark: 65,741 test executions are required by *Cerebro*-100, while *Decision Trees*, and *Random* require 517,040, and 795,304.



(c) C-Benchmark: For same number of test executions, *Cerebro*-50 obtains an MS* of 65.52%, while *Decision Trees*, and *Random* obtain 36.20%, and 36.99%.
Java-Benchmark: *Cerebro*-50 obtains MS* of 95.65%, while *Decision Trees*, and *Random* obtain 50.41%, and 60.21%.



(d) C-Benchmark: *Cerebro*-50 requires 125 test executions, while *Decision Trees*, and *Random* require 1,785, and 2,182 to reach the same MS* as *Cerebro*-50.
Java-Benchmark: 50,622 test executions are required by *Cerebro*-50, while *Decision Trees*, and *Random* require 560,866, and 894,494.

by *Cerebro*-100 are equivalent, while *Decision Trees*, and *Random* analyze 52% (55.31% more), and 57.04% (70.37% more) equivalent mutants. Also, 23.72% of the mutants analyzed by *Cerebro*-50 are equivalent, while *Decision Trees*, and *Random* analyze 56.08% (2.36 times higher), and 57.38% (2.42 times higher) equivalent mutants. This indicates that the baselines suggest the consumption of a large effort to analyze redundant mutants, in comparison to *Cerebro*. The differences are statistically significant. $\hat{A}_{12}$ suggests that *Cerebro* in more than 98% of the cases analyzes fewer equivalent mutants than *Decision Trees*, and *Random*.

4.6.4 Number of Test Executions (RQ4)

Figure 4.7a and 4.7c show the average subsuming mutation score (MS*) when the number of test executions are fixed. In C-Benchmark, on average, *Cerebro*-100 achieves an MS* of 74%, outperforming *Decision Trees*, and *Random* by 62.82%, and 66.52% (*Decision Trees*, and *Random* achieve 45.45%, and 44.44% of MS*). Also, *Cerebro*-50 achieves an MS* of 65.52%, outperforming *Decision Trees*, and *Random* by 80.95%, and 77.14% (*Decision Trees*, and *Random* achieve 36.21%, and 36.99% of MS*). In Java-Benchmark, on average, *Cerebro*-100 and *Cerebro*-50 achieve an MS* of 95.65% in both simulations, an improvement of approx. 67%, and 58% over *Decision Trees*, and *Random* (*Decision Trees*, and *Random* achieve 57.38%, and 60.40% of MS* in first simulation when compared against *Cerebro*-100, and 50.41%, and 60.21% of MS* in the second comparison simulation against *Cerebro*-50). We obtained a statistically significant $p - value$. Also $\hat{A}_{12}$ suggests that *Cerebro* in 94.15%, and 95.7%, of the cases is better than *Decision Trees*, and *Random*.

We also measure the test executions required by the baselines to achieve the same MS* as *Cerebro*. Figure 4.7b shows that, in C-Benchmark, *Cerebro*-100 requires 291 test executions (median), while *Decision Trees*, and *Random* require 3,345, and 3,149. Also, Figure 4.7d shows that *Cerebro*-50 requires 125 test executions (median), while *Decision Trees*, and *Random* require 1,785, and 2,182. This shows that *Cerebro*-100 is 10-12 times less and *Cerebro*-50 is 14-17 times less expensive (computationally) than the baselines.

In Java-Benchmark, *Decision Trees*, and *Random* require 517,040, and 795,304 test executions (median) to achieve the same MS* as *Cerebro*-100, for which 65,741 test executions are required. Moreover, *Decision Trees*, and *Random* require 560,866, and 894,494 test executions to achieve the same MS* as *Cerebro*-50, for which 50,622 test executions are required. This shows that the baselines require 7 to 12 times, and 11 to 17 times higher computational effort than *Cerebro*-100, and *Cerebro*-50.

These differences are statistically significant. $\hat{A}_{12}$ value indicates that in more than 98.7% of the cases, *Cerebro* executes fewer tests than *Decision Trees* and *Random*.

4.7 Discussion

Cerebro is a learning-based method, and thus its performance depends on a number of parameters and design decisions we made. To this end, we discuss the key (intuitive) parameters that make the Machine Translation approach we use effective (Section 4.7.1), together with empirical results demonstrating the potential impact on the model's performance given the design decisions of using unabstracted code sequences (Section 4.7.2), sequences with decreased length during training (Section 4.7.3), and the impact of assuming unkilled mutants as equivalent mutants during testing (Section 4.7.4).

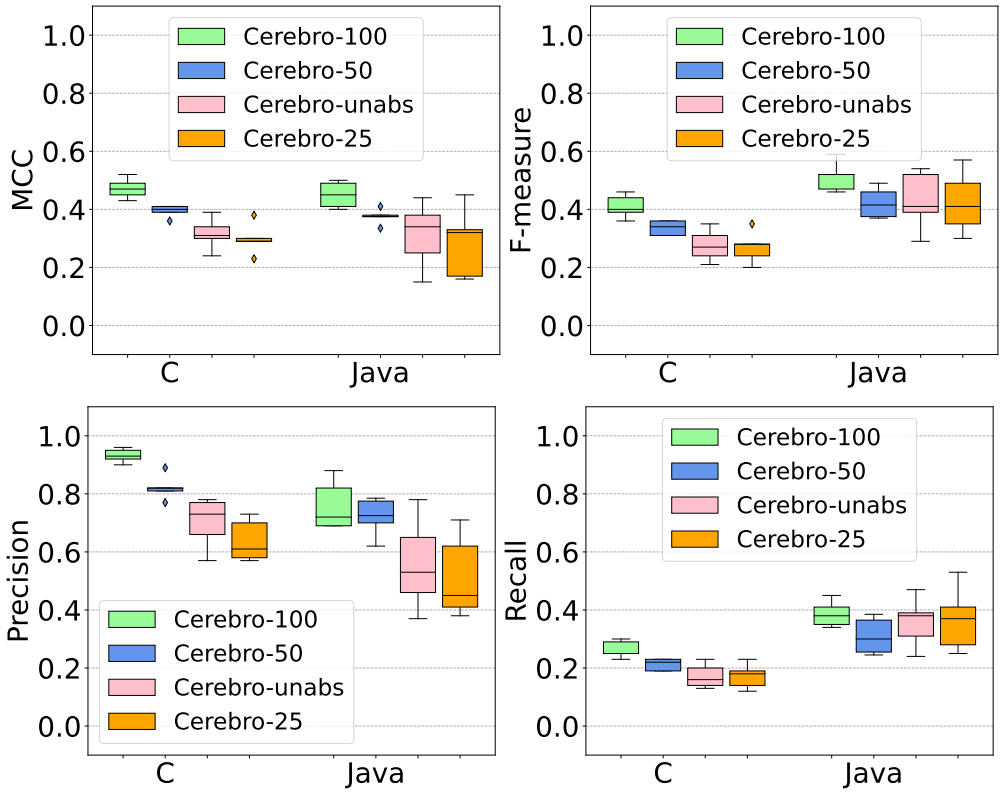
4.7.1 Why *Cerebro* is a good candidate for subsuming mutant prediction?

There are three main factors that make Machine Translation a good candidate for subsuming mutant prediction. The first one is that it learns to select mutants using the exact local context (entire code snippet composed of 50-100 tokens, represented as a sequence), while previous work considers AST and data-flow abstractions [CPB+20], ignoring the exact formulation of the code snippet. In a sense, the key determining factor is the sequence that code tokens appear in the local context (considered code snippet). The second reason is that the machine translator includes a powerful self-attention mechanism, which together with the encoder-decoder architecture makes the learning resistant to noise [TMR+18], and able to learn out of imbalanced data. Overall, previous research has shown that this architecture often makes the best predictions for many NLP tasks [DCL+19]. This is actually the reason why Machine Translation has been successfully used in code analysis tasks such as mutant generation, code clone detection, test assertions generation, etc. The third reason is the diversity of the selected mutants, i.e., Cerebro selects a few mutants per code block, which allows eliminating local redundancies, while spreading testing across the entire code-base.

4.7.2 Impact of removing code abstraction

We analyzed the impact of using unabstracted code sequences to train our models instead of proposed abstracted code sequences and how it affects the model prediction performance (RQ1). In this experiment, we just removed the code comments and kept everything else as it is. We found a prediction performance reduction for projects in both C and Java benchmarks. For C-Benchmark, the model performance deteriorated by 18.9% in MCC, 14.4% in Precision and 18.1% in Recall. For Java-Benchmark, although we found an improvement of 15.4% in Recall, the overall performance deteriorated by 17.9% in MCC and 22.5% in Precision.

Figure 4.8: Impact of the abstraction process and sequence length in *Cerebro*'s prediction performance: On average, MCC is decreased by 18% with unabstracted code and decreased by 24% with sequence length 25.



4.7.3 Impact of reducing the sequence length

We also analyzed the impact of reducing the length of sequences that we use to train our models and how it affects the model prediction performance (RQ1). In this experiment, we reduced the sequence length from 50 tokens per sequence to 25 tokens per sequence. Figure 4.8 and Table 4.4 shows the average and median scores achieved by the models. For simulation details on Effectiveness Evaluation (RQ2), Number of Analyzed Mutants (RQ3) and Number of Test Executions (RQ4), please refer to our online repository. From these results we found that reducing the length of sequences used by the models to train also deteriorated the model prediction performance for projects in both C and Java benchmarks. For C-Benchmark, the model performance deteriorated by 23.5% in MCC, 22.2% in Precision and 18.1% in Recall. For Java-Benchmark, although we found an improvement of 18.7% in Recall, the overall performance deteriorated by 24.7% in MCC and 28.6% in Precision.

4.7.4 Impact of considering subsuming mutants as equivalent, *i.e.,* impact of potential mistakes in evaluation

In our experiments, we considered the mutants that were not killed by our test suite as unkillable a.k.a. equivalent. Although this being an undecidable problem (as we elaborated in Section 2.1.2), we analyzed the impact of what would have happened if the mutants that we considered as equivalent were subsuming instead. Hence, we addressed this by introducing noise in our evaluation, *i.e.,* we assumed 2% equivalent mutants in our evaluation set as subsuming and analyzed the change in performance (MS* achieved) for all the approaches (*Cerebro*, *Decision Trees* and *Random*). We gradually increased the noise percentage from 2% till 10% (*i.e.,* 2%, 4%, 6%, 8%, 10%) and analyzed the change in behaviour for all the approaches (*i.e.,* change in MS*), if it increases or decreases with increase in noise.

We found that *Cerebro*'s and *Decision Trees*' performances are more or less inversely related to the noise in evaluation (Figure 4.9). Higher the noise, lower the MS* achieved by both the approaches (with an exception of 10% noise in C benchmark for *Decision Trees* where *Decision Trees* performed better than in case of 8% noise, as detailed in Table 4.5). For *Random* selection, the performance also deteriorated in most of the cases, with an exception of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where *Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively. Despite the reduction in performance due to introduced noise, *Cerebro* still achieves higher MS* than the baselines (Figure 4.9).

4.8 Threats to Validity

External Validity: Threats may relate to the subjects we used. Although our evaluation expands to both C and Java projects of different sizes, the results may not generalize to other projects or programming languages. We consider this threat as low since we have a large sample of programs, i.e., we perform one of the largest mutation testing studies to date.

Other external threat lies in the operators we used, since our prediction approach might not work for different types of mutants. To reduce this threat, we employ modern mutation tools, for both C and Java that implement a large variety of mutation operators. For the C-Benchmark, taken from [CPC+21], 816 simple operators across 18 categories were considered; while for creating our Java-Benchmark, we consider the group "ALL" of mutation operators provided by PIT [CLH+16], resulting in 112 simple operators across 29 categories.

Internal Validity: Threats may relate to the restriction that we impose on sequence length, *i.e.,*a maximum of *100* tokens. This was done to enable reasonable model training time, approximately *740* hours. Moreover, restricting the sequence length to *50* assisted to reach an appropriate training time of 360 hours. However,

it resulted in a prediction performance deterioration of approximately 15%, as discussed in Section 4.7. Other threats maybe due to the use of machine translation for classification. This choice was made for simplicity, to use the related framework out of the box, similar to the related studies [TWB+19a; TWB+19b]. Still a potential "sequence to class classifier" may yield better results, though such improvements should be marginal given the low number of unexpected labels we get, i.e., on average, 2.15% of the mutants do not get a valid label (4.2% in C and 0.1% in Java).

Threats may also relate to the features we implemented for training the *Decision Trees* baseline. We follow the guidelines provided in [CPB+20], to extract the 16 features for our Java dataset. Unfortunately, many of the 28 features for C programs presented in [CPB+20] depend on the semantic of the C language, that we found unfeasible to be replicated in Java. However, the prediction performance of *Decision Trees* in Java are in line with the results obtained for C, indicating that the impact of this threat is low.

Other internal validity threats could be related to the test suites we used and the mutants considered as subsuming and equivalent. To deal with this issue, we used well-tested programs and state-of-the-art tool to generate extensive pools of tests (KLEE[CDE08], SEMu[CPC+21], and EvoSuite[FZ10]). Since identifying subsuming and equivalent mutants is an undecidable problem, in our experimental setup, we approximate them through an extensive pool of tests. This has been a typical process followed in related mutation testing studies [JH09; ADO14; PHH+16; KAO+16; PCT18]. To be more accurate, our underlying assumption is that the extensive pool of tests used in our experiments are a valid representation of all possible tests that a tester can manually or automatically generate. This assumption allowed us to identify the minimal set of mutants (i.e., subsuming mutants) that a tester needs to kill in order to kill every other killable mutant (i.e., subsumed mutants). Also, we assumed that unkilled mutants are equivalent. Even if this may not be the case, it is likely that the testers guided by mutation won't be able to kill all the killable mutants. Here it must be noted that since Cerebro is quite precise, its feeding with less noisy data, i.e., correct labels, will make it perform better, i.e., more accurate labelling in training will result in better predictions. Nevertheless, we also investigate the impact of having such noisy data and found minor discrepancies, please refer to Section 4.7.4.

Cerebro's use may also pose additional threats. In particular, *Cerebro* required approximately 5 minutes for preprocessing of the projects and 5 minutes for classification (decoding results). While this time overhead is low, compared to the hours of test executions, it may still be important. Although our implementation is non-optimal and involves no parallelism, however our encoding and decoding can easily be parallelized, since mutant instances are independent of one another.

Construct Validity: Our assessment metrics, subsuming mutation score, number of equivalent mutants and number of test executions may not reflect the actual testing cost / effectiveness values. These metrics have been suggested by literature [PKZ+19; ABL+06; KAO+16] and are intuitive, i.e., number of selected and analyzed mutants essentially simulate the manual effort involved by testers, subsuming mutation score the level of covering the test requirements [ADO14; PHH+16], and number of test executions capture the computational effort involved. Here it should be noted that automated test generation tools may reduce this cost but they require testers to check the related test oracles. Similarly, equivalent detection techniques and related heuristics may also reduce the manual effort involved [KPP+16; KPM15]. Though, in C we applied TCE (Trivial Compiler Equivalence) [KPJ+18; HSF+19] to filter out equivalent and duplicated mutants and our approach still provided significant benefits. Similarly, the use of test executions capture the computational effort involved independently of the test execution framework and optimizations used [WXS+17; ZMK13; CZ18; PKZ+19], the machines and the level of parallelization used during test execution. Nevertheless, the differences are substantial making such threats unlikely to happen. Overall, we mitigate these threats by following suggestions from mutation testing literature [PKZ+19; ABL+06; KAO+16], using state-of-the-art tools, performing several simulations, forming very large and diverse test pools, and got consistent and stable results across our subjects.

4.9 Data Availability

The dataset consisting of the codebase gathered for the 48 C programs and 10 Java projects, generated mutants with information in json file format for every program/project with mutant Id, source code file name, mutation type, and line number, subsuming mutant labels, abstracted source code, mutant annotation sequences in pairs, and trained models, along with *Cerebro*'s source code, is publicly available in our GitHub repository[13].

4.10 Conclusion

In this chapter, we presented *Cerebro*, a method that learns to select subsuming mutants (a subset of mutants that subsumes the others, i.e., tests killing them also kill all the mutants of the given mutant set) from given mutant sets. Experiments with 58 programs showed that *Cerebro* identified subsuming mutants with 0.85 precision and 0.33 recall at an inter-project scenario (trained on different projects than the ones it was evaluated). These predictions enable testers to design test cases capable of killing more than two times the subsuming mutants that they

[13]https://github.com/garghub/Cerebro

would kill if they were using either randomly selected mutants or another previously proposed machine learning-based mutant selection technique. At the same time *Cerebro* entails the analysis of 66% fewer equivalent mutants and 90% less mutant executions, indicating a large reduction in the practical effort/cost of the approach.

Table 4.1: Benchmark

Project	**Web URL**	**Version / Commit**
C		
base64, basename, chcon, chgrp, chmod, chown, chroot, cksum, comm, date, df, dirname, echo, expr, factor, false, groups, join, link, logname, ls, md5sum, mkdir, mkfifo, mknod, mktemp, nproc, numfmt, pathchk, printf, pwd, realpath, rmdir, sha256sum, sha512sum, sleep, stdbuf, sum, sync, tee, touch, truncate, tty, uname, uptime, users, wc, whoami [CPC+21]	https://github.com/coreutils/coreutils.git	v8.22
Java		
commons-cli	https://github.com/apache/commons-cli.git	6490067
commons-collections	https://github.com/apache/commons-collections.git	d6eeceb
commons-text	https://github.com/apache/commons-text.git	26a308f
commons-csv	https://github.com/apache/commons-csv.git	865872e
commons-lang	https://github.com/apache/commons-lang.git	2c0429a
commons-io	https://github.com/apache/commons-io.git	c126bdd
commons-net	https://github.com/apache/commons-net.git	33df028
commons-codec	https://github.com/apache/commons-codec.git	475910a
jsoup	https://github.com/jhy/jsoup.git	528ba55
joda-time	https://github.com/JodaOrg/joda-time.git	767c94e

Table 4.2: Test Subjects

Language	**#Programs**	**#Mutants**	**#Killed**	**#Subsuming**	**#Testcases**
C [CPC+21]	48	71,850	49,530 (68.9%)	7,358 (10.2%)	136,412
Java	10	153,823	124,064 (80.6%)	41,219 (26.8%)	21,878

Table 4.3: (RQ1) Prediction Performance of *Cerebro* and *Decision Trees*. On average, *Cerebro* outperforms by 2.78 times higher MCC than *Decision Trees*.

Average (and Median) Performance in C-Benchmark

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Decision Trees	0.17 (0.18)	0.25 (0.26)	0.25 (0.25)	0.25 (0.27)
Cerebro-50	0.39 (0.40)	0.34 (0.34)	0.82 (0.82)	0.21 (0.22)
Cerebro-100	0.47 (0.47)	0.41 (0.40)	0.93 (0.93)	0.26 (0.25)

Average (and Median) Performance in Java-Benchmark

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Decision Trees	0.16 (0.18)	0.28 (0.30)	0.45 (0.48)	0.21 (0.21)
Cerebro-50	0.38 (0.38)	0.42 (0.42)	0.72 (0.73)	0.31 (0.30)
Cerebro-100	0.45 (0.45)	0.51 (0.52)	0.76 (0.73)	0.39 (0.38)

Table 4.4: Impact of the abstraction process and sequence length in *Cerebro*'s prediction performance: On average, MCC is decreased by 18% with unabstracted code and decreased by 24% with sequence length 25.

Average (and Median) Performance in C-Benchmark				
Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Cerebro-100	0.47 (0.47)	0.41 (0.40)	0.93 (0.93)	0.26 (0.25)
Cerebro-50	0.39 (0.40)	0.34 (0.34)	0.82 (0.82)	0.21 (0.22)
Cerebro-unabs	0.32 (0.31)	0.28 (0.27)	0.70 (0.73)	0.17 (0.16)
Cerebro-25	0.30 (0.29)	0.27 (0.28)	0.64 (0.61)	0.17 (0.18)

Average (and Median) Performance in Java-Benchmark				
Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Cerebro-100	0.45 (0.45)	0.51 (0.52)	0.76 (0.73)	0.39 (0.38)
Cerebro-50	0.38 (0.38)	0.42 (0.42)	0.72 (0.73)	0.31 (0.30)
Cerebro-unabs	0.31 (0.34)	0.43 (0.41)	0.56 (0.53)	0.36 (0.38)
Cerebro-25	0.29 (0.32)	0.42 (0.41)	0.51 (0.45)	0.36 (0.37)

Table 4.5: Impact of noise in evaluation on all approaches' performance (MS*): *Cerebro*'s and *Decision Trees*' performances are more or less inversely related to the noise in evaluation. For *Random* selection, the performance also deteriorated in most of the cases, with exceptions of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where *Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively.

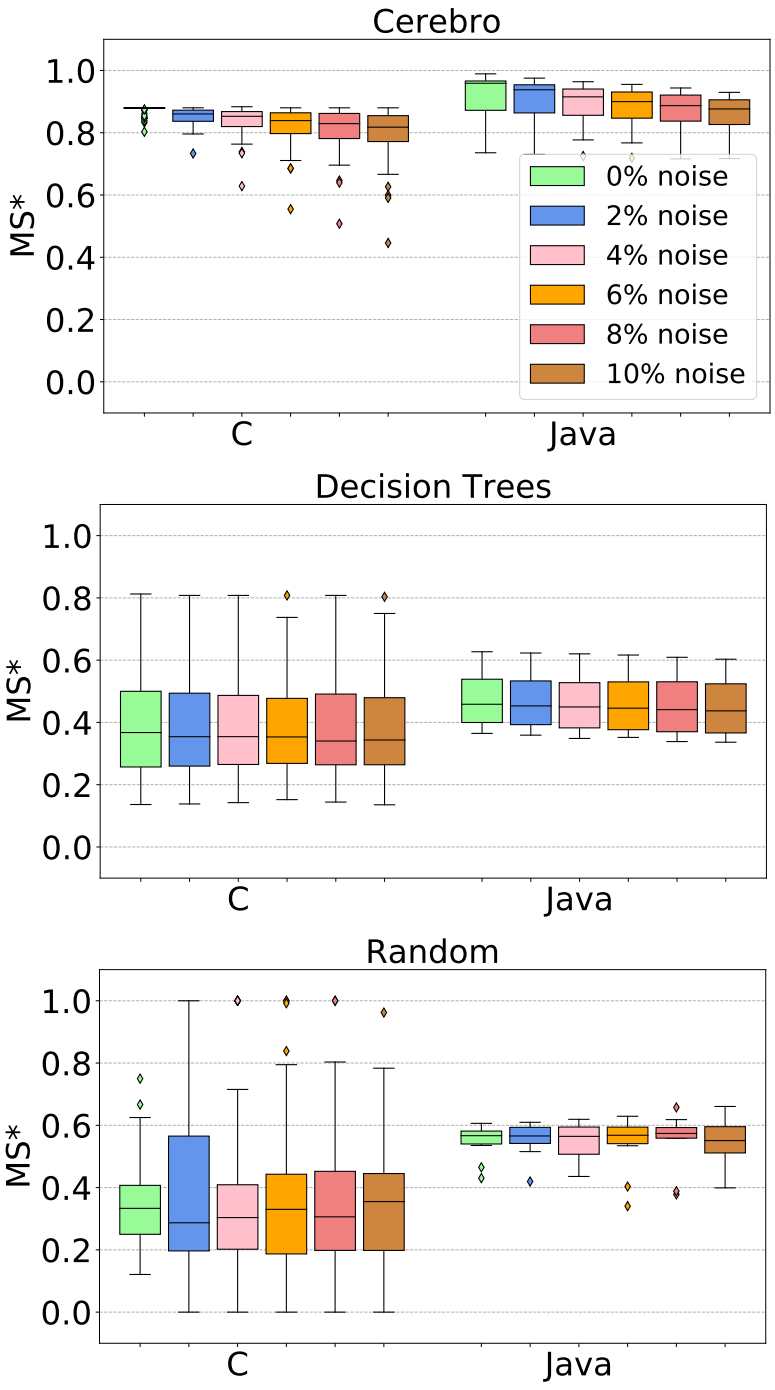
Performance Change % (Median) in MS* w.r.t. noise for C-Benchmark

Noise (%)	***Cerebro***	***Decision Trees***	***Random***
2%	↓ -2.24%	↓ -3.61%	↓ -13.91%
4%	↓ -3.10%	↓ -3.59%	↓ -8.89%
6%	↓ -4.67%	↓ -3.83%	↓ -0.95%
8%	↓ -5.78%	↓ -7.40%	↓ -8.17%
10%	↓ -7.06%	↓ -6.53%	↑ +6.48%

Performance Change % (Median) in MS* w.r.t. noise for Java-Benchmark

Noise (%)	***Cerebro***	***Decision Trees***	***Random***
2%	↓ -2.19%	↓ -1.17%	↓ -0.16%
4%	↓ -4.55%	↓ -1.89%	↓ -0.39%
6%	↓ -6.15%	↓ -2.76%	↑ +0.23%
8%	↓ -7.50%	↓ -3.76%	↑ +1.26%
10%	↓ -8.61%	↓ -4.63%	↓ -2.80%

Figure 4.9: Impact of noise in evaluation on all approaches' performance (MS*): *Cerebro*'s and *Decision Trees*' performances are more or less inversely related to the noise in evaluation. For *Random* selection, the performance also deteriorated in most of the cases, with exceptions of 10% noise in C benchmark, and 6% and 8% noise in Java benchmark where *Random*'s performance improved by 6.48%, and 0.23% and 1.26% improved MS*, respectively.



5

Seeker: Efficient Class Specification Inference

Specification inference techniques aim at automatically inferring sets of program assertions that capture the exhibited software behavior, often by generating and filtering assertions through dynamic test executions and mutation testing. Although powerful, such techniques are computationally expensive due to the large codebase that need to be assessed, and the large number of test cases and code mutants that require execution. In this study, we introduce the notion of Assertion Inferring Mutants*, and perform a study demonstrating that these mutants are sufficient for assertion inference, and correspond to a small subset (12.95%) of the mutants used by mutation testing tools. Moreover,* Assertion Inferring Mutants*, which are well-suited for assertion inference, are significantly different (71.59%) from the* subsuming mutants*, frequently cited by mutation testing literature. We also show that* Assertion Inferring Mutants *can be statically approximated via a learning based method. In particular, we propose* Seeker*, an approach that predicts* Assertion Inferring Mutants *with 0.79 Precision and 0.49 Recall. We further evaluate* Seeker *on 46 programs and demonstrate that it enables a comparable inference capability (missing only 12.49% of assertions) with the full mutation analysis, while significantly reducing the execution cost (*Seeker *achieves 46.29* times *faster inference). At the same time,* Seeker *is more efficient (2.5* times *faster) and more effective (36% more inferred assertions) than other mutant selection strategies used as the baselines. More importantly,* Seeker *enables assertion inference techniques to scale on subjects where full mutation testing is prohibitively expensive and other mutant selection strategies do not lead to acceptable assertion inference.*

Contents

5.1 Introduction

Software specifications aim at describing the software's intended behavior, and can be used to distinguish correct from incorrect software behaviors. While these are typically described informally (e.g., via API documentation), specifications become significantly more useful when expressed formally as executable constraints/assertions. Executable specifications are typically expressed as code assertions for various program points, such as method preconditions and postconditions, that must hold true at the corresponding program points during execution. Program assertions are known to be useful in many software engineering tasks, e.g., test generation [dPX+06; TdH08], bug finding [LCC+05; PLE+07] and automated debugging [DEG+06; LB12; PKL+09]. However, they are tedious to write and maintain, and as a result developers often elude providing them [BGK+18; WTM+20].

To address this issue, different techniques that automatically infer assertions for specific program points have been proposed [TJT+20; MPA+21; MdA22]. These techniques generate candidate assertions, and use dynamic test executions to determine which assertions are consistent with the behavior exhibited by a provided test suite, and mutation testing to discard ineffective/weak assertions that are unable to detect artificially seeded faults (mutants), i.e., assertions that are never falsified during mutants' execution. Though powerful, these techniques are computationally expensive due to the large number of assertions to analyze, and the large numbers of tests and mutants that have to be executed. The problem is further escalated when working with large programs, as the number of mutants grows proportionally to the program size. For instance, the state of the art technique SpecFuzzer [MdA22] times out (requires more than 90 minutes to run) in programs with 180 lines of code.

To reduce the computational demands, it is imperative to limit the number of mutants involved (fewer mutants result in fewer executions). Interestingly, we find that the majority of the mutants used by the existing assertion inference techniques are redundant, meaning that discarding these mutants does not impact the quality of inferred assertions. We thus introduce the notion of *Assertion Inferring Mutants*, the subset of mutants produced by a mutation testing tool that is sufficient to effectively identify relevant candidate assertions (i.e., the assertions that fail at least once on mutants).

We demonstrate that *Assertion Inferring Mutants* represent 12.95% of the mutants supported by Major [JSK11] (the mutation testing tool employed in previous studies), allowing for drastic assertion inference overhead reductions. At the same time, *Assertion Inferring Mutants* are significantly different from *subsuming mutants* (which have been studied in the literature [PKZ+19; GOD+22]) with 71.59% of them not being subsuming. This means that subsuming mutant selection techniques are ineffective for assertion inference, as they would miss many

assertions (48.53% according to our results).

Thus, we propose *Seeker*[1], a learning-based approach to statically identify assertion inferring mutants given their contextual information. In particular, *Seeker* learns the associations between mutants and their surrounding code with respect to the assertion inference task. This means that our learning scope is the area around the mutation point that locally identifies the mutants that are most likely to be useful for assertion inference.

Seeker operates at the lexical level, with a simple code pre-processing that represents mutants and their surrounding code as vectors of tokens with all user-defined identifiers (e.g., variable names) replaced by predefined and predictable identifier names. This representation allows us to restrict the related vocabulary and the learning scope to a relatively small number of tokens around the mutation points enabling inter-project predictions. Code embeddings extracted from an encoder-decoder architecture [KB13] that we train on code fragments, are extracted and learned with corresponding labels using a classifier [Bre01].

We implement *Seeker* and evaluate its ability to predict *Assertion Inferring Mutants* on a large set of 46 programs, composed of 40 taken from previous studies [TJT+20; MPA+21; MdA22] and 6 large Maven projects taken from GitHub, to evaluate scalability. Our results demonstrate that *Seeker* can statically select *Assertion Inferring Mutants* with 0.79 Precision and 0.49 Recall, overall yielding 0.58 MCC[2]. At the same time, since *Seeker* selects fewer mutants than previous work, it improves assertion inference scalability allowing it to run on all the projects we considered where previous work failed.

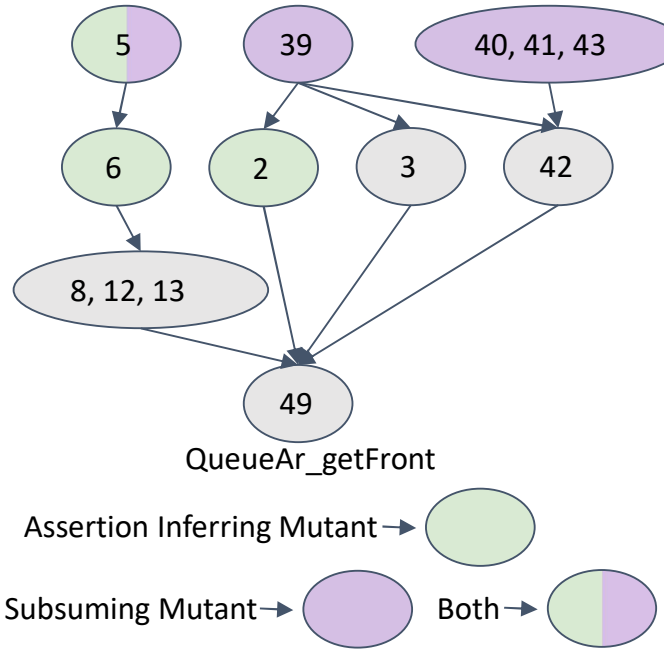
Surprisingly, by performing assertion inference based only on *Seeker*'s predicted mutants (instead of all mutants), we reduce assertion inference time (wall clock) by 46.29 times with only 12.49% assertion missed. Additionally, when comparing with randomly selected sets of mutants (same number as those selected by *Seeker*), we observe a clear superiority of *Seeker* in terms of effectiveness, i.e., *Seeker* infers 36% more assertions while taking approximately equal amount of execution time as *Random Mutant Selection*.

Finally, we show that *Seeker*'s inferring capabilities are almost complete as it infers 96.15% of ground truth assertions, (i.e., the complete set of assertions that were manually validated) while *Random Mutant Selection* only infers 19.23% of them. More importantly, *Seeker* enables assertion inference techniques to scale by allowing its operation on all 6 real-world subjects we selected, where full mutation testing is prohibitively expensive. In half of these subjects, *Random Mutant*

[1]Our approach's name comes from a seeker's role in the fictional sport of Quidditch invented by the author *J.K. Rowling* for her fantasy book series *Harry Potter* [Row97].

[2]*Matthews Correlation Coefficient* (MCC) [YS20] is a reliable metric of the quality of prediction models [SBH14b], relevant when the classes are of different sizes, e.g., *12.95% Assertion Inferring Mutants* in total (in comparison to 87.05% low utility mutants), for subjects in our dataset.

Figure 5.1: Mutant subsumption hierarchy for the subject `QueueAr_getFront` showing the positions of *Assertion Inferring Mutants* and *Subsuming Mutants*



Selection does not lead to any assertion inference and is subsumed by *Seeker* in the other half of the subjects.

5.2 Illustrative Example

Figure 5.1 shows the mutants generated for the method `getFront()` of class `QueueAr`, one of our subjects. The graph depicts the mutants' subsumption hierarchy, which is a standard way of representing subsumption relations between a set of mutants generated from a given subject. Nodes represent mutants of the subject, and an edge connecting mutant M_1 to mutant M_2 represents the fact that M_2 is subsumed by M_1. In our example, mutant 39 subsumes mutants 2, 3 and 42. Mutually subsuming mutants are typically merged into a single node – e.g., mutants 40, 41 and 43 are mutually subsuming. Our figure highlights in purple the subsuming mutants (those at the top of the hierarchy), and in green the *Assertion Inferring Mutants*.

To analyze the impact that mutation analysis has in the inference process, we first inferred assertions with SpecFuzzer [MdA22] on the subject `QueueAr_getFront` with its default configuration, i.e., using all available mutants. SpecFuzzer inferred *27* assertions, with the assertion filtering step via mutation analysis (step 3 of

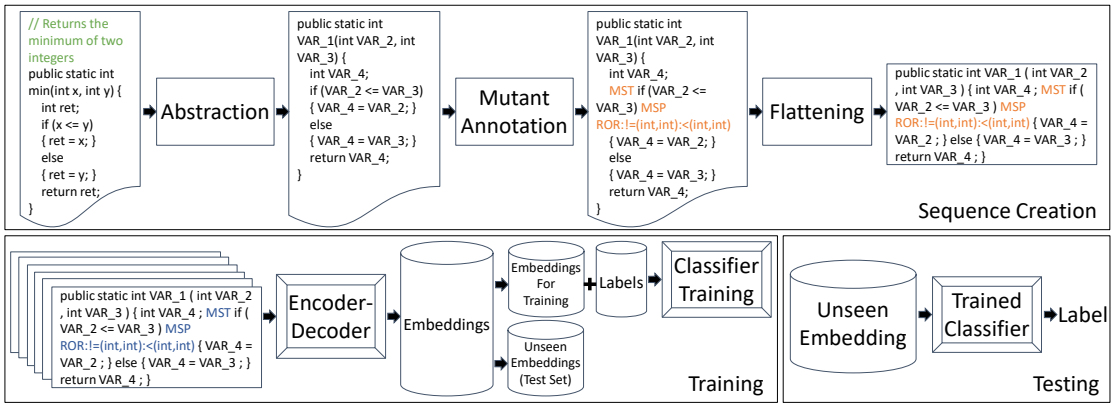
Figure 1.2) taking 91 minutes on our infrastructure (see Section 5.5). By contrast, if we only use subsuming mutants in the filtering step, it only takes 2.5 minutes (36.4 times faster), but produces just 5 assertions. These results evidence that, while reducing the number of mutants to analyze can improve the computational efficiency of the filtering process, subsuming mutants are not appropriate for this task. Intuitively, this is because the initial purpose of subsuming mutants is to minimize the number of tests needed to kill all mutants. In the context of assertion inference one aims instead at inferring all valid assertions that can distinguish the mutants from the original code, that is, generate as many assertions that capture the specific code properties. For instance, in our `QueueAr_getFront` example, 5 out of the 27 inferred assertions are falsified when executing mutant *5*. On the other hand, mutant *6* helps in inferring 21 assertions (i.e., 21 out of the 27 assertions are falsified during mutant *6* execution); while mutant *2* helps in inferring the remaining assertion. In other words, by considering only the five subsuming mutants (i.e., mutants *5*, 39, 40, 41 and 43), and discarding subsumed mutants (including mutants *6* and *2*), the assertion inference results in reporting only 5 assertions, losing 22 strong assertions that could have been inferred by using just the three *Assertion Inferring Mutants* (or the entire pool of mutants at the expense of a significantly higher computational cost).

The above example demonstrates the difference between *Subsuming Mutants* and *Assertion Inferring Mutants*, and the need for an approach that can efficiently identify the latter in order to save valuable time on the mutation analysis step, while maintaining the benefits of assertion inference. The *Seeker* technique that we propose in this chapter is the first mutant selection method especially designed for predicting *Assertion Inferring Mutants*, making existing specification inference techniques more efficient and scalable. As an example, on the *QueueAr_getFront* example, *Seeker* predicts mutant *6* as assertion inferring mutant and helps Spec-Fuzzer to infer 21 assertions (out of 27 assertions when using all mutants), using only a fraction of the computation time (30 seconds) that analyzing all mutants requires (91 minutes).

5.3 Approach

The main objective of *Seeker* is to predict whether a mutant (of a previously unseen piece of code) is likely to be assertion inferring. To make our approach lightweight in terms of engineering and computational effort, we want *Seeker* to be able to (a) learn relevant features of *Assertion Inferring Mutants* without requiring manual feature definition, and (b) do so without costly dynamic analysis of mutant executions. To achieve this, we decompose our problem into two parts: learn a representation of mutants using code embedding techniques, and learn to predict, based on such embeddings, whether the represented mutants are *Assertion Inferring*

Figure 5.2: Overview of *Seeker*: Source code is abstracted and annotated to represent a mutant, which is further flattened to create a space separated sequence of tokens. An encoder-decoder model is trained on token sequences to generate mutant embeddings. A classifier is trained on these embeddings and their corresponding labels (whether or not the mutant is assertion inferring). The trained classifier can then be used for label prediction of an unseen mutant.



Mutants.

5.3.1 Overview of *Seeker*

Figure 5.2 shows an overview of *Seeker*. We decompose our approach into three steps that we detail later on in this section:

1. *Build a token representation*: *Seeker* pre-processes the original code in order to remove irrelevant information and produces abstracted code, which is then tokenized to form a sequence of tokens. Each mutant is ultimately transformed into its corresponding token representation and undergoes the next step.

2. *Representation learning*: We train an encoder-decoder model to generate an embedding, aka vector representation of the mutant. This step is where *Seeker* automatically learns the relevant features of mutants without requiring an explicit definition of these features.

3. *Classification: Seeker* trains a classification model to classify the mutants (based on their embeddings) as *Assertion Inferring Mutants* or not. The true labels used for training are obtained by running SpecFuzzer on the original code, and checking whether the mutants are *Assertion Inferring Mutants* (i.e., which mutants are killed only by assertions coherent with the test-suite).

It is interesting to note that the mutant representation learned by *Seeker* does not depend on the particular set of assertions that SpecFuzzer (or any other assertion inference technique) would check against the mutant. *Seeker* aims instead at learning properties of the mutants (and their surrounding contexts) that are generally useful for assertion inference. This is in line with the recent work on contextual mutant selection [GOD+22; CPB+20] that aims at selecting high utility mutants for mutation testing. This characteristic makes *Seeker* applicable to pieces of code that have not been seen during training. In particular, our experiments reveal the ability of *Seeker* to be effective on projects not seen during training.

The assertion inference technique that is used to build the true labels in the classification task is very important: this technique should produce assertions that capture the software behavior as precisely as possible in order to be capable of distinguishing the buggy versions of the code, i.e., mutants – an essential condition for our classifier to provide relevant prediction results. We use SpecFuzzer [MdA22], a state-of-the-art tool that has been shown to outperform related techniques (GAssert [TJT+20] and EvoSpex [MPA+21]) in assertion inference (SpecFuzzer infers 7 times and 15 times more assertions than GAssert and EvoSpex), with better performance with respect to the ground truth (it obtains better Recall and F-1 score than the existing approaches for producing developer validated assertions).

5.3.2 Training Sequences Generation

A major challenge in learning from raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code [TWB+19b; TPW+19; ABL+19]. In our case, this large vocabulary may hinder *Seeker*'s ability to learn relevant features of *Assertion Inferring Mutants*. Thus, we first abstract original (non-mutated) source code by replacing user-defined entities (function names, variable names, and string literals) with generic identifiers that can be reused across the source code file. During this step, we also remove code comments. This pre-processing yields an abstracted version of the original source code, as the abstracted code snippet in Figure 5.2.

To perform the abstraction, we use the publicly available tool *src2abs* [TPW+19]. This tool first discerns the type of each identifier and literal in the source code. Then, it replaces each identifier and literal in the stream of tokens with a unique ID representing the type and role of the identifier/literal in the code. Each ID <TYPE>_# is formed by a prefix, (i.e., <TYPE>_) which represents the type and role of the identifier/literal, and a numerical ID, (i.e., #) which is assigned sequentially when reading the code. These IDs are reused when the same identifier/literal appears again in the stream of tokens. Although we use src2abs, one can use any other utility that identifies user-defined entities and replaces such with reusable identifiers.

Next, to represent a mutant, we annotate the abstracted code with a mutation

annotation on the statement where the mutation is to be applied. These annotations have the general shape "`MST` statement `MSP MutationOperator`", where `MST` and `MSP` denote mutation annotation start and stop, respectively, and are followed by a `MutationOperator` to indicate the applied mutation operation (as shown in figure 5.2). We repeat the process for every mutant.

Finally, we flatten every mutant (by removing newline, tabs and extra whitespace) to create a single space separated sequence of tokens. Using these sequences, we intend to capture as much code as possible around the mutant without incurring in a prohibitively expensive training time [TWB+19a; TPW+19; GOD+22; GDJ+22; GDP+23]. We found a sequence length of 500 tokens to be a good fit for our task as it does not exceed 24 hours of training time (wall clock) on a Tesla V100 GPU.

5.3.3 Embedding Learning with Encoder-Decoder

Our next step is to learn embeddings, aka vector representations, from mutants' token representation that can later on be used to train a classification model. We develop an encoder-decoder model, a neural architecture commonly used in representation learning tasks [KB13]. The key principles of our encoder-decoder architecture are that the encoder transforms the token representation into an embedding and the decoder attempts to retrieve the original token representation from the encoded embedding. The learning objective is then to minimize the binary cross-entropy between the original token representation and the decoded one. Once the model training has converged, we can compute the embedding from any other mutant's token representation by feeding the latter into the encoder and retrieving the output.

We use a bi-directional Recurrent Neural Network (RNNs) [BGL+17] to develop our encoder-decoder, as previous works on code learning have demonstrated the effectiveness of these models to learn useful representations from code sequences [BCB14; GOD+22; GDJ+22; SVL14]. We build *Seeker* on top of *tf-seq2seq* [AAB+16], an established general-purpose encoder-decoder framework. We use a Gated Recurrent Units (GRU) network [CvMG+14] to act as the RNN cell, which was shown to perform better than simpler alternatives (e.g. simple RNNs) both in software engineering and other learning tasks [SNL19; GOD+22]. To achieve good performance with acceptable model training time, we utilize AttentionLayerBahdanau [BCS+16] as our attention class, configured with 2 layered AttentionDecoder and 1 layered BidirectionalRNNEncoder, both with 256 units.

To determine an appropriate number of training epochs for model convergence, we conducted a preliminary study involving a small validation set (independent of both the training and test sets used in our evaluation) where we monitor the model's performance in replicating (as output) the same mutant sequence provided as input. We pursue training the model until the training performance on the validation set does not further improve. We found 10 epochs for the sequences up

to a length of 500 tokens to be a good default for our validation sets.

5.3.4 Classifying *Assertion Inferring Mutants*

Next, we train a classification model in predicting whether a mutant (represented through the embedding produced by the RNN encoder) is likely to be an assertion inferring mutant. The learning objective here is to maximize the classification performance (which we mainly measure with Matthews Correlation Coefficient (MCC)). To obtain our true classification labels, we run an assertion inference technique (viz. SpecFuzzer) using all available mutants and exhaustively determined which mutants are assertion inferring. As for the classification model, we rely on *random forests* [Bre01] because these are lightweight to train and have shown to be effective in solving various software engineering tasks [JRP+19; PMD+20]. We used standard parameters for random forests, viz. we set the number of trees to 100, use Gini impurity for splitting, and set the number of features (i.e., embedding logits) to consider at each split to the square root of the total number of features.

Once the model training has converged, we can use the random forest to predict whether an unseen mutant is likely to be assertion inferring. For the actual classification, we make the mutant go through the pre-processing pipeline to obtain its abstract token representation, then feed it into the encoder-decoder architecture to retrieve its embedding and finally input it into the classifier to obtain the predicted label (assertion inferring or not).

5.4 Research Questions

We start our analysis by investigating the prediction performance of *Seeker* to select *Assertion Inferring Mutants* and compare whether these can be approximated by other sets of mutants, namely, *subsuming mutants*. Thus, we ask:

RQ1 *Prediction Evaluation:* How effective is *Seeker* in predicting *Assertion Inferring Mutants*? Can subsuming mutants approximate them?

To determine which mutants are assertion inferring (i.e. those killed by at least one assertion), we consider the dataset provided by *Molina et al.* [MdA22] and execute SpecFuzzer, a state of the art assertion inference technique, on the 40 subjects without discarding any mutant. Then, we analyze the performance of *Seeker* in identifying these mutants. We compare the results with the set of subsuming mutants since they form the main objective of mutant selection [KAO+16; GOD+22; PHH+16] with numerous strategies targeting them [MBK+18; GZY+17; GOD+22].

Since *Seeker*'s predictions might not be perfect, we also assess its performance in the context of assertion inference, and contrast it with other mutant selection strategies, namely, *random mutant* selection and *subsuming mutants*. We consider

random mutant selection since it is an untargeted method that is often superior to many mutant selection strategies [GAA+16; ZHH+10] and is considered by the literature as a strong baseline [KAO+16; GOD+22; CPB+20]. Hence, we check the effectiveness (completeness w.r.t. to using all mutants) and efficiency (how much time is required) of SpecFuzzer [MdA22] when utilizing the different mutant subsets over all supported mutants. Therefore we ask:

RQ2 *Inference Evaluation:* How effective and efficient is *Seeker* in comparison to subsuming, randomly selected and all mutants baseline methods with respect to the assertion inference task?

For this task, we re-execute SpecFuzzer on the 40 subjects, by selecting the mutants following *Seeker* and our two baseline mutant selection techniques (subsuming and random mutant selection), and compare its performance when executing SpecFuzzer without discarding any mutant.

In their work [MdA22], *Molina et al.* carefully studied the subjects and manually produced corresponding *Ground Truth* assertions capturing the intended behavior of the subjects. SpecFuzzer was able to infer the ground truth assertions for 26 subjects, when all mutants were considered for assertion inference. Hence, we also compared the effectiveness of all three mutant selection techniques (as explained in the RQ2) in inferring *Ground Truth* assertions. Hence, we ask:

RQ3 *Ground Truth Evaluation:* How *Seeker* compares with the subsuming and randomly selected mutants in terms of inferred ground truth assertions?

The analysis of the above research questions was feasible because we considered subjects from Molina et al. [MdA22], where SpecFuzzer was able to infer assertions considering all mutants of the corresponding subjects. In order to investigate if *Seeker*'s predicted mutants can help SpecFuzzer to scale, i.e., if considering only *Seeker*'s predicted mutants can aid SpecFuzzer to infer assertions in cases where SpecFuzzer would time out if all mutants are considered, other subjects must be taken into account. We thus conduct experiments on 6 subjects from GitHub (Table 5.1) where SpecFuzzer timed out. We also compare SpecFuzzer's performance when it considers *Seeker*'s predicted mutants vs an equal number of randomly selected mutants. Hence, we ask:

RQ4 *Scalability Evaluation:* Can *Seeker* improve the scalability of assertion inference techniques?

5.5 Experimental Setup

5.5.1 Data and Tools

We select 46 Java methods; 40 subjects used in previous studies [TJT+20; MPA+21; MdA22] for evaluating *Seeker*'s performance in RQ1-3, and 6 larger

subjects from GitHub for the scalability evaluation in RQ4. In their study, *Molina et al.* [MdA22] manually constructed *Ground Truth* assertions capturing the intended behavior of these 40 subjects. We use these assertions to answer RQ3.

Table 5.1 records the details of our dataset. For each method analyzed, it reports the total number of mutants generated, the number of *Assertion Inferring Mutants*, the total number of inferred assertions when using all the mutants (i.e., without mutant selection) as well as the number of ground truth assertions produced.

To perform mutation testing we use Major [JSK11], and to construct comprehensive test suites (and improve the chances to infer true assertions), we use EvoSuite [FA11] and Randoop [PLE+07] to augment the developer test suites, similarly to what was done by previous work [MdA22].

5.5.2 Experimental Procedure

To answer our RQs we execute SpecFuzzer to infer assertions for all subjects (Table 5.1) with its default setup, i.e., using all mutants to filter candidate assertions during the mutation analysis step (Figure 1.2). We also determine *Assertion Inferring Mutants* and *Subsuming Mutants* from SpecFuzzer execution logs for the 40 subjects used in RQ1-3. Once the mutants are labeled, we re-execute SpecFuzzer by employing the following 3 mutant selection techniques:

- *Subsuming Mutant Selection.* We execute SpecFuzzer by only considering subsuming mutants for mutation analysis.
- *Seeker.* We train models on *Assertion Inferring Mutants* and perform k-fold cross validation (where k = 5) at the project level, i.e., we train on 32 subjects and evaluate/test on 8 unseen during testing subjects, and repeat 5 times. Once we get the predictions for all 40 subjects, we re-execute SpecFuzzer by only considering the mutants predicted as assertion inferring.
- *Random Mutant Selection.* We randomly select an equal number of mutants (equal to the number of mutants predicted as assertion inferring) from the original set of mutants and re-execute SpecFuzzer by only considering these randomly selected mutants. We repeat this step 10 times to eliminate the chances to report coincidental results. We report the median case results.

To answer *RQ1*, we compute the Prediction Performance Metrics of *Seeker* in order to show its learning ability. This is a sanity check that our prediction modeling framework indeed manages to predict something well. *Seeker*'s predictions can result in four types of outputs. Given a mutant that is assertion inferring, if it is predicted as assertion inferring, then it is a true positive (TP); otherwise, it is a false negative (FN). Vice-versa, if a mutant that does not infer any assertion is predicted as assertion inferring, then it is a false positive (FP); otherwise, it is a true negative (TN). We compute the traditional evaluation metrics, *i.e.,Precision*,*Recall*, and *Matthews Correlation Coefficient (MCC)* to evaluate the performance of prediction models. Here, MCC is more reliable than others as the classes are of very different

sizes, *i.e.,*we have *12.95% Assertion Inferring Mutants* (Positives) in total, for the 40 subjects in the dataset (Table 5.1).

However, prediction results do not reflect the end-task (assertion inference) performance since mutants are not independent, there are large overlaps between the tests and assertions that lead to mutant kills. Thus, to answer *RQ2*, we measure the cost of the employed mutant selection technique, i.e., how many of the assertions inferred when all mutants are considered, are *not* inferred when mutant selection is used, and the benefit gained, i.e., the improvement in terms of wall clock time.

To answer *RQ3*, we check the results of RQ2 and compare how many Ground Truth assertions SpecFuzzer infers with each mutant selection technique. It should be noted that it was able to infer the ground truth assertions for 26 subjects out of the 40, when all mutants were considered for mutation analysis. Hence, we analyze the results only for these 26 subjects.

To answer *RQ4*, i.e., if *Seeker*'s predicted mutants can help SpecFuzzer to infer assertions for 6 subjects where it was not able to infer any assertion (timed out when all mutants were considered for analysis), we retrain *Seeker* on all 40 subjects (with available labeled mutants) and predict likely *Assertion Inferring Mutants* for these 6 subjects. We re-execute SpecFuzzer by only using the predicted mutants and by discarding all other mutants from the original set. Additionally, we randomly select mutants in a similar fashion as before (following RQ2 experimental procedure) and re-execute SpecFuzzer accordingly to compare performance with *Random Mutant Selection*. Thus to answer *RQ4* we measure 1) in how many subjects, the selected mutants lead to assertion inference, and 2) the ratio of assertion inferring mutants from the entire set of mutants.

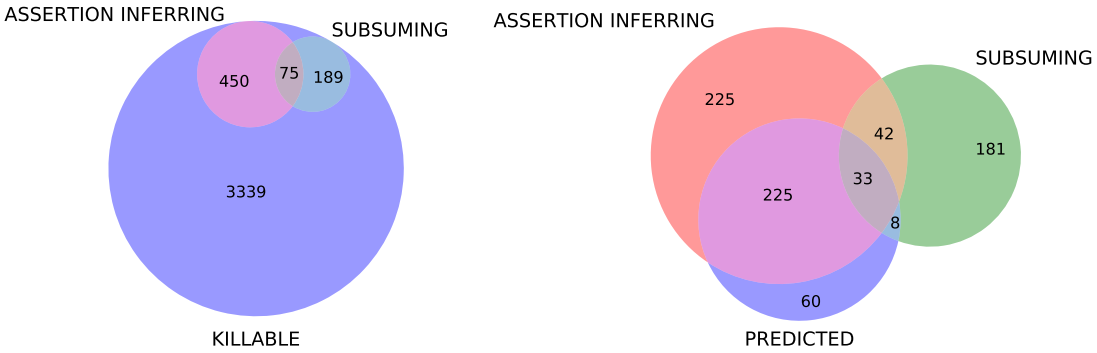
5.6 Experimental Results

5.6.1 Prediction Evaluation (RQ1)

Figure 5.3a shows a Venn diagram recording the distribution of *Assertion Inferring Mutants* and *subsuming* mutant sets. Notice that the set of subsuming mutants is significantly different from the set of *Assertion Inferring Mutants*. Indeed, a small number of subsuming mutants (75 out 264) are also assertion inferring, while a large number of *Assertion Inferring Mutants* (450 out of 525) are not subsuming, showing that subsuming mutant selection is not well suited for the assertion inference task. Moreover, the set of assertion-inferring (resp. subsuming) mutants represents 12.9% (resp. 6.5%) of the killable mutants, suggesting that an effective mutant selection strategy would allow for drastic assertion inference overhead reductions.

Venn diagram from Figure 5.3b shows that *Seeker* detects almost half of *Assertion Inferring Mutants* (258 out 525), while misclassifies just a few of them (68 out of 326). Overall, *Seeker* predicts *Assertion Inferring Mutants* with a predic-

Figure 5.3: Venn diagrams representing the mutant class distribution of *Killable*, *Assertion inferring*, *Seeker*'s predicted, and *Subsuming* mutants



(a) Mutant Class distribution

(b) Prediction distribution

tion performance of 0.79 Precision, 0.49 Recall, and 0.58 MCC, a much better performance than random mutant selection (whose MCC value is 0). Hence, using *Seeker* should provide significant improvements in terms of inferred assertions over baseline methods.

> Answer to RQ1: *Seeker* predicts *Assertion Inferring Mutants* with 0.58 MCC, 0.79 Precision, and 0.49 Recall. The class of subsuming mutants cannot reliably select *Assertion Inferring Mutants* (only 28% of the subsuming mutants are also assertion inferring, while 86% of them are not subsuming).

5.6.2 Inference Evaluation (RQ2)

Table 5.2 records SpecFuzzer's performance w.r.t. assertion inference by employing different mutant sets, i.e, *Subsuming Mutant Selection*, *Seeker*, and *Random Mutant Selection*. The results show that when SpecFuzzer uses *Seeker*'s predicted mutants, it infers 87.51% of total assertions, i.e., only 12.49% of the assertions ares missed (the cost of considering only *Seeker*'s predicted mutants) with 46.29 times faster mutation analysis than using all the mutants (and 2.5 times faster than considering subsuming mutants). *Seeker* enables SpecFuzzer to infer at least one assertion for all subjects, and successfully infers all assertions for 23 subjects.

When SpecFuzzer uses the subsuming mutants, it infers 57.77% of total assertions. It infers all assertions for 5 subjects but fails to infer any for 7 subjects. Although it misses 42.23% of the assertions (the cost of considering only subsuming mutants), diminishing the benefit of an improved mutant analysis time (19.16 times faster than using all mutants).

A good improvement in the mutation testing time is noted when SpecFuzzer uses randomly selected mutants, but it fails to infer 48.53% of total assertions. In 2 cases it infers all assertions and fails to infer any assertion for 2 other cases.

Seeker outperforms both, random and subsuming mutant selection, with a statistically significant[3] sizeable difference.

> Answer to RQ2: *Seeker* enables SpecFuzzer to infer assertions for all subjects, running 46.29 times faster at the expense of 12.49% of the assertions. At the same time, *Seeker* enables SpecFuzzer to infer 36% and 30% more assertions than *Random Mutant Selection* and *Subsuming Mutant Selection*, runs 2.5 times faster than *Subsuming Mutant Selection* and requires similar execution time (wall clock) to *Random Mutant Selection*.

5.6.3 Ground Truth Evaluation (RQ3)

Table 5.3 records SpecFuzzer's performance in ground truth assertion inference by employing the different mutant selection techniques. On considering *Seeker*'s predicted mutants, SpecFuzzer infers almost all (96.15%) ground truth assertions inferred when using all mutants. *Seeker*'s predicted mutants enable SpecFuzzer to infer at least one ground truth assertion for all subjects except for one subject (doublylinkedlistnode_insertRight).

When SpecFuzzer considers only subsuming mutants, it infers 67.31% of all ground truth assertions. It infers all ground truth assertions for 17 subjects but fails to infer any for 8 subjects.

When SpecFuzzer considers randomly selected mutants, it infers 19.23% of all ground truth assertions. It infers all assertions for 5 subjects whereas fails to infer assertions for 21 subjects. *Seeker* outperforms the baselines with a statistically significant sizeable difference.

> Answer to RQ3: *Seeker*'s predicted mutants enable SpecFuzzer to infer ground truth assertions for almost all subjects except one, inferring 96.15% of the total assertions which is superior to both *Subsuming Mutant Selection* (infers 67.31%) and *Random Mutant Selection* (infers 19.23%).

5.6.4 Scalability Evaluation (RQ4)

Table 5.4 records the results of SpecFuzzer's performance in inferring assertions when it employs *Seeker* and *Random Mutant Selection*, for the subjects where

[3]We compared the inferred assertion percentages using Wilcoxon sign-rank-test and obtained a $p - value < 0.05$.

mutation analysis with all mutants timed out. *Seeker* selected 2.99% mutants from the entire mutant set. Among the predicted mutants, 83.33% mutants are assertion inferring. When an equal number of mutants are selected using *Random Mutant Selection*, only 16.67% of mutants selected are assertion inferring. When SpecFuzzer considers only *Seeker*'s predicted mutants for assertion filtering, it infers assertions for all subjects mentioned in Table 5.4 within 16 minutes, on average. On the other hand, for 50% of the subjects (3 out of 6), SpecFuzzer fails to infer any assertion if it uses *Random Mutant Selection*.

Answer to RQ4: *Seeker* enables SpecFuzzer to scale by inferring assertions for all subjects where full mutation analysis timed out and *Random Mutant Selection* failed in 50% of the cases.

5.7 Threats to Validity

External Validity: Threats may relate to the subjects we used. Although our evaluation expands to projects of various sizes, the results may not generalize to other projects. We consider this threat of low importance since we have a large sample of subjects (40 subjects from the previous studies [TJT+20; MPA+21; MdA22] and 6 subjects from GitHub for scalability evaluation). Moreover, our predictions are based on the local mutant context, that has been shown to be determinant of mutants' utility [MCP+21; GOD+22]. Other threats may relate to the assertion inference technique that we used for evaluation. This choice was made since SpecFuzzer is the current state of the art and operates similarly to other techniques (the main differences lie in the grammar used). We consider this threat of low importance since *Seeker* deals with mutation analysis, which is used in the same way by all assertion inference techniques [TJT+20; MPA+21; MdA22], and are directly impacted by the number of mutants involved. Nevertheless, in case other techniques require different predictions, one could re-train, tune and use *Seeker* for the specific method of interest, as we did here with SpecFuzzer.

Internal Validity: Threats may relate to the restriction that we impose on sequence length, i.e., a maximum of *500* tokens. This was done to enable reasonable model training time, approximately *24* hours to learn mutant embeddings on a Tesla V100 gpu. Other threats may be due to the use of *tf-seq2seq* [AAB+16] for learning mutant embeddings. This choice was made for simplicity, to use the related framework out of the box, similar to related studies [TPW+19; GDJ+22]. Other internal validity threats could be related to the test suites we used and the mutants considered as assertion inferring. To deal with this issue, we used well-tested programs and state-of-the-art tools to generate extensive pools of tests (Evosuite [FA11] and Randoop [PLE+07]) as done by previous work [TJT+20;

MPA+21; MdA22]. This is also a typical process followed in mutation testing studies [PHH+16; KAO+16; GOD+22]. To be more accurate, our underlying assumption is that the extensive pool of tests used in our experiments is a reasonable approximation of the program's test executions.

Construct Validity: Our assessment metrics, assertions inferred, ground truth assertions inferred, and incurred time during mutation analysis may not reflect the actual cost / benefit values. These metrics are intuitive, i.e., the inferred assertions are the output of assertion inference techniques, and the incurred time during mutation analysis is the wall clock time these techniques invest in filtering assertions. Overall, we mitigate these threats by following suggestions from mutation testing and assertion inference literature, using state of the art tools, performing several simulations, and confirming consistent and stable results across subjects.

5.8 Data Availability

The dataset consisting of the source code of all subjects, augmented test suites, generated mutants, and SpecFuzzer execution logs, along with *Seeker*'s source code and the tools used in our study, is publicly available in a GitHub repository[4].

5.9 Conclusion

In this chapter, we presented *Seeker*, a method that learns to select *Assertion Inferring Mutants* (a small subset of mutants that is suitable for assertion inference) from given mutant sets. Our experiments on 40 subjects show that *Seeker* identified assertion inferring mutants with 0.58 MCC, 0.79 Precision, and 0.49 Recall. These predictions enable 42.29 times faster inference with minor effectiveness loss (12.49% fewer assertions) compared to the use of all mutants. Similarly, *Seeker*'s predictions infer 96.15% of the total ground truth assertions, which is 40% more than *Subsuming Mutant Selection* and 5 times more than *Random Mutant Selection*. Moreover, *Seeker* enables the assertion inference technique SpecFuzzer to scale on all our large subjects (by inferring assertions where SpecFuzzer failed previously due to timeouts) in comparison to *Random Mutant Selection* which failed to infer any assertion in 50% of the large subjects.

[4]https://github.com/garghub/seeker

Table 5.1: The table records the test subjects, Method details, All Mutants count, Assertion Inferring Mutants count, All Assertions and Ground Truth Assertions inferred when all mutants are used (i.e., Specfuzzer's default execution with no mutant selection).

Subject	Method	All Mutants	Assertion Inferring Mutants	All Assertions	Ground Truth Assertions
ArithmeticUtils_subAndCheck	math.ArithmeticsUtils.subAndCheck	16	2	3	1
BooleanUtils_compare	lang.BooleanUtils.compare	13	13	29	3
composite_addChild	eiffel.Composte.addChild	35	6	185	0
doublylinkedlistnode_insertRight	eiffel.DLLN.insert_right	18	7	16	2
doublylinkedlistnode_remove	eiffel.DLLN.remove	18	4	21	1
Envelope_maxExtent	tsuite.Envelope.maxExtent	56	10	188	0
FastMathNew_floor	math.FastMath.floor	42	18	60	2
IntMath_mod	guava.IntMath.mod	21	15	199	0
listcomp02_insert_r	cozy.ListComp02.insert_r	20	2	1	0
listcomp02_insert_s	cozy.ListComp02.insert_s	20	1	1	0
map_count	eiffel.Map.count	63	3	4	0
map_extend	eiffel.Map.extend	65	9	10	3
map_remove	eiffel.Map.remove	63	1	1	0
MathUtilsNew_copySignInt	math.MathUtils.copySignInt	48	2	16	0
MathUtil_clamp	tsuite.MathUtil.clamp	11	8	12	3
maxbag_add	cozy.MaxBag.add	748	53	49	1
maxbag_getMax	cozy.MaxBag.get_max	749	21	25	1
maxbag_remove	cozy.MaxBag.remove	748	67	26	1
polyupdate_a1	cozy.PolyUpdate.a	54	26	100	2
polyupdate_sm	cozy.PolyUpdate.sm	56	13	73	1
QueueAr_dequeue	daikon.QueueAr.dequeue	66	9	68	3
QueueAr_dequeueAll	daikon.QueueAr.dequeueAll	67	11	69	1
QueueAr_enqueue	daikon.QueueAr.enqueue	66	17	119	2
QueueAr_getFront	daikon.QueueAr.getFront	67	3	27	0
QueueAr_makeEmpty	daikon.QueueAr.makeEmpty	67	20	73	1
ringbuffer_count	eiffel.RingBuffer.count	101	28	119	0
ringbuffer_extend	eiffel.RingBuffer.extend	101	20	148	0
ringbuffer_item	eiffel.RingBuffer.item	101	11	116	0
ringbuffer_remove	eiffel.RingBuffer.remove	101	14	143	0
ringbuffer_wipeOut	eiffel.RingBuffer.wipe_out	101	13	95	1
simple-examples_abs	oasis.SimpleMethods.abs	20	18	30	1
simple-examples_addElementToSet	oasis.SimpleMethods.addElementToSet	3	2	1	1
simple-examples_getMin	oasis.SimpleMethods.getMin	7	6	51	1
StackAr_makeEmpty	daikon.StackAr.makeEmpty	47	13	47	1
StackAr_pop	daikon.StackAr.pop	63	10	35	2
StackAr_push	daikon.StackAr.push	55	6	25	2
StackAr_top	daikon.StackAr.top	50	8	3	0
StackAr_topAndPop	daikon.StackAr.topAndPop	54	13	68	2
structure_foo	cozy.Structure.foo	27	5	1	1
structure_setX	cozy.Structure.setX	26	15	131	1
EmailScanner_findFirst	nibor.autolink.internal.EmailScanner.findFirst	134	Scalability Evaluation (RQ4)*		
EmailScanner_scan	nibor.autolink.internal.EmailScanner.scan	134	Scalability Evaluation (RQ4)*		
IdentityHashSet_isEmpty	leplus.ristretto.util.IdentityHashSet.isEmpty	23	Scalability Evaluation (RQ4)*		
OptionGroup_setRequired	apache.commons.cli.OptionGroup.setRequired	34	Scalability Evaluation (RQ4)*		
OptionGroup_setSelected	apache.commons.cli.OptionGroup.setSelected	34	Scalability Evaluation (RQ4)*		
Scanners_findUrlEnd	nibor.autolink.internal.Scanners.findUrlEnd	111	Scalability Evaluation (RQ4)*		

* Subjects for which SpecFuzzer timed out during mutation analysis are considered for Scalability Evaluation (RQ3).

Table 5.2: RQ2 results - Performance of Assertion Inference

Mutation filtered assertion inference

	With Subsuming Mutant Selection	**With *Seeker***	**With Random Mutant Selection**
Inferred Assertions (per Subject)	57.77%	87.51%	51.47%
Missed Assertions (Cost)	42.23%	12.49%	48.53%
Improvement in Time (Benefit)	19.16 times	46.29 times	47.34 times

Subjects with assertions inferred

Total Subjects# 40	**With Subsuming Mutant Selection**	**With *Seeker***	**With Random Mutant Selection**
Subjects with All assertions inferred	5	23	2
Subjects with No assertion inferred	7	0	2

Table 5.3: RQ3 Results - Inferring Ground Truth Assertions

Ground Truth assertion inference

	With Subsuming Mutant Selection	**With *Seeker***	**With Random Mutant Selection**
Inferred Assertions (per Subject)	67.31%	96.15%	19.23%

Subjects with assertions inferred

Total Subjects# 26	**With Subsuming Mutant Selection**	**With *Seeker***	**With Random Mutant Selection**
Subjects with All assertions inferred	17	25	5
Subjects with No assertion inferred	8	1	21

Table 5.4: RQ4 results - Scalability Evaluation

Assertion Inferring Mutants **(among mutants selected)**		
Mutants selected: 2.99% from the entire mutant set (per subject)	**With *Seeker***	**With Random Mutant Selection**
Assertion Inferring Mutants (among selected mutants)	83.33%	16.67%

Inferred assertions#		
Subject	**With *Seeker***	**With Random Mutant Selection**
EmailScanner_findFirst	85	58
EmailScanner_scan	192	0
IdentityHashSet_isEmpty	3	2
OptionGroup_setRequired	8	8
OptionGroup_setSelected	8	0
Scanners_findUrlEnd	23	0

6

Mystique: Enabling Security conscious Mutation Testing using Language Models

With the increasing release of powerful language models trained on large code corpus (e.g. CodeBERT was trained on 6.4 million programs), a new family of mutation testing tools has arisen with the promise to generate more "natural" mutants in the sense that the mutated code aims at following the implicit rules and coding conventions typically produced by programmers. In this chapter, we study to what extent the mutants produced by language models can semantically mimic the observable behavior of security-related vulnerabilities (a.k.a. Vulnerability-mimicking Mutants*), so that designing test cases that are failed by these mutants will help in tackling mimicked vulnerabilities. Since analyzing and running mutants is computationally expensive, it is important to prioritize those mutants that are more likely to be vulnerability mimicking prior to any analysis or test execution. Taking this into account, we introduce* Mystique*, a machine learning based approach that automatically extracts the features from mutants and predicts the ones that mimic vulnerabilities. We conducted our experiments on a dataset of 45 vulnerabilities and found that 16.6% of the mutants fail one or more tests that are failed by 88.9% of the respective vulnerabilities. More precisely,* 3.9% *of the mutants from the entire mutant set are vulnerability-mimicking mutants that mimic* 55.6% *of the vulnerabilities. Despite the scarcity,* Mystique *predicts vulnerability-mimicking mutants with 0.63 MCC, 0.80 Precision, and 0.51 Recall, demonstrating that the features of vulnerability-mimicking mutants can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features.*

Contents

6.1 Introduction

Research and practice with mutation testing have shown that it is one of the most powerful testing techniques [DLS78; AO08a; GOD+22; Rei99]. Apart from testing the software in general, mutation testing has been proven to be useful in supporting many software engineering activities which include improving test suite strength [CPT+17; ADO14], selecting quality software specifications [TJT+20; MPA+21; MdA22], among others. Though, its use in tackling software security issues has received little attention. A few works focused on model-based testing [BOP11; MFB+08] and proposed security-specific mutation operators to inject potential security-specific leaks into models that can lead to test cases to find attack traces in internet protocol implementations. Other works proposed new security-specific mutation operators that aim to mimic common security bug patterns in Java [LDP+17] and C [NWH+15]. These works empirically showed that traditional mutation operators are unlikely to exercise security-related aspects of the applications and thus, the proposed operators attempt to convert non-vulnerable code to vulnerable by mimicking common real-world security bugs. However, pattern-based approaches have two major limitations. On one hand, the design of security-specific mutation operators is not a trivial task since it requires manual analysis and comprehension of the vulnerability classes that cannot be easily expanded to the extensive set of realistic vulnerability types (e.g. they restrict to memory [LDP+17] and web application [NWH+15] bugs). On the other hand, these mutation operators can alter the program semantics in ways that may not be convincing for developers as they may perceive them as unrealistic/uninteresting [BWB+21], thereby hindering the usability of the method.

With the aim of producing more realistic and natural code, a new family of tools based on language models has recently arisen. Currently, language models are employed for code completion [LLZ+21], test oracle generation [TDS+22], program repair [CKT+21], among many other software engineering tasks. Particularly, language models are been used for mutant generation yielding to several mutation testing tools such as SemSeed [PP21] and DeepMutation [TKW+20]. While these tools are subject to expensive training on datasets containing thousands of buggy code examples, there is an increasing interest in using pre-trained language models for mutant generation [RW22; BSd+22; DP22], e.g. a mutation testing tool *μBERT* [DP22] uses CodeBERT [FGT+20] to generate mutants by masking and replacing tokens with CodeBERT predictions.

Since pre-trained language models were trained on large code corpus (e.g. CodeBERT was trained on more than 6.4 million programs), their predictions are typically considered representative of the code produced by programmers. Hence, we wonder:

Are mutation testing tools using pre-trained language models effective at producing

mutants that semantically mimic the behaviour of software vulnerabilities?

A positive answer to this question can be a promising prospect for the use of these security-related mutants to form an initial step for defining security-conscious testing requirements. We believe that these requirements are particularly useful when building regression test suites for security intensive applications.

The task of analyzing the mutants, and writing and executing tests, in general, is considered expensive. Despite a large number of mutants created, it is well known that many of them are of low utility, i.e., they do not contribute much to the testing process [JH09; KPM10; ADO14]. Due to this, several mutant selection techniques have been proposed to make mutation testing more cost-effective [PKZ+19; OLR+96; ZGM+13; KAO+16; CPB+20]. Therefore, to make our approach useful in practice, we need to filter and select only specific mutants that resemble the behavior of security issues, especially vulnerabilities.

Taking this into account, we propose to enable security-conscious mutation testing by focusing on a minimal set of mutants that rather behave similarly to vulnerabilities a.k.a. *Vulnerability-mimicking Mutants*. Such mutants are the ones that semantically mimic the observable behavior of vulnerabilities, i.e., a mutant is vulnerability-mimicking when it fails the same tests that are failed by the vulnerability that it mimics, proving its existence in the software a.k.a. *PoV* (Proof of Vulnerability). Using *Vulnerability-mimicking Mutants* as test requirements can guide testers to design test suites for tackling vulnerabilities similar to the mimicked ones.

We conducted experiments on a dataset of 45 reproducible vulnerabilities, with severity ranging from high to medium, and found that for 40 out of 45 vulnerabilities, (i.e., for 88.9% vulnerabilities) there exists at least one mutant that fails one or more tests that are also failed by the respective vulnerabilities. More precisely, *3.9%* of the mutants from the entire mutant set are vulnerability-mimicking. Despite being few in quantity, these *Vulnerability-mimicking Mutants* semantically mimic *55.6%* of the vulnerabilities, i.e., these mutants fail the “same" tests that are failed by the respective vulnerabilities that they mimicked.

Since such mutants are very few among the large set of mutants generated, we propose *Mystique*[1], a machine learning based approach that automatically learns the features of *Vulnerability-mimicking Mutants* to identify these mutants *statically*. *Mystique* is very accurate in predicting *Vulnerability-mimicking Mutants* with *0.63 MCC, 0.80 Precision, and 0.51 Recall*. This demonstrates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in manually

[1] *Mystique* is a fictional character appearing in Marvel comics, who can mimic the appearance and voice of a person with exquisite precision, and finds other mutants with similar interests as her, a.k.a. *Brotherhood of Mutants*. More details at https://en.wikipedia.org/wiki/Mystique_(character).

defining any related features. We believe that *Vulnerability-mimicking Mutants* can help in building regression test suites for security intensive applications, and can be particularly useful in evaluating and comparing fuzzing or other security testing tools. In summary, this chapter makes the following contributions:

1. We show that mutation testing tools based on language models can generate mutants that mimic real software vulnerabilities. 3.6% of the mutants semantically mimic 25 out of 45 studied vulnerabilities.

2. We also show that for most of the vulnerabilities (40 out of 45) there exists at least one mutant that fails the one test finding the vulnerability (although not mimicking it).

3. We propose *Mystique*, a machine-learning based approach for identifying *Vulnerability-mimicking Mutants*. Our results show that *Mystique* is very accurate in its predictions as it obtains 0.63 MCC, 0.80 Precision, and 0.51 Recall.

6.2 Motivating Examples

Figures 6.1 and 6.2 show motivating examples of how generated mutants can mimic the behavior of vulnerabilities. Fig. 6.1 demonstrates the example of high severity (7.5) vulnerability CVE-2018-17201[18b] that allows "Infinite Loop", a.k.a., a loop with unreachable exit condition when parsing input files. This makes the code hang which allows an attacker to perform a Denial-of-Service (DoS) attack. The vulnerable code (Fig. 6.1a) is fixed with the use of an "`if`" expression (Fig. 6.1b) to throw an exception and moves out of the loop in case of such an event. Fig. 6.1c shows one of *Vulnerability-mimicking Mutants* in which the "`if`" condition is modified, i.e., the binary operator "`<`" is modified to "`==`". This modification makes the "`if`" condition never executed, nullifying the fix, and behaving exactly the same as the vulnerable code.

Fig. 6.2 demonstrates the example of another high severity vulnerability CVE-2018-1000850[18a] that allows "Directory Traversal" that can result in an attacker manipulating the URL to add or delete resources otherwise unavailable to him/her. The vulnerable code (Fig. 6.2a) is fixed with the use of an "`if`" expression (Fig. 6.2b) to throw an exception in case '`.`' or '`..`' appears in the "`newRelativeUrl`" (Fig. 6.2b). Fig. 6.2c shows one of *Vulnerability-mimicking Mutants* in which the passed argument is changed from "`newRelativeUrl`" to "`name`" which changes the matching criteria, hence nullifying the fix, and introducing same vulnerability behaviour.

Table 6.1: The table records the Vulnerability dataset details that include CVE ID, CWE ID and description, Severity level (that ranges from 0 to 10), number of Files and Methods that were modified during the vulnerability fix, and number of Tests that are failed by the vulnerability a.k.a. Proof of Vulnerability (PoV).

CVE (Vulnerability)	CWE	CWE description (Common Weakness Enumeration)	Severity (0 - 10)	# Files modified	#Methods modified	Failed Tests (PoV)
CVE-2017-18349	CWE-20	Improper Input Validation	9.8	1	1	1
CVE-2013-2186	CWE-20	Improper Input Validation	7.5	1	1	2
CVE-2014-0050	CWE-264	Permissions, Privileges, and Access Controls	7.5	2	5	1
CVE-2018-17201	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	7.5	1	1	1
CVE-2015-5253	CWE-264	Permissions, Privileges, and Access Controls	4.0	1	1	1
HTTPCLIENT-1803	NA	NA	NA	1	1	1
PDFBOX-3341	NA	NA	NA	1	1	1
CVE-2017-5662	CWE-611	Improper Restriction of XML External Entity Reference	7.3	1	2	1
CVE-2018-11797	NA	NA	5.5	1	1	1
CVE-2016-6802	CWE-284	Improper Access Control	7.5	1	1	3
CVE-2016-6798	CWE-611	Improper Restriction of XML External Entity Reference	9.8	1	2	1
CVE-2017-15717	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	2	2
CVE-2016-4465	CWE-20	Improper Input Validation	5.3	1	1	1
CVE-2014-0116	CWE-264	Permissions, Privileges, and Access Controls	5.8	1	4	1
CVE-2016-8738	CWE-20	Improper Input Validation	5.8	1	1	2
CVE-2016-4436	NA	NA	9.8	1	2	1
CVE-2016-2162	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	2	1
CVE-2018-8017	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	5.5	1	2	1
CVE-2014-4172	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	9.8	2	2	1
CVE-2019-3775	CWE-287	Improper Authentication	6.5	1	1	1
CVE-2018-1002200	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	5.5	1	1	1
CVE-2017-1000487	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	9.8	3	17	12
CVE-2018-20227	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	7.5	1	5	1
CVE-2013-5960	CWE-310	Cryptographic Issues	5.8	1	2	15
CVE-2018-1000854	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	9.8	1	2	1
CVE-2016-3720	NA	NA	9.8	1	1	1
CVE-2016-7051	CWE-611	Improper Restriction of XML External Entity Reference	8.6	1	1	1
CVE-2018-1000531	CWE-20	Improper Input Validation	7.5	1	1	1
CVE-2018-1000125	CWE-20	Improper Input Validation	9.8	1	4	1
APACHE-COMMONS-001	NA	NA	NA	1	1	1
CVE-2013-4378	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	4.3	1	1	1
CVE-2018-1000865	CWE-269	Improper Privilege Management	8.8	1	3	1
CVE-2018-1000089	CWE-532	Insertion of Sensitive Information into Log File	7.4	1	2	1
CVE-2015-6748	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	1	1
CVE-2016-10006	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	1	1
CVE-2018-1000615	NA	NA	7.5	1	1	1
CVE-2017-8046	CWE-20	Improper Input Validation	9.8	2	5	1
CVE-2018-11771	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	5.5	1	1	2
CVE-2018-15756	NA	NA	7.5	1	5	2
CVE-2018-1000850	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	7.5	1	2	3
CVE-2017-1000207	CWE-502	Deserialization of Untrusted Data	8.8	1	3	1
CVE-2019-10173	CWE-502	Deserialization of Untrusted Data	9.8	1	7	1
CVE-2019-12402	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	7.5	1	1	1
CVE-2020-1953	NA	NA	10.0	1	7	2

6.3 Approach

The main objective of *Mystique* is to predict whether a mutant is likely to be vulnerability-mimicking. In order for our approach to be lightweight in terms of

engineering and computational effort, we want *Mystique* to be able to (a) learn relevant features of *Vulnerability-mimicking Mutants* without requiring manual feature definition, and (b) to do so without costly dynamic analysis of mutant executions. To achieve this, we divide our task into two parts: learning a representation of mutants using code embedding technique, and learning to predict based on such embeddings whether or not the represented mutants are *Vulnerability-mimicking Mutants*.

6.3.1 Overview of *Mystique*

Figure 6.3 shows an overview of *Mystique*. We divide our approach into three steps that we detail later in this section:

1. *Building a token representation*: *Mystique* pre-processes the original code in order to remove irrelevant information and to produce abstracted code, which is then tokenized to form a sequence of tokens. Each mutant is eventually transformed into its corresponding token representation and undergoes the next step.

2. *Representation learning*: We train an encoder-decoder model to generate an embedding, a.k.a. vector representation of the mutant. This step is where *Mystique* automatically learns the relevant features of mutants without requiring an explicit definition of these features.

3. *Classification: Mystique* trains a classification model to classify the mutants (based on their embeddings) as *Vulnerability-mimicking Mutants* or not. The true labels used for training the model are obtained by i) replacing the fixed code file with a mutated code file in the project, ii) executing the test suite, iii) checking whether or not the tests failed, and iv) if yes, then matching whether the failed tests are the same as the vulnerability's failed tests.

It is interesting to note that the mutant representation learned by *Mystique* does not depend on a particular vulnerability. *Mystique* rather aims to learn properties of the mutants (and their surrounding context) that are generally vulnerability mimicking. This is in line with the recent work on contextual mutant selection [GOD+22; CPB+20] that aims at selecting high-utility mutants for mutation testing. This characteristic makes *Mystique* applicable to pieces of code that it has not seen during training. Our results also confirm the capability of *Mystique* to be effective on projects not seen during training. Certainly, to make our classifier effective in practice, the selection of the mutant generation technique is important. We use *μBERT* since it produces a sufficiently large set of useful mutants by masking and replacing tokens of the class under analysis. Also, since it employs a pre-trained language model, it proposes code (mutants) similar to the one written by programmers.

6.3.2 Token Representation

A major challenge in learning from raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code [TWB+19b; TPW+19; ABL+19; GDJ+22]. In our case, this large vocabulary may hinder *Mystique*'s ability to learn relevant features of *Vulnerability-mimicking Mutants*. Thus, we first abstract original (non-mutated) source code by replacing user-defined entities (function names, variable names, and string literals) with generic identifiers that can be reused across the source code file. During this step, we also remove code comments. This pre-processing yields an abstracted version of the original source code, as the abstracted code snippet in Figure 6.3.

To perform the abstraction, we use the publicly available tool *src2abs* [TPW+19]. This tool first discerns the type of each identifier and literal in the source code. Then, it replaces each identifier and literal in the stream of tokens with a unique ID representing the type and role of the identifier/literal in the code. Each ID <TYPE>_# is formed by a prefix, (i.e., <TYPE>_) which represents the type and role of the identifier/literal, and a numerical ID, (i.e., #) which is assigned sequentially while traversing through the code. These IDs are reused when the same identifier/literal appears again in the stream of tokens. Although we use src2abs, any utility that identifies user-defined entities and replaces such with reusable identifiers can be used as an alternative.

Next, to represent a mutant, we annotate the abstracted code with a mutation annotation on the statement next to the operand/operator that has been mutated. These annotations indicate the applied mutation operation, e.g., *BinaryOperator-Mutator* represents mutation on the binary operator ">=", as shown in figure 6.3. We repeat the process for every mutant.

Finally, we flatten every mutant (by removing newline, extra white space, and tab characters) to create a single-space-separated sequence of tokens. Using these sequences, we intend to capture as much code as possible around the mutant without incurring an exponential increase in training time [TWB+19a; TPW+19; GOD+22; GDJ+22; GOD+22]. We found a sequence length of 150 tokens to be a good fit for our task as it does not exceed 18 hours of training time (wall clock) on a Tesla V100 GPU.

6.3.3 Embedding Learning with Encoder-Decoder

Our next step is to learn the embedding, a.k.a. vector representation (that is later used to train a classification model) from mutants' token representation. We develop an encoder-decoder model, a neural architecture commonly used in representation learning task [KB13]. The key principle of our encoder-decoder architecture is that the encoder transforms the token representation into an embedding and the decoder attempts to retrieve the original token representation from

the encoded embedding. The learning objective is then to minimize the binary cross-entropy between the original token representation and the decoded one. Once the model training has converged, we can compute the embedding from any other mutant's token representation by feeding the latter into the encoder and retrieving the output embedding.

We use a bi-directional Recurrent Neural Network (RNNs) [BGL+17] to develop our encoder-decoder, as previous works on code learning have demonstrated the effectiveness of these models to learn useful representations from code sequences [BCB14; GOD+22; GDJ+22; SVL14]. We build *Mystique* on top of KerasNLP [WQB+22] which is a natural language processing library providing a general purpose *Transformer Encoder-Decoder* architecture following the work of Vaswani et. al [VSP+17] which has shown to perform good both in software engineering and other learning tasks [SNL19; GDJ+22].

To determine an appropriate number of training epochs for model convergence, we conducted a preliminary study involving a small validation set (independent of both the training and test sets used in our evaluation) where we monitor the model's performance in replicating (as output) the same mutant sequence provided as input. We pursue training the model till the training performance on the validation set does not improve anymore. We found 10 epochs for the sequences up to a length of 150 tokens to be a good default for our validation sets.

6.3.4 Classifying Vulnerability-mimicking mutants

Next, we train a classification model to predict whether a mutant, which is represented by the embedding produced by the Encoder, is likely to be *Vulnerability-mimicking Mutants*. The learning objective here is to maximize the classification performance, which we mainly measure with Matthews Correlation Coefficient (MCC), Precision, and Recall, as detailed in section 2.2.4 in chapter 2. To obtain our true classification labels, we replace the fixed code file with a mutated code file in the project, execute the test suite, and check whether or not the tests failed. If the tests fail, we match if the failed tests are the same as the vulnerability's failed tests to determine whether or not the mutant is a vulnerability-mimicking mutant. For developing the classification model, we rely on *random forests* [Bre01] because these are lightweight to train and have shown to be effective in solving various software engineering tasks [JRP+19; PMD+20]. We used standard parameters for random forests, viz. we set the number of trees to 100, use Gini impurity for splitting, and set the number of features (i.e. embedding logits) to consider at each split to the square root of the total number of features.

Once the model training has converged, we use the random forest to predict whether a mutant (in the testing set) is likely to be *Vulnerability-mimicking Mutants*. We make the mutant go through the preprocessing pipeline to obtain its abstract token representation, then feed this representation into the trained encoder-decoder

model to retrieve its embeddings, and input this embedding into the classifier to obtain the predicted label (vulnerability-mimicking or not).

6.4 Research Questions

We start our analysis by investigating how many vulnerabilities in our dataset can be behaviourally mimicked by one or more mutants, i.e., how many mutants fail the same PoVs (tests that were failed by the respective vulnerabilities). Therefore we ask:

RQ1 *Empirical observation I:* How many vulnerabilities can be mimicked by the mutants?

For this task, we rely on *Vul4J* dataset [MdA22] (section 3.3.1) for obtaining vulnerable projects with vulnerabilities, corresponding fixes, and PoV tests, and on *μBERT* [DP22] (section 3.3.2) for generating mutants. In the *Vul4J* dataset, the fixes (for the vulnerabilities) passed the corresponding project's test suite (containing the PoV tests) in 45 cases for which we mention the details in Table 6.1. *μBERT* produces mutants of the fixed code, which are checked for mimicking the corresponding vulnerability by replacing the fixed code file with the mutant and executing the test suite. Apart from checking how many vulnerabilities can be mimicked by the mutants, we also analyze how semantically similar the generated mutants are with the vulnerabilities. We measure the semantic similarity of a mutant with the vulnerability by calculating the Ochiai coefficient [OCH57] as explained in the following section 6.5.1. Hence, we ask:

RQ2 *Empirical observation II:* How similar are the generated mutants with vulnerabilities?

Next, we analyze if the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features. We do so by training models as explained in section 6.3 and check the performance of *Mystique* in predicting *Vulnerability-mimicking Mutants*. Hence, we ask:

RQ3 *Prediction Performance:* How effective is *Mystique* in automatically defining and learning the features associated with *Vulnerability-mimicking Mutants*?

6.5 Experimental Setup

6.5.1 Semantic similarity

Mutation seeds artificial faults, a.k.a. mutants, by performing slight syntactic modifications to the program under analysis. For instance, in Figure 6.3, the

expression `x >= y` can be mutated to `x < y`. Semantic similarity is usually used to evaluate fault seeding [JH09; PSY+18; PHH+16], i.e. how similar is a mutant (seeded artificial fault) to the desired (real) fault. In the case of this study, the desired fault is the corresponding vulnerability.

To compute the semantic similarity we resort to dynamic test executions. We use a similarity coefficient, i.e., Ochiai coefficient [OCH57], to compute the similarity of the passing and failing test cases. This is a common practice in many different lines of work, such as mutation testing [JH09; PSY+18], program repair [GNF+12], and code analysis [GBH+17] studies. Since semantic similarity compares the behavior between two program versions using a reference test suite, the Ochiai coefficient approximates program semantics using passing and failing test cases.

The Ochiai coefficient represents the ratio between the set of tests that fail in both versions over the total number of tests that fail in the sum of the two. For instance, let P_1, P_2, fTS_1 and fTS_2 be two programs and their respective set of failing tests, then the Ochiai coefficient between programs P_1 and P_2 is computed as:

$$Ochiai(P_1, P_2) = \frac{|fTS_1 \cap fTS_2|}{\sqrt{|fTS_1| \times |fTS_2|}}$$

The Ochiai coefficient ranges from 0 to 1, with 0 in case of none of the failed tests is the same between both versions of the programs, (i.e., a mutant and the vulnerability that it is trying to mimic), and 1 in case of all the failed tests match between both versions. Intuitively, a mutant M *mimics* vulnerability V, if and only if its semantic similarity is equal to 1, i.e., $Ochiai(V, M) = 1$. The mutants shown in Figures 6.1 and 6.2 have an Ochiai coefficient equal to 1 with their corresponding vulnerability.

6.5.2 Experimental Procedure

To answer our RQs, we first execute the test suite for every mutant produced by *μBERT* and analyze which mutants fail the same tests that were failed by the vulnerability to determine *Vulnerability-mimicking Mutants*. In total, *μBERT* produces *16,409* mutants for the fixed versions of the 45 projects (for which the 45 corresponding vulnerabilities are mentioned in Table 6.1). We repeated the test suite execution process for every project to label *Vulnerability-mimicking Mutants* that mimic the corresponding vulnerability.

Once the labeling is complete, to answer *RQ1*, we perform an exact match of the mutant's failed tests with the vulnerability's failed tests to determine how many vulnerabilities are mimicked by the generated mutants. To answer *RQ2*, we rely on the Ochiai similarity coefficient (elaborated in Section 6.5.1) to measure how similar the generated mutants are with the vulnerabilities. We calculate the

Ochiai coefficient to compute the similarity of the passing and the failing test cases of every vulnerability with all the corresponding project's mutants. To answer *RQ3*, we train models on *Vulnerability-mimicking Mutants* and perform k-fold cross-validation (where k = 5) at project level (our dataset has only 1 vulnerability per project) where each fold contains 9 projects. So, we train on mutants of 36 projects (4 training folds) and test on mutants of the remaining 9 projects (1 test fold). Once we get the predictions for all 45 subjects, we compute the Prediction Performance Metrics, *i.e.,Precision*, *Recall*, and *Matthews Correlation Coefficient (MCC)* for *Mystique* in order to show its learning ability. Given a mutant is vulnerability-mimicking if it is predicted as vulnerability-mimicking, then it is a true positive (TP); otherwise, it is a false negative (FN). Vice-versa, if a mutant does not mimic the vulnerability and, if it is predicted as vulnerability-mimicking then it is a false positive (FP); otherwise, it is a true negative (TN). Here, MCC is more reliable than others as the classes are of very different sizes, *i.e.,*we have *3.9% Vulnerability-mimicking Mutants* (Positives) in total, for 45 vulnerabilities in our dataset (as shown in Table 6.2).

6.6 Experimental Results

6.6.1 Empirical observation I (RQ1)

µBERT generates *16,409* mutants in total, for all projects in our dataset. Out of *16,409* mutants, *646* mutants are *Vulnerability-mimicking Mutants* mimicking *25* out of *45* vulnerabilities, i.e., at least one or more mutants behave the same as 25 vulnerabilities. Overall, *3.9%* of the generated mutants mimicked *55.6%* of the vulnerabilities in our dataset. Table 6.2 shows the project-wise distribution of *Vulnerability-mimicking Mutants* including the total number of mutants generated and the number (and percentage) of mutants that mimic the vulnerabilities. These results are encouraging and evidence the potential value of using *Vulnerability-mimicking Mutants* as test requirements in practice for security-conscious testing, leading to test suites that can tackle similar mimicked vulnerabilities.

Answer to RQ1: *µBERT*-generated 646 out of 16,409 mutants mimicked 25 out of 45 vulnerabilities, i.e., *3.9%* of the generated mutants mimicked *55.6%* of the vulnerabilities. This evidence that pre-trained language models can produce test requirements (mutants) that behave the same as vulnerabilities, making security-conscious mutation testing feasible.

6.6.2 Empirical observation II (RQ2)

In addition to 646 mutants mimicking 25 vulnerabilities, i.e., 646 mutants failed by exactly the same tests as the respective 25 vulnerabilities of the corresponding projects, *2,720* mutants achieved the Ochiai similarity coefficient greater than 0 with 40 vulnerabilities of the corresponding projects. This shows that 2,720 mutants, i.e., *16.6%* of the mutants fail one or more tests (of the corresponding projects) that were failed by 40 respective vulnerabilities, i.e., *88.9%* of the vulnerabilities in our dataset. Figure 6.4 provides an overview of the mutant-vulnerability similarity in terms of Ochiai similarity coefficient distribution across all the vulnerabilities in our dataset when compared for similarity with the generated mutants.

Despite not behaving exactly the same as the vulnerability, there are many mutants that share some vulnerable behaviors which can help testers to identify the cause of the vulnerability. Moreover, vulnerability-similar mutants can help to design more thorough and complete suites to tackle vulnerabilities.

Answer to RQ2: *μBERT*-generated 2,720 out of 16,409 mutants achieved an Ochiai similarity coefficient greater than 0 with 40 out of 45 vulnerabilities, i.e., *16.6%* of the generated mutants fail one or more tests (of the corresponding projects) that were failed by 88.9% of the respective vulnerabilities.

6.6.3 Prediction Performance (RQ3)

Despite the class imbalance, *Mystique* effectively predicts *Vulnerability-mimicking Mutants* with a prediction performance of 0.63 MCC, 0.80 Precision, and 0.51 Recall outperforming a random selection of *Vulnerability-mimicking Mutants* (i.e., MCC equals 0). These scores indicate that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features. Indeed, any improvement in the mutation testing tools or the pre-trained language models that allow producing better *Vulnerability-mimicking Mutants*, can leverage *Mystique* to select a more complete set of security-related test requirements.

Answer to RQ3: *Mystique* achieves a prediction performance of 0.63 MCC, 0.80 Precision, and 0.51 Recall in predicting *Vulnerability-mimicking Mutants*. This indicates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically prioritize these prior to any analysis or execution.

6.7 Threats to Validity

External Validity: Threats may relate to the vulnerabilities we considered in our study. Although our evaluation expands to vulnerabilities of severity ranging from high to low, spanning from single method fix to multiple methods modified during the fix (as shown in Table 6.1, the results may not generalize to other vulnerabilities. We consider this threat of low importance since we verify all the vulnerabilities and also their fixes by executing tests provided in the Vul4J dataset [BSF22]. Moreover, our predictions are based on the local mutant context, which has been shown to be a determinant of mutants' utility [GOD+22; CPB+20]. Other threats may relate to the mutant generation tool, i.e., *µBERT* that we used. This choice was made since *µBERT* relies on CodeBERT to produce mutations that look natural and are effective for mutation tesing. We consider this threat of low importance since one can use a better mutant generation tool that can produce more *Vulnerability-mimicking Mutants*, which will help *Mystique* in achieving better prediction performance. Nevertheless, in case other techniques produce different predictions, one could re-train, tune and use *Mystique* for the specific method of interest, as we did here with *µBERT* mutants.

Internal Validity: Threats may relate to the restriction that we impose on sequence length, i.e., a maximum of *150* tokens. This was done to enable reasonable model training time, approximately *18* hours to learn mutant embeddings on Tesla V100 gpu. Other threats may be due to the use of *Transformer Encoder-Decoder* following the work of Vaswani et. al [VSP+17] for learning mutant embeddings. This choice was made for simplicity to use the related framework out of the box similar to the related studies [SNL19; GDJ+22]. Other internal validity threats could be related to the test suites we used and the mutants considered as vulnerability mimicking. We used well-tested projects provided by the Vul4J dataset [BSF22]. To be more accurate, our underlying assumption is that the extensive pool of tests including the Proof-of-Vulnerability (PoV) available in our experiments is a valid approximation of the program's test executions, especially the proof of a vulnerability and its verified fix.

Construct Validity: Threats may relate to our metric to measure the semantic similarity of a mutant and a vulnerability, i.e., the Ochiai coefficient. We relied on the Ochiai coefficient because it is widely known in the fault-seeding community as a representative metric to capture the semantic similarity between a seeded and real fault. In the context of this study, the seeded fault is a mutant and the real fault is a vulnerability. We consider this treat of low importance as the Ochiai coefficient takes into consideration the failed tests of a mutant and a vulnerability (as explained in section 6.5.1) representing the observable behavior and serving its purpose for this study.

6.8 Data Availability

The dataset consisting of the source code of all projects (both, vulnerable and fixed), individual classes modified during the fix, i.e., vulnerable and fixed (where fixed classes were used for mutation), generated mutants, separated vulnerability fixes with sentence-level information of the fix, along with *Mystique*'s source code and the tools used in our study, are publicly available in our GitHub repository[2].

6.9 Conclusion

In this chapter, we showed that language model based mutation testing tools can produce *Vulnerability-mimicking Mutants*, i.e., mutants that mimic the observable behavior of vulnerabilities. Since these mutants are a few, i.e., 3.9% of the entire mutant set, there is a need for a static approach to identify such mutants. To achieve this, we presented *Mystique*, a method that learns to select *Vulnerability-mimicking Mutants* from a given mutant's code context. Our experiments show that *Mystique* identified *Vulnerability-mimicking Mutants* with 0.63 MCC, 0.80 Precision, and 0.51 Recall, which indicates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features.

[2]https://github.com/garghub/mystique

Figure 6.1: Vulnerability CVE-2018-17201 (Fig. 6.1a) that allows "Infinite Loop" making code hang, which further enables Denial-of-Service (DoS) attack is fixed with the conditional exception using "`if`" expression (Fig. 6.1b). Vulnerability-mimicking Mutant (Fig. 6.1c) modifies the "`if`" condition that nullifies the fix and re-introduces the vulnerability.

(a) Vulnerable Code (CVE-2018-17201)

```
1  private static void decompress (final InputStream in, final byte[]
       out) throws IOException {
2    int position = 0; final int total = out.length;
3    while (position < total) { final int n = in.read();

4     if (n > 128) { final int value = in.read();
5      for (int i = 0; i < (n & 0x7f); i++) {
6       out[position++] = (byte) value; }
7     } else { for (int i = 0; i < n; i++) {
8       out[position++] = (byte) in.read(); } } } }
```

(b) Fixed Code

```
1  private static void decompress (final InputStream in, final byte[]
       out) throws IOException {
2    int position = 0; final int total = out.length;
3    while (position < total) { final int n = in.read();(
4     if (n<0) { throw new ImageReadException
5       ("Error decompressing RGBE file"); }
6     if (n > 128) { final int value = in.read();
7      for (int i = 0; i < (n & 0x7f); i++) {
8       out[position++] = (byte) value; }
9     } else { for (int i = 0; i < n; i++) {
10      out[position++] = (byte) in.read(); } } } }
```

(c) Vulnerability-mimicking Mutant

```
1  private static void decompress (final InputStream in, final byte[]
       out) throws IOException {
2    int position = 0; final int total = out.length;
3    while (position < total) { final int n = in.read();(
4     if (n==0) { throw new ImageReadException // '<' modified to '=='
5       ("Error decompressing RGBE file"); }
6     if (n > 128) { final int value = in.read();
7      for (int i = 0; i < (n & 0x7f); i++) {
8       out[position++] = (byte) value; }
9     } else { for (int i = 0; i < n; i++) {
10      out[position++] = (byte) in.read(); } } } }
```

Figure 6.2: Vulnerability CVE-2018-1000850 that allows "Path Traversal", which further enables access to a Restricted Directory (Fig. 6.2a) is fixed with the conditional exception in case '`.`' or '`..`' appears in the "`newRelativeUrl`" (Fig. 6.2b). Vulnerability-mimicking Mutant (Fig. 6.2c) in which the passed argument is changed from "`newRelativeUrl`" to "`name`" nullifies the fix and re-introduces the vulnerability.

(a) Vulnerable Code (CVE-2018-1000850)

```
1  void addPathParam(String name, String value, boolean encoded) {
2   if (relativeUrl == null) { throw new AssertionError(); }
3   relativeUrl = relativeUrl.replace("{" + name + "}" ,
4                  canonicalizeForPath(value, encoded));
5  }
```

(b) Fixed Code

```
1  void addPathParam(String name, String value, boolean encoded) {
2   if (relativeUrl == null) { throw new AssertionError(); }
3   String replacement = canonicalizeForPath(value, encoded);
4   String newRelativeUrl =
5     relativeUrl.replace("{" + name + "}", replacement);
6   if (PATH_TRAVERSAL.matcher(newRelativeUrl)
7       .matches()) { throw new IllegalArgumentException(
8         "@Path parameters shouldn't perform path traversal
9         ('.' or '..'): " + value ); }
10  relativeUrl = newRelativeUrl;
11 }
```

(c) Vulnerability-mimicking Mutant

```
1  void addPathParam(String name, String value, boolean encoded) {
2   if (relativeUrl == null) { throw new AssertionError(); }
3   String replacement = canonicalizeForPath(value, encoded);
4   String newRelativeUrl =
5     relativeUrl.replace("{" + name + "}", replacement);
6   if (PATH_TRAVERSAL.matcher(name)// <- passed argument changed here
7       .matches()) { throw new IllegalArgumentException(
8         "@Path parameters shouldn't perform path traversal
9         ('.' or '..'): " + value ); }
10  relativeUrl = newRelativeUrl;
11 }
```

Figure 6.3: Overview of *Mystique*: Source code is abstracted and annotated to represent a mutant which is further flattened to create a single-space-separated sequence of tokens. An encoder-decoder model is trained on sequences to generate mutant embeddings. A classifier is trained on these embeddings and their corresponding labels (whether or not the mutants are *Vulnerability-mimicking Mutants*). The trained classifier is then used for label prediction of test set mutants.

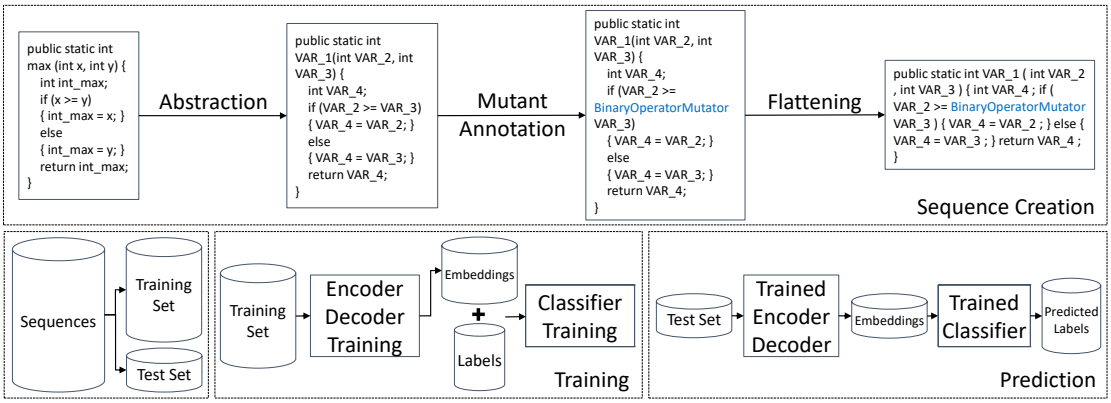


Figure 6.4: RQ2: Distribution of the mutant-vulnerability similarity in terms of Ochiai similarity coefficient across all the vulnerabilities when compared for similarity with the generated mutants. Overall, 16.6% of the mutants fail one or more tests that were failed by 88.9% of the respective vulnerabilities.

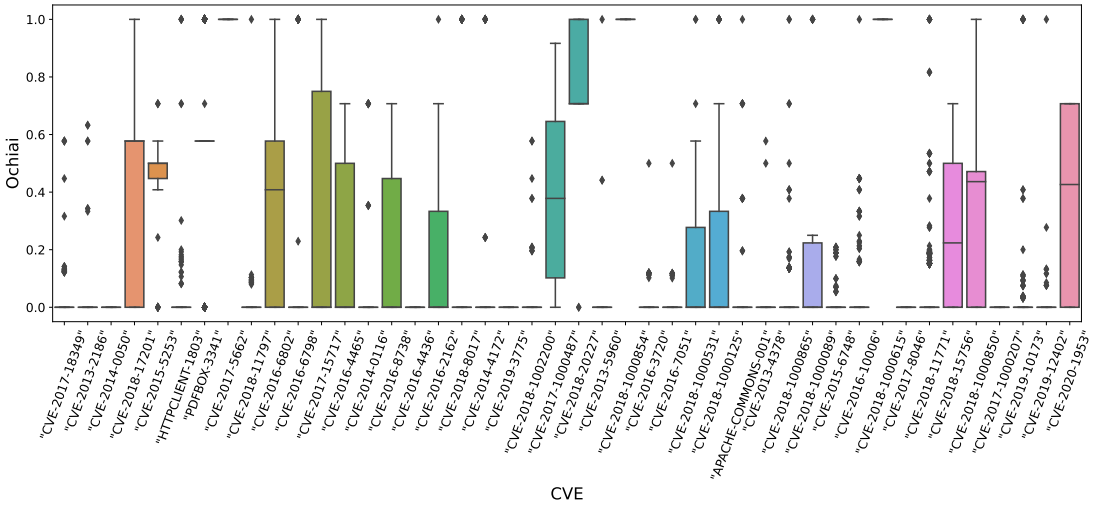


Table 6.2: RQ1: The table records vulnerability-mimicking mutant distribution details that include the number of generate mutants across all projects with vulnerabilities, and the number and percentage of *Vulnerability-mimicking Mutants* among them. Overall, 3.9% of the generated mutants mimic 55.6% of the vulnerabilities.

CVE (Vulnerability)	**# Total mutants**	**Vulnerability-mimicking mutants**	
		(#)	(%)
CVE-2017-18349	286	0	0%
CVE-2013-2186	191	0	0%
CVE-2014-0050	456	0	0%
CVE-2018-17201	375	8	2.13%
CVE-2015-5253	257	0	0%
HTTPCLIENT-1803	553	5	0.9%
PDFBOX-3341	2169	308	14.2%
CVE-2017-5662	511	86	16.83%
CVE-2018-11797	266	1	0.38%
CVE-2016-6802	338	16	4.73%
CVE-2016-6798	441	19	4.31%
CVE-2017-15717	437	77	17.62%
CVE-2016-4465	48	0	0%
CVE-2014-0116	167	0	0%
CVE-2016-8738	50	0	0%
CVE-2016-4436	74	0	0%
CVE-2016-2162	169	1	0.59%
CVE-2018-8017	738	17	2.3%
CVE-2014-4172	212	12	5.66%
CVE-2019-3775	9	0	0%
CVE-2018-1002200	177	0	0%
CVE-2017-1000487	586	0	0%
CVE-2018-20227	18	3	16.67%
CVE-2013-5960	112	1	0.89%
CVE-2018-1000854	9	2	22.22%
CVE-2016-3720	387	0	0%
CVE-2016-7051	387	0	0%
CVE-2018-1000531	158	2	1.27%
CVE-2018-1000125	155	14	9.03%
APACHE-COMMONS-001	144	1	0.69%
CVE-2013-4378	189	0	0%
CVE-2018-1000865	432	2	0.46%
CVE-2018-1000089	205	7	3.41%
CVE-2015-6748	989	0	0%
CVE-2016-10006	356	1	0.28%
CVE-2018-1000615	67	38	56.72%
CVE-2017-8046	12	0	0%
CVE-2018-11771	1754	12	0.68%
CVE-2018-15756	274	0	0%
CVE-2018-1000850	307	2	0.65%
CVE-2017-1000207	29	0	0%
CVE-2019-10173	1658	10	0.6%
CVE-2019-12402	246	1	0.41%
CVE-2020-1953	11	0	0%

7

Learning from What We Know: How to Perform Vulnerability Prediction using Noisy Historical Data

Vulnerability prediction refers to the problem of identifying system components that are most likely to be vulnerable. Typically, this problem is tackled by training binary classifiers on historical data. Unfortunately, recent research has shown that such approaches underperform due to the following two reasons: a) the imbalanced nature of the problem, and b) the inherently noisy historical data, i.e., most vulnerabilities are discovered much later than they are introduced. This misleads classifiers as they learn to recognize actual vulnerable components as non-vulnerable. To tackle these issues, we propose TROVON*, a technique that learns from known vulnerable components rather than from vulnerable and non-vulnerable components, as typically performed. We perform this by contrasting the known vulnerable, and their respective fixed components. This way,* TROVON *manages to learn from the things we know, i.e.,vulnerabilities, hence reducing the effects of noisy and unbalanced data. We evaluate* TROVON *by comparing it with existing techniques on three security-critical open source systems, i.e.,Linux Kernel, OpenSSL, and Wireshark, with historical vulnerabilities that have been reported in the National Vulnerability Database (NVD). Our evaluation demonstrates that the prediction capability of* TROVON *significantly outperforms existing vulnerability prediction techniques such as* Software Metrics*,* Imports*,* Function Calls*,* Text Mining*,* Devign*,* LSTM*, and* LSTM-RF *with an improvement of 40.84% in* Matthews Correlation Coefficient *(MCC) score under* Clean Training Data Settings*, and an improvement of 35.52% under* Realistic Training Data Settings*.*

Contents

7.1 Introduction

A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders, *i.e.,*the application owner, application users, and other entities that rely on the application [20b]. While vulnerabilities can be thought of as specific types of software defects (or bugs), there are subtle and significant differences that make their identification considerably more complex and challenging than the problem of finding bugs [TZY+15; PM04].

Vulnerabilities are fewer in comparison to defects, limiting the information one can learn from. Also, their identification requires an attacker's mindset [MHM+15], which developers or code reviewers may not possess. Lastly, the continuous growth of codebases makes it difficult to investigate them entirely and track all code changes. For example, the Linux kernel, one of the projects with the highest number of publicly reported vulnerabilities, reached 27.80 million LoC (Lines of Codes) at the beginning of 2020 [20a].

Vulnerability prediction approaches were proposed to tackle these challenges by prioritizing the efforts that developers and code reviewers have to put on when testing or reviewing code to find vulnerabilities. These methods take advantage of the large amounts of historical data available based on which they learn a set of features and/or code properties that associate with vulnerabilities. For instance, the presence of vulnerabilities has been linked to high code churn [SMW+11], to the use of specific library imports and function calls [NZH+07], and the frequency of suspicious code tokens [TZY+15]. Unfortunately, building models around such features is challenging due to the small number of available vulnerable code instances, which limit the learning ability of the predictors [ZNG+09].

Furthermore, *Jimenez et al.* [JRP+19] demonstrated that vulnerability prediction approaches have been built under a *"clean"* training data assumption, *i.e.,*all the component's labeling information (vulnerable / non-vulnerable) is always available irrespective of time. Their study showed that under these settings the approaches do not account for the gradual revelation of vulnerabilities over time. This results in prediction models training on even those vulnerabilities that have not been uncovered yet, *e.g.* all vulnerabilities known from time t onwards are available at all times, even before time t.

Jimenez *et al.* advocated *Realistic Training Data Settings* where the vulnerability labels used for training the prediction models are more realistically available at training time. For example, in such settings, at a given time t, only the vulnerabilities known till time t should be available for training. All vulnerabilities known from time t onwards should *not* be available for training beforehand. Their study demonstrated that *Realistic Training Data Settings* results in unavoidable noise in the training data because every component with no reported vulnerability

till training time is considered non-vulnerable during training, which makes existing approaches perform poorly. This establishes a need for robust vulnerability prediction techniques.

We advance in this direction by developing *TROVON*[1]– a method that learns from validated data, i.e., we train only on components known to be vulnerable and leave aside the (supposedly) non-vulnerable ones. This way, we do not make any assumptions on non-vulnerable components and bypass the key problem faced by previous works. To do so, we rely on a simple yet powerful language-agnostic machine translation technique [BGL+17] which we train on pairs of vulnerable and fixed code fragments, available at projects' release time. In particular, we contrast the code fragments pairs (pairs of vulnerable and fixed fragments) that were modified when fixing a vulnerability, with fragment pairs from other functions of the same components (fragments less likely to be vulnerable) in order to learn to distinguish likely vulnerable from non-vulnerable code.

TROVON focuses on vulnerability fixes, i.e., code transformations that turn vulnerable code into a non-vulnerable one, to train the machine translation model that aims at capturing silent features related to the differences between vulnerable and fixed components. Therefore, predictions are guided by actual points of interest, (*i.e.,*diff points) in the vulnerable code where the transformations should happen. This means that *TROVON* learns to identify code characteristics that are similar to those (vulnerable) seen during training.

We empirically assess the effectiveness of *TROVON* on available releases of three security-critical open source systems, *i.e.,*Linux Kernel, Wireshark, and OpenSSL. Our evaluation demonstrates that *TROVON* significantly outperforms existing vulnerability prediction approaches under both *Clean Training Data Settings* and *Realistic Training Data Settings*.

In particular, our results show that when we train all the approaches (including *TROVON*) with clean training data, *TROVON* outperforms the existing approaches by 83.96% in Precision, 155.33% in Recall, 132.95% in F-measure, and 80.39% in Matthews Correlation Coefficient (MCC). In addition to these metrics, we also evaluate *TROVON* on predicting unseen vulnerable components specifically. This is a new metric that we introduce in this chapter to help evaluate the extent to which vulnerability prediction generalizes, i.e., ability to predict unseen components (components not used for training) as being vulnerable or not. The percentages of unseen vulnerable components predicted by TROVON, on average, are 40.05%, 64.34%, and 42.28% higher than the ones obtained by existing techniques in Linux Kernel, Wireshark, and OpenSSL releases, reflecting *TROVON*'s better generalization capability. Under *Realistic Training Data Settings*, on average,

[1] ***TROVON*** is an abbreviation for "**Tr**aining **o**n **v**ulnerabilities **on**ly", which is the core focus of our study.

TROVON achieved 0.39 MCC, (*i.e.,*3.63 times higher than the baselines), 0.69 F-measure, (*i.e.,*11.82 times higher), 0.86 Precision, (*i.e.,*2.66 times higher), and 0.58 Recall, (*i.e.,*15.25 times higher than the baselines).

In summary, we make the following contributions:

1. We present *TROVON*, a novel vulnerability prediction method via machine translation.

2. We demonstrate that *TROVON* significantly outperforms existing methods through a large empirical study.

3. We corroborate that *TROVON* remains robust when trained in *Realistic Training Data Settings* that includes unavoidable noise, where almost all previous methods that we compared with, fail [JRP+19].

7.2 Approach

The key idea of *TROVON* is to train a machine translator (viz. an encoder-decoder sequence to sequence model) to identify vulnerable code, by feeding it with vulnerable code fragments and their corresponding fixes. Machine translators can automatically recognize: (i) features of the language (to be translated) and (ii) required translation (to the desired language). In our case, it is used to automatically identify vulnerability features with minimum overhead.

It should be noted that we do not aim at fixing vulnerable code, but rather at identifying likely vulnerable code instances. The point here is that we use the translator to indicate the presence of vulnerabilities without considering the fixes produced by the model. In other words, we leverage the ability of the translators to learn the vulnerabilities' context and not their instance and location. We assert that since vulnerable code instances are scarce, information gained from historical data is inevitably partial and incomplete. Therefore, it can be used to indicate the presence of vulnerabilities but not their instance context.

The translator is trained on *input - desired output* pairs,*i.e.,*on *vulnerable - fixed* code fragments. For prediction, one can input an unseen code into the trained translator to check whether it is likely to be vulnerable. If the translator changes the code then it can be concluded that the code is likely to be vulnerable. To avoid many false positives (the translator changing every input code fragment), we also train it to leave non-vulnerable code fragments *unchanged*. To this end, we also feed the translator with input-output pairs where each of which is a non-vulnerable code fragment (input = output). It must be noted that we train only on the components (files) that were fixed, leaving aside the unchanged ones. This way we aim at reducing the noise from the training data, *i.e.,*by focusing on what we are certain of; the information provided by the vulnerability fixes.

Figure 7.1: Implementation: Sequences generated from the source-code are used to train the model to generate desired output sequences. The trained model is provided with sequences generated from an unseen source code. The component prediction is based on the generated output sequences.

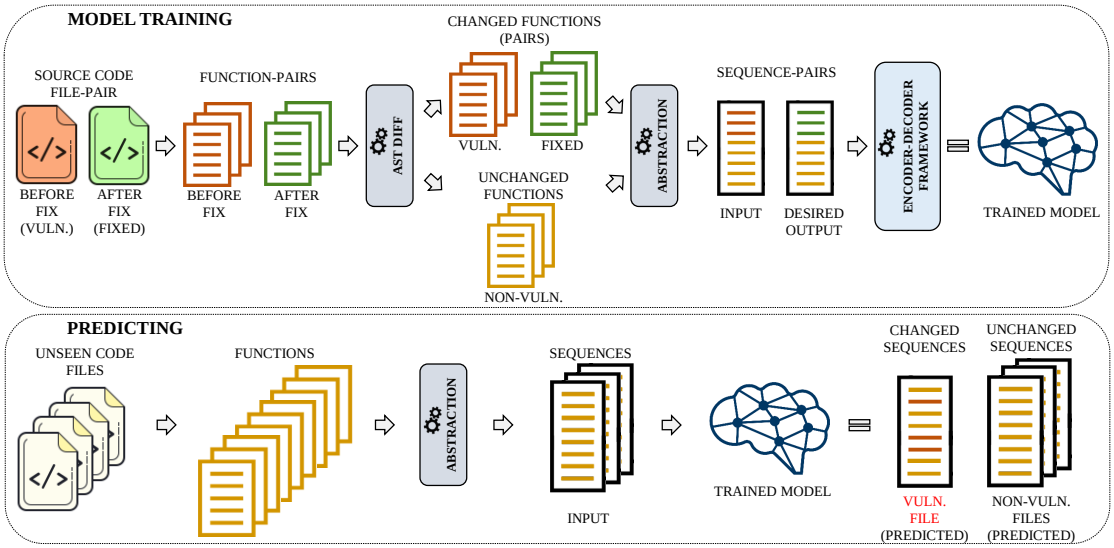


Figure 7.1 shows an overview of the implementation. Starting from vulnerable code components and their fixes, it involves the following activities: 1) decomposing the components into code fragments; 2) identifying which code-fragments are responsible for the vulnerability; 3) producing abstracted code-fragments by removing irrelevant information (e.g. user-defined names, comments); 4) configuring and training the machine translator. 5) producing abstracted code-fragments of an unseen code component and using the trained machine translator to predict whether it is likely to be vulnerable.

7.2.1 Decomposing Components into Code Fragments

We target our predictions at the component, (*i.e.,*file) level due to: a) the empirical evidence provided by Morrison *et al.* [MHM+15] and b) to account for the context of code (vulnerability-fixes) that can be fixed at multiple locations throughout the component. A code-fix can be an addition, removal, and/or modification of code. Since functions are the basic building blocks of a program, we use them to establish function-level mappings between the vulnerable components and their fixed counterparts (based on the function headers).

Thus, we extract all the functions from both, a vulnerable component and its fixed counterpart, and pair each before-fix function with the corresponding after-fix

Figure 7.2: Abstraction: Actual Functions (left) are abstracted by replacing user-defined Function names, Type names, Variable names, and String Literals to F_num, T_num, V_num, and L_num, respectively to achieve Abstracted Functions (right).

(a) Actual Function

```
1  void dev_load (struct net *netw,
2      const char *name ) {
3    struct net_device *dev;
4    rcu_read_lock();
5    dev = dev_get_by_name_rcu (
6      netw , name);
7    rcu_read_unlock();
8    if ( !dev && capable (
9        CAP_NET_ADMIN ) )
10     request_module("%s", name); }
```

(b) Abstracted Function

```
1  void F_1 (struct T_1 *V_1,
2      const char *V_2 ) {
3    struct T_2 *V_3;
4    F_2();
5    V_3 = F_3 (
6      V_1, V_2);
7    F_4();
8    if ( !V_3 && F_5 (
9        V_4 ) )
10     F_6(L_1, V_2); }
```

function. The functions that cannot be paired, i.e., having no counterpart, are discarded. This can happen due to the creation and/or deletion of a function to fix a vulnerability, *e.g.* a function added during the fix which was not present before or vice-versa.

7.2.2 Categorizing Functions as Vulnerable or Non-Vulnerable

As typically performed in this line of work, we consider as vulnerable, any function that was modified to fix the vulnerability. The remaining are considered as non-vulnerable (not vulnerable to the specific vulnerability). When comparing a before-fix copy to its after-fix counterpart, we ignore irrelevant syntactical changes, *e.g.* additional blank spaces and new lines. If there remain syntactical differences between the two copies, we label the before-fix as vulnerable.

7.2.3 Abstracting Irrelevant Information

A major challenge in dealing with raw source code is the huge vocabulary created by the abundance of identifiers and literals used in the code. Vocabulary, on such a large scale, hinders the learning of relevant code patterns [TWB+19a]. Thus, to reduce the vocabulary size, we transform the source code into an abstract representation by replacing user-defined entities with re-usable IDs.

Figure 7.2 shows a code snippet of a real function (Figure 7.2a) converted into its abstract representation (Figure 7.2b). The purpose of this abstraction is to replace any reference to user-defined entities (function name, type name, variable name, and string literal) with IDs that can be reused across functions (thereby reducing vocabulary size). Thus, we replace identifiers and string literals with unique IDs. Additionally, comments and annotations are removed.

New IDs follow the regular expression `(F|T|V|L)_(num)`$^+$, where `num` stands for numbers $0, 1, 2, \ldots$ assigned in a sequential and positional fashion based on the occurrence of that entity. All the entities - user-defined *Function* names, *Type* names, *Variable* names, and *String Literals* are replaced with `F_num`, `T_num`, `V_num`, and `L_num`, respectively. Thus, the first function name receives the ID `F_1`, the second receives the ID `F_2`, and so on. If any of these entities appear multiple times in a function, it is replaced with the same ID.

Each function (pair) is abstracted in isolation to yield an abstracted function code, *i.e.,*same IDs can be reused across functions without impacting *TROVON*. ID references are not preserved across functions, e.g., `V_1` may refer to two different variable names from one function to another. This is the key to reduce the vocabulary size, *e.g.* the name of the first function called in any pair is replaced with the ID `F_1`, regardless of its original name.

In the case of vulnerable functions, the before-fix copy is abstracted first and then the after-fix copy. IDs are shared between the two copies (before-fix and after-fix) of the functions and new IDs are generated only when new (*Function*, *Type*, *Variable*) names and *String Literals* are found.

The abstracted code is rearranged in a single sentence to represent a sequence of space-separated entities, which is the representation supported by the machine translator. Sequences generated from vulnerable (before-fix), fixed (after-fix), and unchanged functions are named vulnerable, fixed, and unchanged sequences, respectively. In these settings, fixed and unchanged sequences represent non-vulnerable cases. To limit the computation cost involved in training the translator, large sequences are split into multiple sequences of no more than *50* tokens each.

7.2.4 Building the Machine Translator

To build our machine translator, we train an encoder-decoder model that can transform an input sequence to the desired sequence (output of the model).

A representation of a sequence is similar to a sentence in a natural language that consists of words separated by spaces and ends with a full stop. Instead of words and full stop character, a sequence has tokens and a newline character. Thus, we train the encoder-decoder by feeding it with pairs of sequences. More precisely, we use two types of pairs: (i) vulnerable sequences with their corresponding fixed sequences, and (ii) non-vulnerable sequences paired with themselves. Non-vulnerable sequence-pairing is essential to allow the learner to identify what should not be changed. Thereby, avoiding to raise many false positives (incorrectly predicting non-vulnerable sequences as vulnerable) while learning only from “clean" data.

7.2.5 Predicting Vulnerable Components

To predict whether an unseen component, (*i.e.,*file) is potentially vulnerable, we decompose it into sequences following the process depicted in Figure 7.1. Then, we feed the resulting sequences into the machine translator which produces output sequences. If one (or more) of the output sequences returned by the model is different from the original one, (*i.e.,*input sequences), we consider the component as likely to be vulnerable. Otherwise, we consider component as likely non-vulnerable, *i.e.,*in case of no change in any of the output sequences in comparison to the input sequences, the component is considered as likely non-vulnerable.

7.3 Experimental Evaluation

7.3.1 Research Questions

TROVON aims to support code reviews by predicting vulnerable components in new releases, based on the information learned from previous (historical) data, i.e., the previous project release. Therefore, our first research question regards the prediction ability of *TROVON*. We measure the prediction ability of *TROVON* to correctly predict vulnerable and non-vulnerable components. We do so with the help of classification assessment metrics, i.e., Precision, Recall, F-1, and MCC. We evaluate this by training on all available vulnerabilities of one release and testing on the next release, for all available release pairs. Thus, we ask:

RQ1 What is the prediction performance of *TROVON* in a release-based scenario ?

After assessing the prediction ability of *TROVON*, we turn our attention to existing techniques. Hence, we investigate:

RQ2 What is the prediction performance of *TROVON* in comparison to existing techniques?

In *TROVON*, we train a model on the vulnerabilities of a release and test the trained model on the components of the next release. Since we perform a release-based evaluation, vulnerabilities spanning across multiple releases could be either seen by the trained model (used during training) or not (newly appearing component). Thus, we may have the knowledge in advance that a component is vulnerable in a given release irrespective of the vulnerability detection date. As these vulnerable components may remain unfixed and reappear in the next release, it is essential to assess the learning potential of our models by evaluating how proficient are the studied models in classifying correctly components that were "seen" during training, in a sense checking how well the model remembers, and in classifying new components, i.e., components that were "unseen" during training, in a sense checking how well a model can actually perform on new instances. Hence, we aim at controlling for seen and unseen vulnerable components and ask:

RQ3 What is the prediction performance of the studied techniques in predicting

seen and unseen vulnerable components?

Until now, we consider that in every release all known vulnerable components are labelled as such, *i.e.,*following the *clean* training data settings. This analysis provides indications on what the potential prediction ability of the approaches is when the available data are clean, *i.e.,*all the component's labeling information (vulnerable / non-vulnerable) is always available irrespective of time. Unfortunately, in practice, such information is unavailable and inflates the actual performance of the prediction models. The actual performance in *Realistic Training Data Settings* is much lower due to real-world labeling issues [JRP+19], *i.e.,*vulnerabilities are frequently reported at a much later time than they are actually introduced. This has adverse effects as they cause the classifiers to treat vulnerable components as non-vulnerable. Hence, it is imperative to study performance under *Realistic Training Data Settings*, where a prediction model is trained only on those vulnerabilities that were detected till the release date of a version for which the vulnerability prediction is performed. For this reason, we also evaluate the approaches under *Realistic Training Data Settings*. Hence, we ask:

RQ4 How effective (in predicting vulnerable components) is *TROVON* in comparison to existing techniques under *Realistic Training Data Settings*?

7.3.2 Data

For our study, we need projects with many releases and vulnerabilities. We consider three large security-intensive open-source systems that were used by previous research [JRP+19] – the Linux Kernel, the OpenSSL library, and the Wireshark tool. These systems are widely used, mature, and have a long history of releases and vulnerability reports.

Linux Kernel [91] is an operating system, integrated into billions of systems and devices, such as Android. Linux is one of the largest open-source code-bases and has a long history (since 1991), recorded in its repository. It is relevant for our evaluation since it has many security aspects and is among the projects with a higher number of reported vulnerabilities in NVD. *OpenSSL* [98a] is a library implementing the SSL and TLS protocols, commonly used in communications. It is of critical importance as highlighted by the *Heartbleed* vulnerability, which made half of a million web servers vulnerable to attacks [14]. *Wireshark* [98b] is a network packet analyzer mainly used for troubleshooting and debugging. The project is open source and is relevant for the study because it is integrated with most operating systems.

We use *VulData7* [JPT18] which is a publicly available[2] tool to gather the vulnerabilities, *i.e.,*the vulnerable and the corresponding fixed components of the aforementioned systems. As we mention in section 2.3.3, for every vulnerability,

[2] `https://github.com/electricalwind/data7`

Table 7.1: The table records the total number of releases, average number of components, average number of vulnerable components, and the ratio of vulnerable components for the systems we study.

System	**#Releases**	**#Avg.Comp**	**#Avg.Vuln.Comp**	**%Vuln.**
Linux Kernel	36	16456	456	3%
Wireshark	10	2012	134	7%
OpenSSL	10	664	59	9%

NVD provides a Git commit IDs of the code related to vulnerability-fix commit. By using these NVD provided Git commitIDs, *VulData7* extracts the code of vulnerabilities, (i.e., vulnerable code and its patch) and creates a vulnerability dataset.

To gather the code-base of these systems, we use *FrameVPM* [JRP+19] which is also a publicly available tool[3]. *FrameVPM* is a framework built to evaluate and compare vulnerability prediction models. We also used *FrameVPM* to perform a prediction comparison with existing techniques. Section 7.3.5 elaborates on the re-implementation of existing techniques that we compare with. Table 7.1 provides the details of our dataset. The dataset composed of the vulnerabilities reported in National Vulnerability Database (NVD) [02], and the codebase gathered for the 36 releases of Linux Kernel project [91], 10 releases of Openssl project [98a], and 10 releases of Wireshark project [98b] is publicly available[4], along with our source-code and our re-implemented source-code of the baselines that we compared *TROVON* with.

7.3.3 Implementation and Model Configuration

During the abstraction phase, we rely on the *srcML* tool [CM16] to convert source code into an XML format including tags to identify literals, keywords, identifiers, and comments. This helps in separating user-defined identifiers and string literals (the largest part of the vocabulary) from language keywords (a limited set). Then, ID replacement is performed by a dedicated tool that we implemented. To check whether before and after-fix copies are different, we input the XML produced by *srcML* into the *Gumtree Spoon AST Diff* [FMB+14] tool. The purpose of using *Gumtree Spoon AST Diff* is to achive a fine-grained diff which can ignore irrelevant changes such as whitespaces and/or new line characters. It should be noted that *TROVON* is not bound to the above-mentioned third-party tools. As an alternative, one can use any utility that identifies user-defined entities

[3] `https://github.com/electricalwind/framevpm`
[4] `https://github.com/garghub/TROVON`

and performs a diff.

Our encoder-decoder model is built on top of *tf-seq2seq* [MAP+15], a general-purpose encoder-decoder framework. To configure it, we learn from previous works that apply machine translation to solve software engineering tasks other than vulnerability prediction, *e.g.* [TWB+19a; TWB+19b; GOD+22]. Thus, we rely on a bidirectional encoder as it generally outperforms a unidirectional encoder [BCB14]. We use a Long Short-Term Memory (LSTM) network [HS97] to act as the Recurrent Neural Network (RNN) cell, which was shown to perform better than other common alternatives like simple RNNs or gated recurrent units, in other software engineering prediction tasks [SNL19; Bro18b]. Bucketing and padding are used to deal with the variable length of sequences. To strike a balance between performance and training time, we utilize AttentionLayerBahdanau as our attention class, configured with 2 layered AttentionDecoder and 1 layered BidirectionalRNNEncoder, both with 256 units.

To determine an appropriate number of training steps, we conducted a preliminary study involving a validation set (independent of both the training set and the test set that we use in our experimental evaluation) and trained the model by iterations of 5,000 steps. At the end of each iteration, we check whether the prediction accuracy on the validation set improved. If it improved, then we pursued the training for another iteration, otherwise, stopped. We found out that the model stopped improving at 50,000 steps, which we thus set as a threshold. This order of magnitude is in line with previous research applying machine translation to solve software engineering prediction tasks, e.g., [GOD+22; TWB+19a].

7.3.4 Experimental Settings

Our experimental evaluation is designed to evaluate techniques under *Clean Training Data Settings* and *Realistic Training Data Settings*. We train a model on each release and test the trained model on the following release, (*i.e.,*next release) simulating a typical release-based vulnerability prediction evaluation scenario [JRP+19].

Clean Training Data Settings - *Used in RQs 1, 2 & 3:* In these settings, a prediction model is trained using all the vulnerabilities (vulnerable, *i.e.,* before-fix sequences transformation to non-vulnerable, *i.e.,* after-fix sequences) of a release of a system (Linux Kernel / OpenSSL / Wireshark). The trained models are evaluated based on their predictions in the following release of the same system (e.g., trained on vulnerable components in Linux Kernel release v4.0 and evaluated on all components of v4.1). The components of the following release are converted into sequences that are input to the trained model to get the output sequences. Then, *TROVON* compares the output sequences generated by the trained model with the input sequences. A component is considered vulnerable if any of the output sequences differ from the input sequences, otherwise considered as non-vulnerable.

This training-testing process is repeated for all available releases.

For our release-based experiments where we train the models of different approaches on one release and test the trained models on the next release, in total we have 36 releases of Linux Kernel, 10 releases of Wireshark, and 10 releases of OpenSSL, as mentioned in Table 7.1. In case of *(n)* releases available to us for a system, we can only perform *(n–1)* experiments because in chronological order, the last experiment would be to train a model on $(n–1)^{th}$ release and test the trained model on $(n)^{th}$ release. The reason for such is that we do not have a release to test a model trained on the *nth* release. Hence, for 1 approach, we performed 35 experiments for Linux Kernel, 9 experiments for Wireshark, and 9 experiments for Wireshark. That results to 53 experiments in total (35 + 9 + 9 = 53), for 1 approach.

Realistic Training Data Settings - *Used in RQ4:* In contrast to the *clean* training data settings, in *Realistic Training Data Settings* we consider the date when the vulnerability was fixed. Vulnerability fixing date determines whether a vulnerability is included in the training dataset or not. In these settings, a prediction model (for one release of the system) is trained only on those vulnerabilities that were fixed before the next release date. Then, the trained model is evaluated on all the components of the following release of the same system.

7.3.5 Benchmarks for Vulnerability Prediction

To assess effectiveness, we compare *TROVON* with existing vulnerability prediction techniques. To perform the comparison we use *FrameVPM*, a framework enabling the replication and comparison of vulnerability prediction approaches, introduced by Jimenez *et al.* [JRP+19]. Overall, we compare *TROVON* with:

Software Metrics: Complexity metrics have been extensively used for defect prediction (e.g. [HBB+12]) and vulnerability prediction (e.g. [SW08; SMW+11; CZ11; TW20]). It is based on the idea that complex code is difficult to maintain and test, and thus has a higher chance of having vulnerabilities than simple code. Using FrameVPM, we replicate and compare with the original study from Shin *et al.* [SMW+11] that rely on features related to following metrics:

1. Complexity and Coupling
   (a) *LinesOfCode*: lines of code;
   (b) *PreprocessorLines*: preprocessing lines of code;
   (c) *CommentDensity ratio*: lines of comments to lines of code;
   (d) *CountDeclFunction*: number of functions defined;
   (e) *CountDeclVariable*: number of variables defined;
   (f) *CC*(sum, avg, max): sum, average and max cyclomatic complexity;

(g) *SCC*(sum, avg, max): strict cyclomatic complexity[SMW+11];

(h) *CCE*(sum, avg, max): essential cyclomatic complexity[SMW+11];

(i) *MaxNesting*(sum, avg, max): maximum nesting level of control constructs;

(j) *fanIn*(sum, avg, max): number of inputs, i.e., input parameters and global variables to functions;

(k) *fanOut*(sum, avg, max): number of outputs, i.e., assignments to global variables and parameters of function calls.

2. Code Churn: *added lines, modified lines and deleted lines in the history of a component.*

3. Developer Activity Metrics:

   (a) *number of commits impacting a component;*

   (b) *number of developers modified a component;*

   (c) *current number of developers working on a component.*

Text Mining: It considers a source code component as a collection of terms associated with frequencies, also known as *Bag of Words* (BoW), used for vulnerability prediction [SWH+14]. The source code is broken into a vector of code tokens, and the frequency of each token is then used as the features to build the vulnerability prediction model. Further improvements have been performed to significantly improve its performance, e.g., by pooling frequency values in different bins according to particular criteria to discretize BoW's features [SWH+14; Kon95; TW20].

Imports and *Function Calls*: The work of *Neuhaus et al.* [NZH+07] is based on the observation that the vulnerable components tend to import and call a particular small set of functions. Thus, the features of this simple prediction model are the components' imports and function calls. Following the suggestions of *FrameVPM*, we use imports and function calls as separate sets of features. We train one model based on *Imports* and another based on *Function Calls*, thus implementing one model per set of features.

Devign: The work of *Zhou et al.* [ZLS+19] emphasizes the use of graph neural network for vulnerability detection. With Abstract Syntax Tree (AST) as the backbone, *Zhou et al.* proposed to convert components (vulnerable/non-vulnerable) as code property graphs which helps to solve the problem of information loss during learning. To perform component classification, (*i.e.,*graph-level classification), graph neural network models are trained which are composed of gated graph recurrent layer and convolutional layer, that enables to learn the vulnerable programming

pattern. Since the authors' implementation of the approach is not available, we implemented *Devign* based on our understanding of [ZLS+19] and made it publicly available[5].

LSTM and *LSTM-RF*: The work of *Dam et al.* [DTP+18] focuses to capture *semantic* features of code components (vulnerable/non-vulnerable) and using these features to perform vulnerability prediction. *Dam et al.* asserted that *Long Short Term Memory* (LSTM) [HS97] is highly effective in learning long-term dependencies in sequential data such as text and speech, and can be used to learn features that represent both the semantics of code tokens (semantic features) and the sequential structure of source code (syntactic features). In this approach, components are encoded using the embedding layer, and along with labels (vulnerable/non-vulnerable), are used to train LSTM models. Although these trained LSTM models are capable of prediction, *i.e.,*to provide a probability of a component being vulnerable, the approach extends a step further. The embeddings for the components are extracted using the trained LSTM models, and are used to train binary classifier. Finally, the trained binary classifier provides the probability/likelihood of a component being vulnerable. For *LSTM* approach, we used the trained LSTM models for predictions, and for *LSTM-RF* approach, we used trained binary classifiers for predictions. Here as well, due to unavailable authors' implementation, we implemented the approach based on our understanding of [HS97] and made it publicly available[6].

7.3.6 Performance measurement

Vulnerability prediction modeling is a binary classification problem, thus it can result in four types of outputs: Given a vulnerable component, if it is predicted as vulnerable, then it is a true positive (TP); otherwise, it is a false negative (FN). Given a non-vulnerable component, if it is predicted as non-vulnerable, then it is a true negative (TN); otherwise, it is a false positive (FP). From these, we can compute the evaluation metrics, *i.e.,Precision*, *Recall*, *F-measure*, and *Matthews Correlation Coefficient (MCC)* to evaluate the prediction performance of vulnerability prediction models. Here, MCC is more reliable than others as the classes are of very different sizes, e.g. in the case of Linux Kernel, 3% vulnerable components (Positives) over 97% non-vulnerable components (Negatives).

Table 7.2: Prediction with clean training data, *Clean Training Data Settings* (RQ1)

Release	**MCC**	**F-measure**	**Precision**	**Recall**	**Total Vuln. Comp.**
Linux Kernel					
v3.0	0.70	0.86	0.84	0.89	598
v3.1	0.72	0.87	0.82	0.92	612
v3.2	0.75	0.88	0.86	0.91	612
v3.3	0.70	0.86	0.82	0.91	609
v3.4	0.73	0.88	0.84	0.91	607
v3.5	0.72	0.86	0.94	0.79	609
v3.6	0.74	0.88	0.86	0.90	640
v3.7	0.67	0.85	0.82	0.89	640
v3.8	0.78	0.89	0.92	0.87	632
v3.9	0.69	0.86	0.83	0.90	633
v3.10	0.77	0.89	0.88	0.90	637
v3.11	0.85	0.93	0.93	0.92	613
v3.12	0.76	0.89	0.88	0.90	584
v3.13	0.72	0.87	0.82	0.92	578
v3.14	0.85	0.93	0.93	0.93	573
v3.15	0.78	0.89	0.89	0.90	554
v3.16	0.80	0.91	0.92	0.89	553
v3.17	0.81	0.91	0.91	0.91	443
v3.18	0.81	0.91	0.93	0.89	428
v3.19	0.72	0.87	0.84	0.91	420
v4.0	0.88	0.94	0.96	0.92	417
v4.1	0.86	0.93	0.94	0.93	417
v4.2	0.77	0.88	0.96	0.82	410
v4.3	0.84	0.92	0.94	0.90	391
v4.4	0.82	0.92	0.91	0.93	371
v4.5	0.79	0.90	0.92	0.88	347
v4.6	0.79	0.90	0.88	0.93	330
v4.7	0.79	0.90	0.91	0.90	310
v4.8	0.83	0.92	0.91	0.92	284
v4.9	0.80	0.90	0.90	0.90	259
v4.10	0.79	0.90	0.92	0.88	233
v4.11	0.75	0.88	0.87	0.90	194
v4.12	0.78	0.89	0.93	0.86	176
v4.13	0.79	0.90	0.94	0.86	133
v4.14	0.80	0.91	0.91	0.90	113
Wireshark					
v1.8.0	0.50	0.69	0.97	0.53	138
v1.10.0	0.58	0.77	0.92	0.67	168
v1.11.0	0.78	0.88	0.97	0.81	168
v1.12.0	0.58	0.76	0.95	0.63	165
v1.99.0	0.71	0.85	0.95	0.77	156
v2.0.0	0.59	0.78	0.93	0.67	123
v2.1.0	0.74	0.86	0.98	0.76	116
v2.2.0	0.67	0.83	0.93	0.75	93
v2.4.0	0.17	0.65	0.69	0.61	79
OpenSSL					
v0.9.3	0.83	0.91	1.00	0.83	53
v0.9.4	0.83	0.91	1.00	0.83	56
v0.9.5	0.83	0.91	1.00	0.83	56
v0.9.6	0.67	0.80	1.00	0.67	65
v0.9.7	0.71	0.83	1.00	0.71	78
v0.9.8	0.71	0.83	1.00	0.71	75
v1.0.0	0.71	0.84	0.96	0.75	71
v1.0.1	0.73	0.87	0.91	0.82	48
v1.0.2	0.67	0.80	1.00	0.67	26
Overall					
Average	0.74	0.87	0.91	0.84	334
Median	0.76	0.88	0.92	0.89	330

7.4 Experimental Results

7.4.1 Prediction with clean training data, aka *Clean Training Data Settings* (RQ1)

Table 7.2 records the prediction performance results for the experiments conducted on the 56 releases we study, *i.e.,*36 releases of Linux Kernel, 10 of Wireshark, and 10 of OpenSSL, and the total number of vulnerable components present in every release. As mentioned earlier, here the model is trained on a release and evaluated against the following (next) release of the same system. *TROVON* obtained an overall average (and median) of *MCC= 0.74* (0.76), *F-measure= 0.87* (0.88), *Precision= 0.91* (0.92), and *Recall= 0.84* (0.89) in prediction of vulnerable components in the next release of a project. For almost all releases, *TROVON*'s prediction models trained with the clean data achieved above 0.65 MCC (49 out of 53 releases), above 0.75 F-measure (51 out of 53 releases), above 0.80 Precision (52 out of 53 releases), and above 0.70 Recall (49 out of 53 releases). The results achieved by *TROVON* indicate that the suggested predictions can be considered actionable for security engineers looking to prioritize security inspection and testing efforts [SW13].

> Answer to RQ1: The vulnerability prediction models built on *TROVON* successfully predict the vulnerable components with an average MCC score of 0.74, which can be considered actionable for security engineers to prioritize components for security inspection.

7.4.2 Comparison with existing techniques (RQ2)

Figure 7.3 shows the performance comparison of *TROVON* with existing approaches in a box plot format. Box plots show the distribution of performance indicators (MCC, F-measure, Precision, Recall) for the techniques per project.

We can observe that *TROVON* outperforms the others by achieving higher MCC scores. Table 7.3 summarizes the overall performance of the techniques. Interestingly, *TROVON* achieved higher prediction performance in comparison to existing techniques, with a statistically significant[7] difference. We can also observe that the technique *Function Calls* outperforms the others (*Software Metrics*,

[5]https://github.com/garghub/TROVON/tree/main/devign

[6]https://github.com/garghub/TROVON/tree/main/lstm-rf

[7]We compared the MCC values by using Wilcoxon sign-rank-test [Wil45], and obtained a $p - value < 6.2e{-}9$ with existing approaches. We also compared the effect size of MCC values, by using the Vargha-Delaney A measure [VD00], and obtained a value of lower than 0.07 in every case, clearly indicating that *TROVON* significantly outperforms existing techniques.

Table 7.3: (RQ2) Comparison between existing techniques and *TROVON* under *Clean Training Data Settings* - average (and median)

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Software Metrics	0.49 (0.53)	0.44 (0.48)	0.85 (0.92)	0.32 (0.34)
Imports	0.46 (0.49)	0.43 (0.44)	0.83 (0.88)	0.30 (0.29)
Function Calls	0.52 (0.56)	0.48 (0.50)	0.84 (0.89)	0.36 (0.35)
Text Mining	0.52 (0.55)	0.48 (0.51)	0.83 (0.88)	0.36 (0.38)
Devign	0.33 (0.36)	0.29 (0.32)	0.79 (0.89)	0.19 (0.19)
LSTM	0.25 (0.22)	0.23 (0.18)	0.15 (0.09)	0.92 (0.93)
LSTM-RF	0.47 (0.49)	0.43 (0.42)	0.80 (0.89)	0.32 (0.29)
TROVON	0.74 (0.76)	0.87 (0.88)	0.91 (0.92)	0.84 (0.89)

Imports, *Text Mining*, *Devign*, *LSTM*, and *LSTM-RF*) with its average MCC of 0.52. *TROVON* even outperforms *Function Calls* with its 40.84% higher MCC and 80.67% higher F-measure. It is worth mentioning that the average improvement offered by *TROVON* is 8.68% in Precision and 134.73% in Recall, in comparison to *Function Calls*.
The results show that *TROVON* can provide comparatively better guidance to security engineers than existing techniques, to prioritize components for security inspection [SW13].

Answer to RQ2: When trained with clean data, *TROVON* has significantly higher prediction ability, *i.e.,*, on average, 80.39% improvement in MCC score than existing approaches, which shows that *TROVON* can guide security engineers comparatively better than existing techniques to prioritize security inspection and testing efforts.

7.4.3 Predictions on Seen vs Unseen Vulnerable Components (RQ3)

Table 7.4 shows the average percentages of the seen vulnerable components correctly predicted by *TROVON* and existing techniques across 56 releases of the systems. On average, the models that are based on *TROVON* predict 92.79%, 69.48% and 87.19% of the seen vulnerable components in Linux Kernel, Wireshark, and OpenSSL project releases, respectively. The models based on *LSTM* performs

Table 7.4: (RQ3) Comparison between existing techniques and *TROVON* wrt to their ability to predict correctly already seen vulnerable components, i.e., (classify then as vulnerable)

Approach	**Linux Kernel** 36 releases	**Wireshark** 10 releases	**OpenSSL** 10 releases
Software Metrics	48.12%	54.84%	54.17%
Imports	48.12%	60.76%	50.00%
Function Calls	58.65%	52.69%	64.58%
Text Mining	57.14%	56.99%	64.58%
Devign	32.34%	39.64%	35.69%
LSTM	96.69%	76.43%	95.77%
LSTM-RF	47.66%	48.81%	51.25%
TROVON	92.79%	69.48%	87.19%

the best in identifying already seen vulnerable components, *i.e.,*96.69%, 76.43%, and 95.77% of the vulnerable components identified correctly in Linux Kernel, Wireshark, and OpenSSL project releases, respectively. The percentages gained by *TROVON* are higher than existing techniques, except *LSTM*, by 44.12% for Linux Kernel releases, 17.19% for Wireshark releases, and 33.81% for OpenSSL releases, indicating a high learning potential.

Table 7.5 shows the average percentages of the unseen vulnerable component prediction. On average, *TROVON* based trained models predict 76.53%, 91.03% and 60.07% of the unseen vulnerable components in Linux Kernel, Wireshark, and OpenSSL project releases, respectively. The percentages gained by *TROVON* are higher than existing techniques by 40.05% for Linux Kernel releases, 64.34% for Wireshark releases, and 42.28% for OpenSSL releases, reflecting higher generalization capability in comparison to existing techniques. It is worth noting that *TROVON* obtains all the above mentioned percentages with an MCC of 0.74, on average, which is 80.39% higher than existing techniques.

Answer to RQ3: The models trained on *TROVON* have higher learning potential and generalization capability in comparison to existing approaches in almost all cases.

Table 7.5: (RQ3) Comparison between existing techniques and *TROVON* wrt to their ability to predict correctly already unseen vulnerable components, i.e., (classify then as vulnerable)

Approach	**Linux Kernel** 36 releases	**Wireshark** 10 releases	**OpenSSL** 10 releases
Software Metrics	09.09%	15.48%	18.18%
Imports	50.00%	08.93%	23.08%
Function Calls	56.10%	60.00%	09.09%
Text Mining	45.45%	16.07%	18.18%
Devign	32.54%	33.13%	14.99%
LSTM	25.79%	27.63%	23.02%
LSTM-RF	36.39%	25.62%	18.01%
TROVON	76.53%	91.03%	60.07%

7.4.4 Comparison with existing techniques under *Realistic Training Data Settings* (RQ4)

As mentioned before, in *Realistic Training Data Settings*, a model is trained only on the vulnerabilities of a release that were detected / made public before the next release date of the system. This unavoidably introduces mislabeling noise because every component that has no vulnerabilities uncovered before the next release date, is considered non-vulnerable during training. Figure 7.4 shows that the performance of all the techniques is considerably reduced in the *Realistic Training Data Settings*, in comparison to the *Clean Training Data Settings*. The results are in accordance with Jimenez *et al.* [JRP+19]. Despite this drop in performance, *TROVON* outperforms existing techniques with a statistically significant[8] sizeable difference.

Table 7.6 shows the overall average and median performance statistics for each technique. We can observe that the technique *LSTM-RF* outperforms the other existing techniques (*Software Metrics*, *Imports*, *Function Calls*, *Text Mining*, *Devign*, and *LSTM*) with its average MCC of 0.29. *TROVON* even outperforms *LSTM-RF* in all the performance measures, *i.e.,*35.52% higher MCC, 148.91% higher

[8]We compared the MCC values using Wilcoxon sign-rank-test and obtained a $p - value < 7.7e{-}9$ with existing approaches. We also compared the MCC values with the Vargha-Delaney A measure and obtained a value lower than 0.03 in every case, indicating that *TROVON* significantly outperforms existing techniques.

Table 7.6: (RQ4) Comparison between existing techniques and *TROVON* under *Realistic Training Data Settings* - average (median)

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Software Metrics	0.06 (0.03)	0.03 (0.01)	0.31 (0.30)	0.02 (0.01)
Imports	0.06 (0.06)	0.04 (0.02)	0.34 (0.33)	0.02 (0.01)
Function Calls	0.07 (0.05)	0.04 (0.02)	0.34 (0.33)	0.03 (0.01)
Text Mining	0.06 (0.05)	0.04 (0.01)	0.29 (0.28)	0.02 (0.01)
Devign	0.13 (0.02)	0.12 (0.03)	0.34 (0.06)	0.18 (0.02)
LSTM	0.16 (0.14)	0.14 (0.11)	0.08 (0.06)	0.83 (0.86)
LSTM-RF	0.29 (0.27)	0.28 (0.23)	0.47 (0.49)	0.21 (0.15)
TROVON	0.39 (0.41)	0.69 (0.68)	0.86 (0.87)	0.58 (0.56)

F-measure, 81.61% higher Precision, and 183.90% higher Recall, in comparison to *LSTM-RF*. This indicates that *TROVON* has much higher accuracy in vulnerability prediction than existing techniques in the *Realistic Training Data Settings* as well.

Answer to RQ4: Under the *Realistic Training Data Settings*, *TROVON* based models obtain significantly higher accuracy in vulnerability prediction, *i.e.,* 3.63 times higher MCC scores than existing techniques.

7.5 TROVON with Bi-LSTM

Although training a machine translator (viz. an encoder-decoder sequence to sequence model) to identify vulnerable components, is an integral part of *TROVON*'s architecture, we also replicated our experiments with *Bi-LSTM* models. We kept the entire experimental setting the same, (i.e., both *Clean Training Data Settings* and *Realistic Training Data Settings* with the corresponding training and test sets) and trained *Bi-LSTM* models instead of training sequence to sequence models. For this experiment, we adhere to the key idea of *TROVON* and train *Bi-LSTM* models on the validated data, (i.e., only on components known to be vulnerable and leave aside the non-vulnerable ones). We name this approach *TROVON-BILSTM*.

Tables 7.7 and 7.8 show the average and median performance statistics of *TROVON-BILSTM* in *Clean Training Data Settings* and *Realistic Training Data Settings*, respectively. We also mention the results of *TROVON* for comparison. On average, in *Clean Training Data Settings*, *TROVON-BILSTM* achieved 0.73 MCC,

Table 7.7: Comparison between *TROVON-BILSTM* and *TROVON* under *Clean Training Data Settings* - average (median)

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Linux Kernel				
TROVON-BILSTM	0.73 (0.70)	0.84 (0.83)	0.84 (0.83)	0.84 (0.84)
TROVON	0.78 (0.78)	0.89 (0.89)	0.89 (0.91)	0.90 (0.90)
Wireshark				
TROVON-BILSTM	0.54 (0.54)	0.72 (0.72)	0.85 (0.85)	0.63 (0.61)
TROVON	0.59 (0.59)	0.79 (0.78)	0.92 (0.95)	0.69 (0.67)
OpenSSL				
TROVON-BILSTM	0.71 (0.68)	0.82 (0.79)	0.93 (0.98)	0.73 (0.68)
TROVON	0.74 (0.71)	0.86 (0.84)	0.99 (0.99)	0.76 (0.75)

0.84 F-1, 0.84 Precision, and 0.84 Recall for Linux Kernel releases; 0.54 MCC, 0.72 F-1, 0.85 Precision, and 0.63 Recall for Wireshark releases; and 0.71 MCC, 0.82 F-1, 0.95 Precision, and 0.73 Recall for OpenSSL releases. In *Realistic Training Data Settings*, *TROVON-BILSTM* achieved 0.38 MCC, 0.65 F-1, 0.84 Precision, and 0.53 Recall for Linux Kernel releases; 0.34 MCC, 0.66 F-1, 0.73 Precision, and 0.61 Recall for Wireshark releases; and 0.37 MCC, 0.68 F-1, 0.75 Precision, and 0.62 Recall for OpenSSL releases.

Figures 7.5 and 7.6 show the performance comparison of *TROVON-BILSTM* and *TROVON* in *Clean Training Data Settings* and *Realistic Training Data Settings*, respectively. The figures show that *TROVON* performs comparatively better than *TROVON-BILSTM*. Overall, when trained with vulnerabilities, in *Clean Training Data Settings*, *TROVON* outperforms *TROVON-BILSTM* by 6.49% in MCC, 6.63% in F-1, 6.40% in Precision, and 6.75% in Recall. In *Realistic Training Data Settings*, *TROVON* outperforms *TROVON-BILSTM* by 5.08% in MCC, 5.21% in F-1, 5.01% in Precision, and 5.43% in Recall.

7.6 Threats To Validity

Construct Validity: We use *VulData7* [JPT18] for data collection using the Git commit IDs provided in the CVE-NVD database. This process ensures the retrieval of known and fixed vulnerabilities, whereas undiscovered or unfixed vulnerabilities are ignored. This may result in false negatives with a potential impact on our

Table 7.8: Comparison between *TROVON-BILSTM* and *TROVON* under *Realistic Training Data Settings* - average (median)

Approach	**MCC**	**F-measure**	**Precision**	**Recall**
Linux Kernel				
TROVON-BILSTM	0.38 (0.39)	0.65 (0.67)	0.84 (0.87)	0.53 (0.54)
TROVON	0.40 (0.41)	0.68 (0.68)	0.88 (0.88)	0.56 (0.55)
Wireshark				
TROVON-BILSTM	0.34 (0.36)	0.66 (0.65)	0.73 (0.70)	0.61 (0.62)
TROVON	0.37 (0.38)	0.72 (0.72)	0.79 (0.79)	0.66 (0.66)
OpenSSL				
TROVON-BILSTM	0.37 (0.31)	0.68 (0.68)	0.75 (0.74)	0.62 (0.61)
TROVON	0.41 (0.31)	0.73 (0.68)	0.81 (0.78)	0.67 (0.62)

measurements. However, given the size of Linux Kernel, Wireshark, and OpenSSL and their long history of vulnerability reports, we believe that it is unlikely to have many such cases.

Another concern originates from our choice to learn from the vulnerable and fixed pairs of components. Since *TROVON* has access to this information one can argue that the improved performance is due to this additional knowledge of fixed components. To diminish this concern we also included the fixed versions of the vulnerable files in the training set for training existing techniques, but this resulted in negligible differences in their performance.

One may wonder if most of the vulnerabilities are introduced due to code changes performed between the releases and whether every changed component between adjacent releases can be flagged as vulnerable. We analyzed our data and found that the results are close to random guessing with MCC- 0.06, 0.09, 0.1 and Precision- 0.04, 0.08, 0.14 for Linux Kernel, Wireshark, and OpenSSL project releases, respectively. These results are in accordance with the findings of Jimenez et al. [JRP+19] that most vulnerabilities span across multiple releases without being detected, and mislead the predictions, *e.g.* an existing vulnerability in release *R1* may get detected and fixed in the release *R4*. Also, many files are modified between the releases, *i.e.,*29.95%, 72.53%, and 73.58% of the files, on average, are changed for Linux Kernel, Wireshark, and OpenSSL, which adds to the imprecision of this baseline by producing excessive numbers of false positives/negatives.

Internal Validity: We do not consider non-vulnerable components for training

as these files can in fact be vulnerable (vulnerability undetected till date) and may mislead our predictor. Still, we train on the unchanged and fixed parts of the vulnerable components as we believe that these are unlikely to be vulnerable. To support this intuition, we checked our data and found that it is indeed true, *i.e.,*components having more than one vulnerability, with one fixed and the other not, are on average 0.037%, 0.19%, and 0.24% of the Linux Kernel, Wireshark, and OpenSSL components per release.

We use *FrameVPM* [JRP+19] to implement vulnerability prediction models for *Software Metrics*, *Imports*, *Function Calls*, and *Text Mining*. As none of the replicated approaches provide a replication package, the framework may not have implemented precisely the original approaches. To reduce this threat we inspected the code, parameters, and experiment decisions to perform the most accurate replication possible. Given that our results are in line with the previous replication studies [JPL16; JRP+19] and the original studies [SMW+11; NZH+07], we believe this threat is of less significance.

Similarly, we implement *Devign*, *LSTM*, and *LSTM-RF* based on our understanding of authors' work described in the available articles because the author's implementation/source-code of these approaches is not available. Still, there is a possibility that we may not have implemented the original approaches as precisely as the authors of these approaches would have. Nevertheless, these approaches make the clean labeling assumption [JRP+19] thereby experimenting fundamental limitations on their performance. This is actually the key reason why previous work reports much better results. Nevertheless, when using *Clean Training Data Settings*, we found F-1 scores of 32.73% and 36.54% for Linux Kernel and Wireshark, which are in line with the results reported by Zhou *et al.* [ZLS+19] (*i.e.,*F-1 score of 24.64% and 42.05% for Linux Kernel and Wireshark), in their case of imbalanced data (the only case that is somehow comparable with our analysis).

External Validity: Although the study expands its evaluation to three security-critical open source systems, the results may not generalize to other projects (e.g., Android). Additional studies are required to sufficiently take care of the generalization threat. Also, we split the methods into sequences of no more than *50* tokens each. Method-splitting in larger sequences may require more training time and computational resources but can lead to better results.

7.7 Data Availability

The dataset consisting of the codebase and the vulnerabilities, (i.e., the vulnerable and the corresponding fixed components) of the 36 releases of Linux Kernel, 10 releases of Openssl, and 10 releases of Wireshark, along with *TROVON*'s source code, is publicly available in our GitHub repository[9]. In addition to the source

[9]https://github.com/garghub/TROVON

code of *TROVON*, we also implement existing approaches due to their unavailable authors' implementation. Our implementations of the existing approaches which we compare *TROVON* with are also available in this repository.

7.8 Conclusion

In this chapter, we presented *TROVON*, a machine translation based approach to automatically learn to predict vulnerable components from noisy historical data. Taking advantage of the large amounts of historical data, our predictions can be used to assist developers in code reviews and security testing. The important advantage of *TROVON* is that it is completely automatic as it learns latent features (context, patterns, etc.) linked with vulnerabilities based on information mining from code repositories (in particular by analyzing historical vulnerability fixes and their context). We empirically evaluated the effectiveness of *TROVON* following the methodological guidelines set by Jimenez et al. [JRP+19]. In particular, we demonstrated that *TROVON* can mitigate the problem of real-world noisy data on the releases of the three security-critical open source systems that were used by previous research. Moreover, we showed that *TROVON* outperforms existing techniques under both, clean and realistic, (*i.e.,*noisy) training data settings. On average, when trained on clean data, *TROVON* achieved an overall improvement of 80.39% in MCC score. Moreover, in *Realistic Training Data Settings*, *TROVON* achieved 3.63 times higher MCC score in comparison to existing approaches.

Figure 7.3: Comparison with existing approaches (RQ2) in *Clean Training Data Settings*: When trained with clean data, *TROVON* outperforms existing approaches with an average improvement in MCC, F-measure, Precision, and Recall of 80.39%, 132.95%, 83.96%, and 155.33%, respectively.

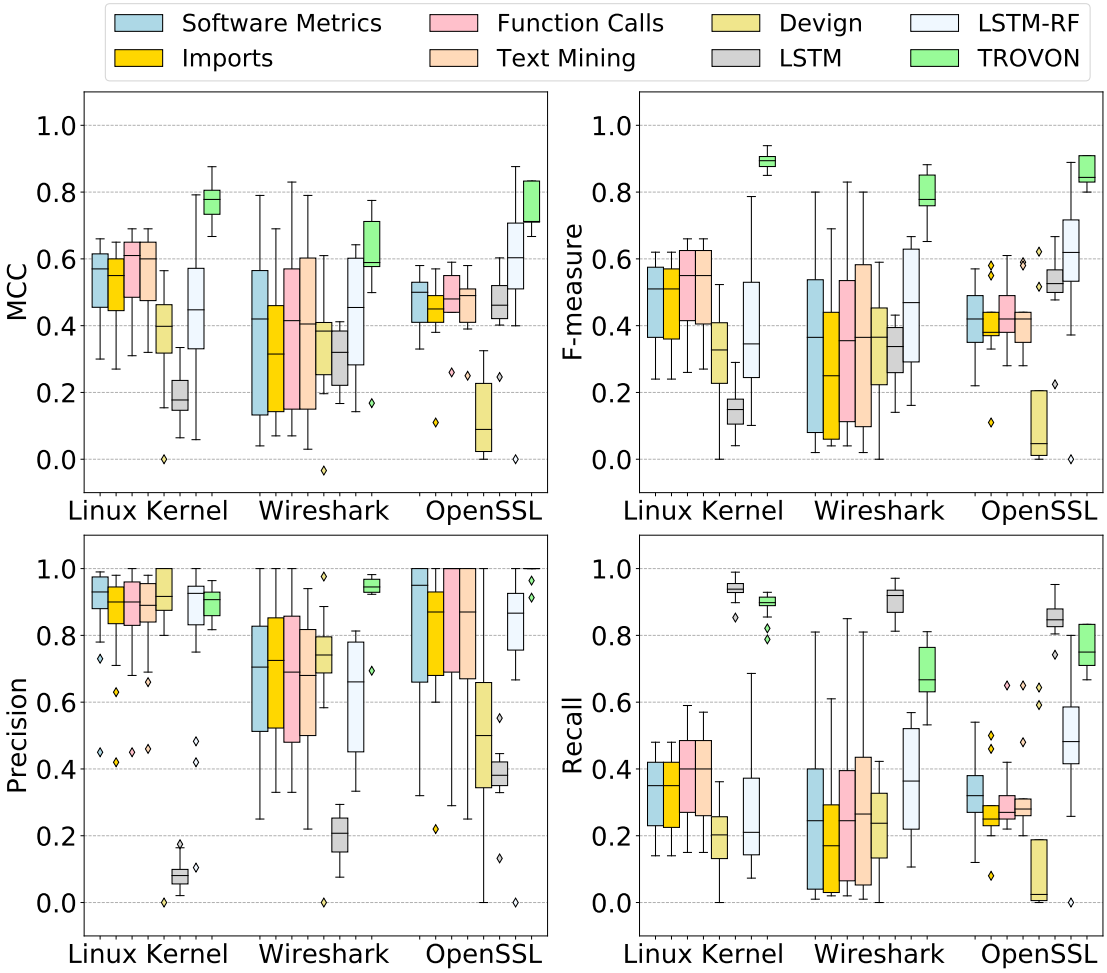


Figure 7.4: Comparison with existing techniques in *Realistic Training Data Settings* (RQ4): Despite a reduced performance of models when trained with realistic training data, *TROVON* significantly outperforms existing techniques with 3.63 times higher MCC, 11.82 times higher F-measure, 2.66 times higher Precision, and 15.25 times higher Recall, respectively.

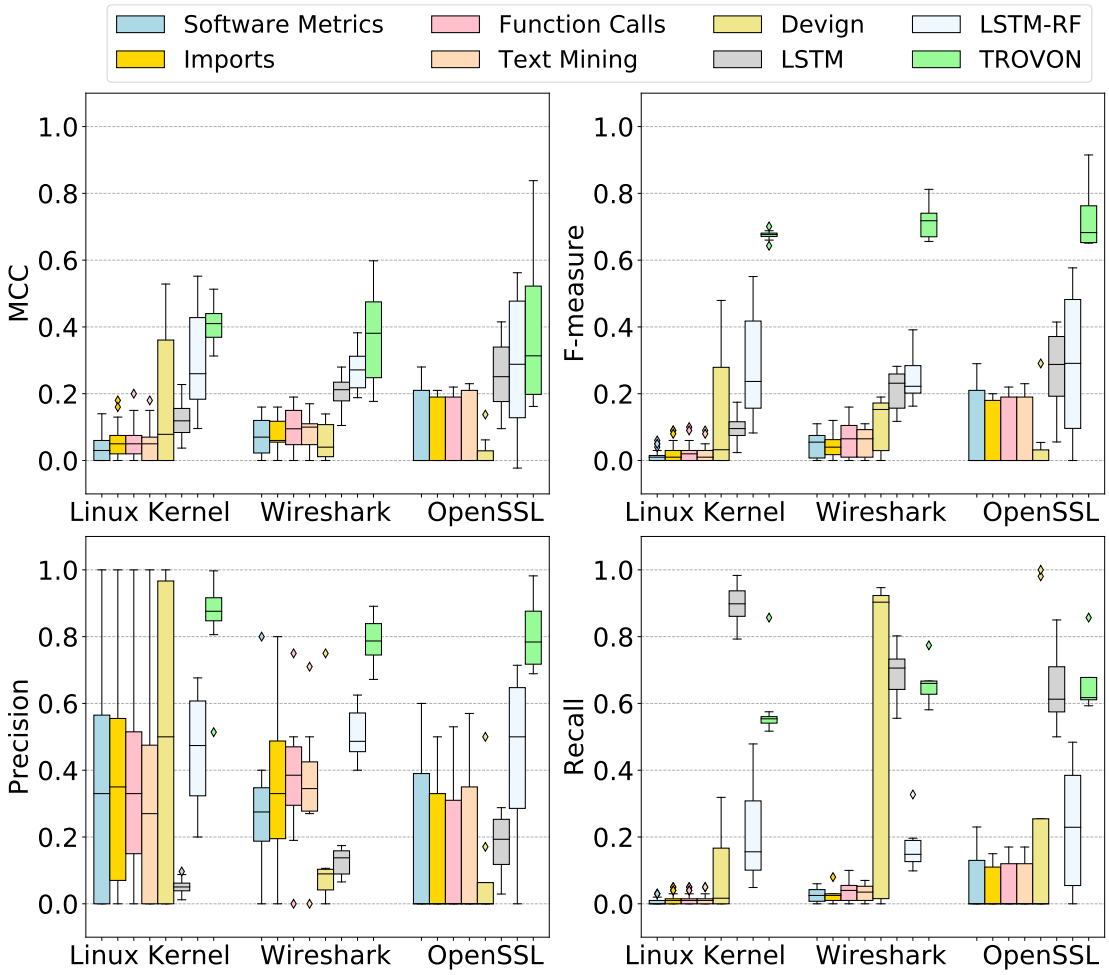


Figure 7.5: Comparison between *TROVON-BILSTM* and *TROVON* under *Clean Training Data Settings*: *TROVON* outperforms *TROVON-BILSTM* by 6.49% in MCC, 6.63% in F-1, 6.40% in Precision, and 6.75% in Recall.

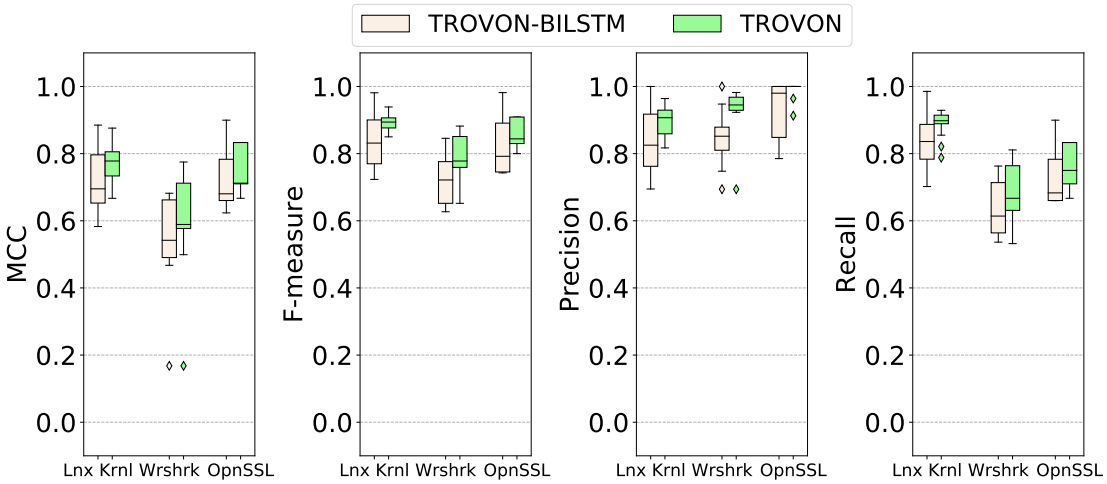
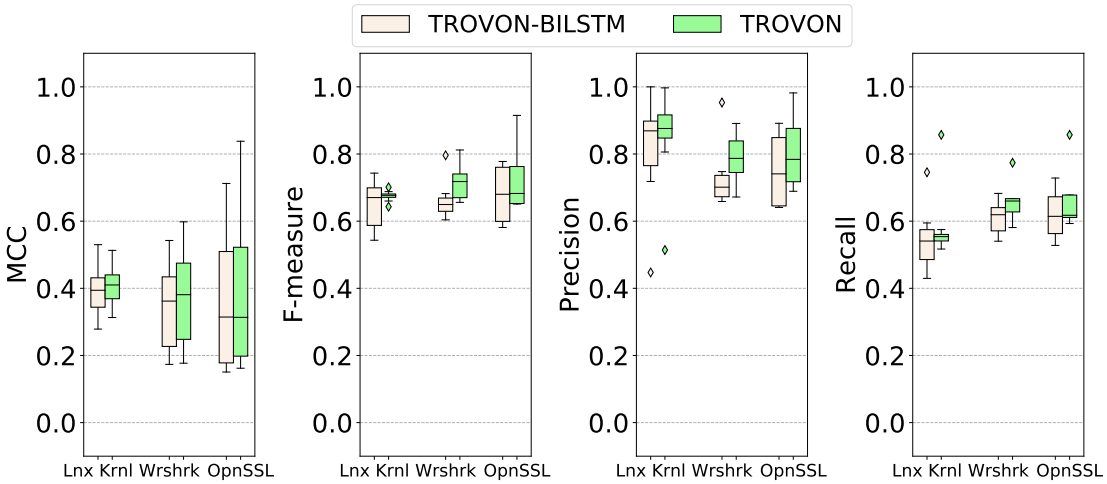


Figure 7.6: Comparison between *TROVON-BILSTM* and *TROVON* under *Realistic Training Data Settings*: *TROVON* outperforms *TROVON-BILSTM* by 5.08% in MCC, 5.21% in F-1, 5.01% in Precision, and 5.43% in Recall.



8

Conclusion

This chapter presents the overall conclusion of the dissertation and proposes potential research directions.

Contents

8.1 Summary of contributions

In this dissertation, we aim to tackle one of the most significant challenges of mutation testing - its application cost. The complexity of selecting the most useful mutants to test is compounded by the sheer volume of generated mutants and the lack of advanced knowledge of which ones are effective for analysis or testing. To overcome this hurdle, we explore the application of context-aware machine learning techniques to identify subsets of mutants that are most effective for a given task. Our research also addresses the limitations of mutation testing in security testing, particularly vulnerability prediction. We present the following four distinct contributions to the field that address various aspects of this problem: 1) *Cerebro* - a deep learning approach to statically select subsuming mutants in order to preserve the mutation testing benefits while limiting the application cost, 2) *Seeker* - a deep learning approach to statically select *Assertion Inferring Mutants* in order to enable an assertion inference capability comparable to the full mutation analysis while significantly limiting the execution cost and improving the scalability of assertion inference techniques, 3) *Mystique* - a deep learning approach to automatically identify vulnerability-mimicking mutants in order to design test cases to tackle mimicked vulnerabilities, and 4) *TROVON* - a deep learning based vulnerability prediction approach that bypasses the key problem faced by previous techniques and outperforms them. Through our experiments, we demonstrate the effectiveness and efficiency of our proposed methods in achieving significant improvements in respective metrics compared to existing approaches. Our work provides valuable insights into the effectiveness of machine learning guided mutation and security testing. The source code, datasets, and our implementation of existing approaches resulting from the work presented in this dissertation are made publicly available. The following details our contributions in detail.

As our first contribution, we proposed *Cerebro*, a method that learns to select subsuming mutants (a subset of mutants that subsumes the others, i.e., tests killing them also kill all the mutants of the given mutant set) from given mutant sets. Our experiments showed that *Cerebro* identified subsuming mutants with high precision and recall at an inter-project scenario (trained on different projects than the ones it was evaluated). These predictions enable testers to design test cases capable of killing more than twice the subsuming mutants that they would kill if they were using either randomly selected mutants or another previously proposed machine learning-based mutant selection technique. At the same time *Cerebro* entails the analysis of significantly fewer equivalent mutants and mutant executions, indicating a large reduction in the practical effort/cost of the approach.

As our second contribution, we proposed *Seeker*, a method that learns to select *Assertion Inferring Mutants* (a small subset of mutants that is suitable for assertion inference) from given mutant sets. Our experiments show that *Seeker* identified

assertion inferring mutants with high prediction performance. These predictions enable many times faster inference with minor effectiveness loss compared to the use of all mutants. Similarly, *Seeker*'s predictions infer almost all of the total ground truth assertions, which is substantially greater than *Subsuming Mutant Selection* and *Random Mutant Selection*. Moreover, *Seeker* enables the assertion inference technique SpecFuzzer to scale on all our large subjects (by inferring assertions where SpecFuzzer failed previously due to timeouts) in comparison to *Random Mutant Selection* which failed to infer any assertion in half of the cases.

As our third contribution, we showed that language model based mutation testing tools can produce *Vulnerability-mimicking Mutants*, i.e., mutants that mimic the observable behavior of vulnerabilities. Since these mutants are significantly fewer among the entire mutant set, there is a need for a static approach to identify such mutants. To achieve this, we proposed *Mystique*, a method that learns to select *Vulnerability-mimicking Mutants* from a given mutant's code context. Our experiments show that *Mystique* identified *Vulnerability-mimicking Mutants* with high prediction performance, which indicates that the features of *Vulnerability-mimicking Mutants* can be automatically learned by machine learning models to statically predict these without the need of investing effort in defining such features.

As our fourth contribution, we proposed *TROVON*, a machine translation based approach to automatically learn to predict vulnerable components from noisy historical data. Taking advantage of the large amounts of historical data, our predictions can be used to assist developers in code reviews and security testing. The important advantage of *TROVON* is that it is completely automatic as it learns latent features (context, patterns, etc.) linked with vulnerabilities based on information mining from code repositories (in particular by analyzing historical vulnerability fixes and their context). We empirically evaluated the effectiveness of *TROVON* following the methodological guidelines set by Jimenez et al. [JRP+19]. In particular, we demonstrated that *TROVON* can mitigate the problem of real-world noisy data on the releases of the three security-critical open source systems that were used by previous research. Moreover, we showed that *TROVON* significantly outperforms existing techniques under both, clean and realistic, (*i.e.,*noisy) training data settings.

8.2 Future prospects

In the following, we discuss potential future research that follows the contributions and ideas presented in this dissertation:

- **Transfer learning via pre-trained models trained on large code corpus:** Recently, it has become increasingly common to pre-train the entire model on a data-rich task, which causes the model to develop general-purpose abilities and knowledge that can then be transferred to downstream

tasks [RSR+19]. In this practice *aka* Transfer Learning and its applications to computer vision [OBL+14; JSD+14], pre-training is typically done via supervised learning on a large labeled data set like ImageNet [RDS+15]. In contrast, modern techniques for transfer learning in Natural Language Processing (*NLP*) often pre-train using unsupervised learning on unlabeled data [DCL+19; LLS+21]. The resulting pre-trained models are further trained on specialized datasets to accomplish the desired tasks. Unsupervised pre-training for *NLP* is attractive and seems a good fit for neural networks as it has been shown to exhibit remarkable scalability, *i.e.,*it is often possible to achieve better performance simply by training a larger model on a larger data set [HNA+17; SMM+17; JVS+16; MGR+18]. It will be worthwhile to explore such available pre-trained models [MSC+21; FGT+20] and if these can be further refined to address our specific prediction tasks.

- **Expanding Mutation Techniques for Improved Vulnerability Mimicking** In Chapter 6, we study the extent to which the mutants produced by the language models can semantically mimic the behavior of vulnerabilities aka *Vulnerability-mimicking Mutants*. Since our vulnerability-mimicking mutants cannot mimic all the vulnerabilities, we perceive that these mutants are not a complete representation of all the vulnerabilities. This suggests that augmenting the mutation techniques to include a wider range of mutation operators may improve the ability to mimic more vulnerabilities. In future work, we plan to include additional mutation operators that have been shown to be effective in generating diverse and realistic mutants in the context of software testing, such as syntactic and semantic mutations (e.g., [OL17; CDH13]). Such mutation techniques have been shown to be effective in generating high-quality mutants that more accurately reflect real-world software faults. By including additional mutation techniques, we can increase the proportion of mutants that mimic the observable behavior of vulnerabilities. Furthermore, increasing the proportion of mutants mimicking vulnerabilities should also improve the effectiveness of *Mystique* in predicting *Vulnerability-mimicking Mutants*, as the models will have more samples to learn from.

Abbreviations

API Application Programming Interface.
AST Abstract Syntax Tree.

BoW Bag of Words.

CVE Common Vulnerability Exposures.

EMT Evolutionary Mutation Testing.

F2 Weighted Harmonic Mean of Precision and Recall.
FN False Negative.
FP False Positive.

IEEE Institute of Electrical and Electronics Engineers.

LSTM Long Short Term Memory.

MCC Matthews Correlation Coefficient.
MS* Subsuming Mutation Score.

NIST National Institute of Standards and Technology.
NVD National Vulnerability Database.

PIT Pitest.
PMT Predictive Mutation Testing.
PoV Proof of Vulnerability.

RBAC Role Based Access Control.
RNN Recurrent Neural Network.

SQL Structured Query Language.

TAC Test Adequacy Criteria.
TCE Trivial Compiler Equivalence.
TN True Negative.

TP True Positive.

USD United States Dollar.

XSS Cross-Site Scripting.

List of Publications and Software

Papers included in the dissertation

- Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: static subsuming mutant selection. *IEEE Transactions on Software Engineering*:1–1, 2022. DOI: 10.1109/TSE.2022.3140510
- Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: how to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022. DOI: 10.1007/s10664-022-10197-4. URL: https://doi.org/10.1007/s10664-022-10197-4
- Aayush Garg, Renzo Degiovanni, Facundo Molina, Mike Papadakis, Nazareno Aguirre, Maxime Cordy, and Yves Le Traon. Assertion inferring mutants. *CoRR*, abs/2301.12284, 2023. DOI: 10.48550/arXiv.2301.12284. arXiv: 2301.12284. URL: https://doi.org/10.48550/arXiv.2301.12284
- Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Vulnerability mimicking mutants. *CoRR*, abs/2303.04247, 2023. DOI: 10.48550/arXiv.2303.04247. arXiv: 2303.04247. URL: https://doi.org/10.48550/arXiv.2303.04247

Papers not included in the dissertation

- Milos Ojdanic, Ahmed Khanfir, Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. On Comparing Mutation Testing Tools through Learning-based Mutant Selection. In IEEE/ACM AST@ICSE 2023. ACM/IEEE 2023
- Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies. *CoRR*, abs/2112.14508, 2021. arXiv: 2112.14508. URL: https://arxiv.org/abs/2112.14508

Software developed during Ph.D.

- Cerebro (Cerebro: Static Subsuming Mutant Selection)
  - `https://github.com/garghub/Cerebro`
- Seeker (Seeker: Efficient Class Specification Inference)
  - `https://github.com/garghub/seeker`
- Mystique (Mystique: Enabling Security conscious Mutation Testing using Language Models)
  - `https://github.com/garghub/mystique`
- TROVON (Learning from What We Know: How to Perform Vulnerability Prediction using Noisy Historical Data)
  - `https://github.com/garghub/TROVON`

List of figures

List of tables

Bibliography

[02] National vulnerability database. https://nvd.nist.gov, 2002 (cited on pages 25, 36, 129).

[09] Definition of vulnerability. https://www.cve.org/ResourcesSupport/Glossary, 2009 (cited on pages 10, 24).

[12] Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012. DOI: 10.1109/MSP.2012.2205597. URL: https://doi.org/10.1109/MSP.2012.2205597 (cited on page 20).

[14] The heartbleed bug. https://heartbleed.com/, 2014 (cited on page 128).

[18a] Cve-2018-1000850. https://nvd.nist.gov/vuln/detail/CVE-2018-1000850, 2018 (cited on page 103).

[18b] Cve-2018-17201. https://nvd.nist.gov/vuln/detail/CVE-2018-17201, 2018 (cited on page 103).

[20a] Linux in 2020: 27.8 million lines of code in the kernel. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/, 2020 (cited on page 121).

[20b] Vulnerabilities. https://owasp.org/www-community/vulnerabilities/, 2020 (cited on page 121).

[91] Linux kernel. https://www.kernel.org, 1991 (cited on pages 128, 129).

[98a] Openssl. https://www.openssl.org, 1998 (cited on pages 128, 129).

[98b] Wireshark. https://www.wireshark.org, 1998 (cited on pages 128, 129).

[AAB+16] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467 (cited on pages 87, 94).

[ABL+06] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006. DOI: 10.1109/TSE.2006.83 (cited on pages 3, 43, 53, 72).

[ABL+19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. Code2seq: generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=H1gKYo09tX (cited on pages 86, 106).

[Acr80] Allen Troy Acree. *On Mutation*. PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1980 (cited on page 17).

[ADO14] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 21–30. IEEE Computer Society, 2014. DOI: 10.1109/ICST.2014.13. URL: https://doi.org/10.1109/ICST.2014.13 (cited on pages 7, 18, 19, 33, 43, 55, 71, 72, 101, 102).

[Alp04] Ethem Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning. MIT Press, 2004. ISBN: 978-0-262-01211-9 (cited on pages 7, 19).

[AO08a] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN: 978-0-521-88038-1. DOI: 10.1017/CBO9780511809163. URL: https://doi.org/10.1017/CBO9780511809163 (cited on pages 3, 22, 101).

[AO08b] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1st edition, 2008. ISBN: 0521880386 (cited on page 5).

[BA82] Timothy Alan Budd and Dana Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, March 1982 (cited on pages 17, 43).

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. Yoshua Bengio and Yann LeCun, editors, 2014. arXiv: 1409.0473 [cs.CL]. URL: http://arxiv.org/abs/1409.0473 (cited on pages 21, 87, 107, 130).

[BCS+16] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philémon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4945–4949. IEEE, 2016. DOI: 10.1109/ICASSP.2016.7472618. URL: https://doi.org/10.1109/ICASSP.2016.7472618 (cited on pages 56, 87).

[BGK+18] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018. DOI: 10.1145/3213846.3213872. URL: https://doi.org/10.1145/3213846.3213872 (cited on page 81).

[BGL+17] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017. arXiv: 1703.03906. URL: http://arxiv.org/abs/1703.03906 (cited on pages 21, 44, 87, 107, 122).

[BGV10] Luciano Baresi, Angelo Gargantini, and Paolo Vavassori. Using mutation analysis for testing security-critical software. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2010 (cited on page 11).

[BHM+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015. DOI: 10.1109/TSE.2014.2372785. URL: https://doi.org/10.1109/TSE.2014.2372785 (cited on page 22).

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN: 9780387310732. URL: https://www.worldcat.org/oclc/71008143 (cited on pages 19, 20).

[BNM21] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In Shane McIntosh, Xin Xia, and Sousuke Amasaki, editors, *PROMISE '21: 17th International Conference on Predictive Models and Data Analytics in Software Engineering, Athens Greece, August 19-20, 2021*, pages 30–39. ACM, 2021. DOI: 10.1145/3475960.3475985. URL: https://doi.org/10.1145/3475960.3475985 (cited on page 36).

[BOP11] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 69–77. Springer, 2011. DOI: 10.1007/978-3-642-21768-5_6. URL: https://doi.org/10.1007/978-3-642-21768-5%5C_6 (cited on page 101).

[BOP12] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 253–262. IEEE, 2012. DOI: 10.1109/SERE.2012.38. URL: https://doi.org/10.1109/SERE.2012.38 (cited on page 37).

[Bre01] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324 (cited on pages 82, 88, 107).

[Bro18a] Jason Brownlee. Encoder-decoder recurrent neural network models for neural machine translation. https://machinelearningmastery.com/encoder-decoder-recurrent-neural-network-models-neural-machine-translation/, 2018 (cited on page 21).

[Bro18b] Jason Brownlee. When to use mlp, cnn, and rnn neural networks. https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks, 2018 (cited on page 130).

[BSd+22] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code. *CoRR*, abs/2206.01335, 2022. DOI: 10.48550/arXiv.2206.01335. arXiv: 2206.01335. URL: https://doi.org/10.48550/arXiv.2206.01335 (cited on page 101).

[BSF22] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Diaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 464–468. ACM, 2022. DOI: 10.1145/3524842.3528482. URL: https://doi.org/10.1145/3524842.3528482 (cited on pages 36, 112).

[BWB+21] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*, pages 268–277. IEEE, 2021. DOI: 10.1109/ICSE-SEIP52600.2021.00036. URL: https://doi.org/10.1109/ICSE-SEIP52600.2021.00036 (cited on pages 4, 101).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, San Diego, California. USENIX Association, 2008 (cited on pages 17, 55, 71).

[CDH13] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. *Sci. Comput. Program.*, 78(4):345–363, 2013. DOI: 10.1016/j.scico.2011.03.011. URL: https://doi.org/10.1016/j.scico.2011.03.011 (cited on page 150).

[CKT+21] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021. DOI: 10.1109/TSE.2019.2940179. URL: https://doi.org/10.1109/TSE.2019.2940179 (cited on page 101).

[CLH+16] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, Saarbrücken, Germany. Association for Computing Machinery, 2016. ISBN: 9781450343909. DOI: 10.1145/2931037.2948707. URL: https://doi.org/10.1145/2931037.2948707 (cited on pages 37, 55, 70).

[CM16] M. L. Collard and J. I. Maletic. Srcml 1.0: explore, analyze, and manipulate source code. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 649–649, 2016 (cited on pages 56, 129).

[CPB+20] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020. DOI: 10.1007/s10664-019-09778-7. URL: https://doi.org/10.1007/s10664-019-09778-7 (cited on pages 17, 33, 43, 53, 56, 57, 68, 71, 86, 89, 102, 105, 112).

[CPC+21] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. Killing stubborn mutants with symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 30(2), January 2021. ISSN: 1049-331X. DOI: 10.1145/3425497. URL: https://doi.org/10.1145/3425497 (cited on pages 17, 54, 55, 70, 71, 74, 75).

[CPL19] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Mart: a mutant generation tool for llvm. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1080–1084, Tallinn, Estonia. Association for Computing Machinery, 2019. ISBN: 9781450355728. DOI: 10.1145/3338906.3341180. URL: https://doi.org/10.1145/3338906.3341180 (cited on page 54).

[CPT+17] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608. IEEE / ACM, 2017. DOI: 10.1109/ICSE.

2017.61. URL: https://doi.org/10.1109/ICSE.2017.61 (cited on pages 3–5, 33, 34, 43, 101).

[CR06] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006. DOI: 10.1145/1127878.1127900. URL: https://doi.org/10.1145/1127878.1127900 (cited on page 22).

[CvMG+14] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics, October 2014. DOI: 10.3115/v1/D14-1179. URL: https://www.aclweb.org/anthology/D14-1179 (cited on pages 56, 87).

[CZ11] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J. Syst. Archit.*, 57(3):294–313, March 2011. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.06.003. URL: https://doi.org/10.1016/j.sysarc.2010.06.003 (cited on pages 38, 131).

[CZ18] Lingchao Chen and Lingming Zhang. Speeding up mutation testing via regression test selection: an extensive study. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 58–69. IEEE Computer Society, 2018. DOI: 10.1109/ICST.2018.00016. URL: https://doi.org/10.1109/ICST.2018.00016 (cited on page 72).

[DCL+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*:4171–4186, 2019. Jill Burstein, Christy Doran, and Thamar Solorio, editors. DOI: 10.18653/v1/n19-1423. URL: https://doi.org/10.18653/v1/n19-1423 (cited on pages 68, 150).

[DEG+06] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In Lori L.

Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244. ACM, 2006. DOI: 10.1145/1146238.1146266. URL: https://doi.org/10.1145/1146238.1146266 (cited on page 81).

[DEG+11] Juan José Dominguez-Jiménez, Antonia Estero-Botaro, Antonio Garcia-Dominguez, and Inmaculada Medina-Bulo. Evolutionary mutation testing. *Inf. Softw. Technol.*, 53(10):1108–1123, 2011. DOI: 10.1016/j.infsof.2011.03.008. URL: https://doi.org/10.1016/j.infsof.2011.03.008 (cited on page 34).

[DHK+15] Frédéric Dadeau, Pierre-Cyrille Héam, Rafik Kheddam, Ghazi Maatoug, and Michaël Rusinowitch. Model-based mutation testing from security protocols in HLPSL. *Softw. Test. Verification Reliab.*, 25(5-7):684–711, 2015. DOI: 10.1002/stvr.1531. URL: https://doi.org/10.1002/stvr.1531 (cited on page 37).

[DLR12] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4–5):531–577, August 2012. ISSN: 1382-3256. DOI: 10.1007/s10664-011-9173-9. URL: https://doi.org/10.1007/s10664-011-9173-9 (cited on page 39).

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136. URL: https://doi.org/10.1109/C-M.1978.218136 (cited on pages 33, 101).

[Dom12] Pedro M. Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, 2012. DOI: 10.1145/2347736.2347755. URL: https://doi.org/10.1145/2347736.2347755 (cited on page 19).

[dOP17] Francisco Bento de Oliveira Neto and Henrique da Silva Pereira. Applying mutation testing for web application security. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 395–400. IEEE, 2017 (cited on page 11).

[DP22] Renzo Degiovanni and Mike Papadakis. μbert: mutation testing using pre-trained language models. In *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022, Valencia, Spain, April 4-13, 2022*, pages 160–169. IEEE, 2022. DOI: 10.1109/ICSTW55395.2022.00039. URL: https:

//doi.org/10.1109/ICSTW55395.2022.00039 (cited on pages 11, 36, 101, 108).

[DPP+16] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 655–666. ACM, 2016. DOI: 10.1145/2884781.2884821. URL: https://doi.org/10.1145/2884781.2884821 (cited on page 7).

[dPX+06] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 59–68. IEEE Computer Society, 2006. DOI: 10.1109/ASE.2006.13. URL: https://doi.org/10.1109/ASE.2006.13 (cited on page 81).

[DTP+18] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*:1–1, 2018 (cited on page 133).

[Elb11] Sebastian Elbaum. Mutation testing in the real world. *IEEE Software*, 28(1):67–69, 2011 (cited on page 11).

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. DOI: 10.1016/j.scico.2007.01.015. URL: https://doi.org/10.1016/j.scico.2007.01.015 (cited on page 35).

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *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. DOI: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179 (cited on pages 3, 90, 94).

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: https://doi.org/10.18653/v1/2020.findings-emnlp.139 (cited on pages 101, 150).

[FLW+20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM, 2020. DOI: 10.1145/3379597.3387501. URL: https://doi.org/10.1145/3379597.3387501 (cited on page 36).

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, Västeras, Sweden, 2014. DOI: 10.1145/2642937.2642982. URL: https://hal.archives-ouvertes.fr/hal-01054552. update for oadoi on Nov 02 2018 (cited on page 129).

[Fra00] Phyllis G. Frankl. Assessing and enhancing software testing effectiveness. *ACM SIGSOFT Softw. Eng. Notes*, 25(1):50–51, 2000. DOI: 10.1145/340855.340888. URL: https://doi.org/10.1145/340855.340888 (cited on page 3).

[FZ10] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, Trento, Italy. ACM, 2010. ISBN: 978-1-60558-823-0. DOI: 10.1145/1831708.1831728. URL: http://doi.acm.org/10.1145/1831708.1831728 (cited on pages 17, 18, 55, 71).

[FZ12] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012. DOI: 10.1109/TSE.2011.93. URL: https://doi.org/10.1109/TSE.2011.93 (cited on page 22).

[GAA+16] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the limits of mutation reduction strategies. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 511–522. ACM, 2016. DOI: 10.1145/2884781.2884787. URL: https://doi.org/10.1145/2884781.2884787 (cited on page 89).

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN: 978-0-262-03561-3. URL: http://www.deeplearningbook.org/ (cited on pages 7, 19).

[GBH+17] Nicolas E. Gold, David W. Binkley, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. Generalized observational slicing for tree-represented modelling languages. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 547–558. ACM, 2017. DOI: 10.1145/3106237.3106304. URL: https://doi.org/10.1145/3106237.3106304 (cited on page 109).

[GDJ+22] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: how to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022. DOI: 10.1007/s10664-022-10197-4. URL: https://doi.org/10.1007/s10664-022-10197-4 (cited on pages 12, 21, 36, 87, 94, 106, 107, 112, iii).

[GDM+23] Aayush Garg, Renzo Degiovanni, Facundo Molina, Mike Papadakis, Nazareno Aguirre, Maxime Cordy, and Yves Le Traon. Assertion inferring mutants. *CoRR*, abs/2301.12284, 2023. DOI: 10.48550/arXiv.2301.12284. arXiv: 2301.12284. URL: https://doi.org/10.48550/arXiv.2301.12284 (cited on pages 10, iii).

[GDP+23] Aayush Garg, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Vulnerability mimicking mutants. *CoRR*, abs/2303.04247, 2023. DOI: 10.48550/arXiv.2303.04247. arXiv: 2303.04247. URL: https://doi.org/10.48550/arXiv.2303.04247 (cited on pages 11, 87, iii).

[GNF+12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012. DOI: 10.1109/TSE.

2011.104. URL: https://doi.org/10.1109/TSE.2011.104 (cited on page 109).

[GOD+22] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: static subsuming mutant selection. *IEEE Transactions on Software Engineering*:1–1, 2022. DOI: 10.1109/TSE.2022.3140510 (cited on pages 7, 39, 81, 86–89, 94, 95, 101, 105–107, 112, 130, iii).

[GOM98] Anup K. Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 104–114. IEEE Computer Society, 1998. DOI: 10.1109/SECPRI.1998.674827. URL: https://doi.org/10.1109/SECPRI.1998.674827 (cited on page 37).

[GPK+17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 1345–1351, San Francisco, California, USA. AAAI Press, 2017 (cited on page 39).

[GRP+10] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010. DOI: 10.1145/1831708.1831712. URL: https://doi.org/10.1145/1831708.1831712 (cited on page 22).

[GZY+17] Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. Mutant reduction based on dominance relation for weak mutation testing. *Information & Software Technology*, 81:82–96, 2017. DOI: 10.1016/j.infsof.2016.05.001. URL: https://doi.org/10.1016/j.infsof.2016.05.001 (cited on pages 33, 88).

[GZZ+16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, Seattle, WA, USA. Association for Computing Machinery, 2016. ISBN: 9781450342186. DOI: 10.1145/2950290.2950334. URL: https://doi.org/10.1145/2950290.2950334 (cited on page 39).

[HBB+12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012 (cited on pages 39, 131).

[HLZ12] RM Hierons, M Li, and H Zhu. Using branch coverage to refine mutation testing. *Journal of Systems and Software*, 85(7):1498–1513, 2012 (cited on page 4).

[HLZ16] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1606–1612, New York, New York, USA. AAAI Press, 2016. ISBN: 9781577357704 (cited on page 39).

[HNA+17] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017. arXiv: 1712.00409. URL: http://arxiv.org/abs/1712.00409 (cited on page 150).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (cited on pages 130, 133).

[HSF+19] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. Comparing mutation testing at the levels of source code and compiler intermediate representation. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 114–124. IEEE, 2019. DOI: 10.1109/ICST.2019.00021. URL: https://doi.org/10.1109/ICST.2019.00021 (cited on pages 17, 72).

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. ISBN: 9780387848570. DOI: 10.1007/978-0-387-84858-7. URL: https://doi.org/10.1007/978-0-387-84858-7 (cited on page 21).

[IEE17] IEEE. The importance of addressing vulnerabilities in medical devices. *IEEE Standards Association*, 2017. URL: https://spectrum.ieee.org/the-importance-of-addressing-vulnerabilities-in-medical-devices (cited on page 25).

[Ins20] Ponemon Institute. The Cost of Software Vulnerabilities in 2020. Technical report, 2020. URL: https://www.ibm.com/downloads/cas/5N5K5AJZ (cited on pages 10, 24).

[JH09] Yue Jia and Mark Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, 2009. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.016. URL: https://doi.org/10.1016/j.infsof.2009.04.016. Source Code Analysis and Manipulation, SCAM 2008 (cited on pages 18, 35, 43, 44, 71, 102, 109).

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011. DOI: 10.1109/TSE.2010.62. URL: https://doi.org/10.1109/TSE.2010.62 (cited on pages 4, 6, 7).

[JPL16] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112. IEEE, 2016 (cited on page 142).

[JPT18] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Enabling the continous analysis of security vulnerabilities with vuldata7. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2018, Madrid, Spain, September 23-24, 2018*, 2018 (cited on pages 128, 140).

[JRP+19] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 695–705, Tallinn, Estonia. Association for Computing Machinery, 2019. ISBN: 9781450355728. DOI: 10.1145/3338906.3338941. URL: https://doi.org/10.1145/3338906.3338941 (cited on pages 14, 27, 88, 107, 121, 123, 128–131, 138, 141–143, 149).

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: convolutional architecture for fast feature embedding. In Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu, editors, *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 675–678. ACM, 2014. DOI: 10.1145/2647868.

2654889. URL: https://doi.org/10.1145/2647868.2654889 (cited on page 150).

[JSK11] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 612–615. IEEE Computer Society, 2011. DOI: 10.1109/ASE.2011.6100138. URL: https://doi.org/10.1109/ASE.2011.6100138 (cited on pages 81, 90).

[JVS+16] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016. arXiv: 1602.02410. URL: http://arxiv.org/abs/1602.02410 (cited on page 150).

[KAD+14] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185. IEEE Computer Society, 2014. DOI: 10.1109/ICSTW.2014.20. URL: https://doi.org/10.1109/ICSTW.2014.20 (cited on pages 18, 44, 53, 55, 59).

[KAO+16] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 571–582, Seattle, WA, USA. Association for Computing Machinery, 2016. ISBN: 9781450342186. DOI: 10.1145/2950290.2950322. URL: https://doi.org/10.1145/2950290.2950322 (cited on pages 7, 17, 18, 33, 43, 53, 71, 72, 88, 89, 95, 102).

[KB13] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1700–1709. ACL, 2013. URL: https://aclanthology.org/D13-1176/ (cited on pages 82, 87, 106).

[KBV09] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. DOI: 10.1109/MC.2009.263. URL: https://doi.org/10.1109/MC.2009.263 (cited on page 20).

[KFZ+19] Marinos Kintis, Minas Fragkoulis, Dimitra Zoni, Aphrodite Tsalgatidou, and Georgios Gousios. Mutation testing at scale: an empirical study of pit on the apache software suite. *Empirical Software Engineering*, 24(2):680–722, 2019 (cited on page 4).

[Kon95] Igor Kononenko. On biases in estimating multi-valued attributes. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1034–1040, Montreal, Quebec, Canada. Morgan Kaufmann Publishers Inc., 1995. ISBN: 1558603638 (cited on page 132).

[KPJ+18] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Trans. Software Eng.*, 44(4):308–333, 2018. DOI: 10.1109/TSE.2017.2684805. URL: https://doi.org/10.1109/TSE.2017.2684805 (cited on pages 17, 72).

[KPM10] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: a collateral experiment. In Jun Han and Tran Dan Thu, editors, *2010 Asia Pacific Software Engineering Conference*, pages 300–309. IEEE Computer Society, 2010. DOI: 10.1109/APSEC.2010.42. URL: https://doi.org/10.1109/APSEC.2010.42 (cited on pages 7, 18, 19, 43, 102).

[KPM15] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Test. Verification Reliab.*, 25(5-7):508–535, 2015. DOI: 10.1002/stvr.1529. URL: https://doi.org/10.1002/stvr.1529 (cited on page 72).

[KPP+16] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 147–156. IEEE Computer Society, 2016. DOI: 10.1109/SCAM.2016.28. URL: https://doi.org/10.1109/SCAM.2016.28 (cited on page 72).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (cited on page 20).

[LB12] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 133–146. ACM, 2012. DOI: 10.1145/2384616.2384626. URL: https://doi.org/10.1145/2384616.2384626 (cited on page 81).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nat.*, 521(7553):436–444, 2015. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539 (cited on page 19).

[LCC+05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005. DOI: 10.1016/j.scico.2004.05.015. URL: https://doi.org/10.1016/j.scico.2004.05.015 (cited on page 81).

[LDP+17] Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, and Patrick Heymans. Towards security-aware mutation testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 97–102. IEEE Computer Society, 2017. DOI: 10.1109/ICSTW.2017.24. URL: https://doi.org/10.1109/ICSTW.2017.24 (cited on pages 11, 37, 101).

[Lip71] Richard J Lipton. Fault diagnosis of computer programs. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 302–308. Morgan Kaufmann Publishers Inc., 1971 (cited on page 4).

[LLS+21] Zhuang Liu, Wayne Lin, Ya Shi, and Jun Zhao. A robustly optimized bert pre-training approach with post-training. In *China National Conference on Chinese Computational Linguistics*, pages 471–484. Springer, 2021 (cited on page 150).

[LLZ+21] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, Virtual Event, Australia. Association for Computing Machinery, 2021. ISBN: 9781450367684. DOI: 10.1145/3324884.3416591. URL: https://doi.org/10.1145/3324884.3416591 (cited on page 101).

[LZT+19] Yang Li, Ruicheng Zhang, Changqing Tian, and Xihong Chen. An effective approach to test case design with boundary value analysis and equivalence partitioning. *Journal of Systems and Software*, 156:48–63, 2019 (cited on page 3).

[LZX+18] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_03A-2%5C_Li%5C_paper.pdf (cited on page 38).

[MAP+15] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/. Software available from tensorflow.org (cited on pages 56, 130).

[Mat75] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975. ISSN: 0005-2795. DOI: https://doi.org/10.1016/0005-2795(75)90109-9.

URL: http://www.sciencedirect.com/science/article/pii/0005279575901099 (cited on pages 22, 44, 57).

[MBK+18] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loic Correnson. Time to clean your test objectives. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 456–467. ACM, 2018. DOI: 10.1145/3180155.3180191. URL: https://doi.org/10.1145/3180155.3180191 (cited on pages 33, 88).

[MCP+21] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. Mudelta: delta-oriented mutation testing at commit time. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 897–909. IEEE, 2021. DOI: 10.1109/ICSE43902.2021.00086. URL: https://doi.org/10.1109/ICSE43902.2021.00086 (cited on pages 33, 94).

[MdA22] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022. DOI: 10.1145/3510003.3510120. URL: https://doi.org/10.1145/3510003.3510120 (cited on pages 9, 23, 35, 81–83, 86, 88–90, 94, 95, 101, 108).

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. ISBN: 0-13-629155-4. URL: http://www.eiffel.com/doc/oosc/page.html (cited on page 22).

[MFB+08] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 537–552. Springer, 2008. DOI: 10.1007/978-3-540-87875-9_38. URL: https://doi.org/10.1007/978-3-540-87875-9%5C_38 (cited on page 101).

[MGR+18] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens Van Der Maaten. Exploring the limits of weakly supervised pretraining. In

Proceedings of the European conference on computer vision (ECCV), pages 181–196, 2018 (cited on page 150).

[MHM+15] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, pages 1–9, Urbana, Illinois. Association for Computing Machinery, 2015. ISBN: 9781450333764. DOI: 10.1145/2746194.2746198. URL: https://doi.org/10.1145/2746194.2746198 (cited on pages 27, 121, 124).

[MKK17] Mohamed Wiem Mkaouer, Marouane Kessentini, and Ahmed Hadj Kacem. On the use of mutation testing for assessing and improving the test effectiveness of safety-critical software systems. *Journal of Systems and Software*, 131:121–135, 2017 (cited on page 4).

[MLB07] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Mutation analysis for security tests qualification. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 233–242, 2007. DOI: 10.1109/TAIC.PART.2007.21 (cited on page 37).

[MMP15] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Guided mutation testing for javascript web applications. *IEEE Trans. Software Eng.*, 41(5):429–444, 2015. DOI: 10.1109/TSE.2014.2371458. URL: https://doi.org/10.1109/TSE.2014.2371458 (cited on page 33).

[MPA+21] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: an evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021. DOI: 10.1109/ICSE43902.2021.00112. URL: https://doi.org/10.1109/ICSE43902.2021.00112 (cited on pages 9, 23, 35, 81, 82, 86, 89, 94, 95, 101).

[MS16] Sara Moshtari and Ashkan Sami. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In Sascha Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1415–1421. ACM, 2016. DOI: 10.1145/2851613.2851777. URL: https://doi.org/10.1145/2851613.2851777 (cited on page 27).

[MSC+21] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. *CoRR*, abs/2102.02017, 2021. arXiv: 2102.02017. URL: https://arxiv.org/abs/2102.02017 (cited on page 150).

[MZS+22] Wei Ma, Mengjie Zhao, Ezekiel O. Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. Graphcode2vec: generic code embedding via lexical and program dependence analyses. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 524–536. ACM, 2022. DOI: 10.1145/3524842.3528456. URL: https://doi.org/10.1145/3524842.3528456 (cited on page 33).

[NWH+15] Jay Nanavati, Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Mutation testing of memory-related operators. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015. DOI: 10.1109/ICSTW.2015.7107449. URL: https://doi.org/10.1109/ICSTW.2015.7107449 (cited on pages 37, 101).

[NZH+07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, Alexandria, Virginia, USA. Association for Computing Machinery, 2007. ISBN: 9781595937032. DOI: 10.1145/1315245.1315311. URL: https://doi.org/10.1145/1315245.1315311 (cited on pages 38, 121, 132, 142).

[OBL+14] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014 (cited on page 150).

[OCH57] Akira OCHIAI. Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii. *NIPPON SUISAN GAKKAISHI*, 22(9):526–530, 1957. DOI: 10.2331/suisan.22.526 (cited on pages 108, 109).

[OGK+21] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies. *CoRR*, abs/2112.14508, 2021. arXiv: 2112.14508. URL: https://arxiv.org/abs/2112.14508 (cited on page iii).

[OL17] Jeff Offutt and Ye Wu Lee. Mutation: past, present and future. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, volume 43 of number 3, pages 661–670. IEEE, IEEE, 2017. DOI: 10.1109/ICSE.2017.67 (cited on pages 6, 150).

[OLR+96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996. DOI: 10.1145/227607.227610. URL: https://doi.org/10.1145/227607.227610 (cited on pages 6, 17, 33, 43, 102).

[oST19] National Institute of Standards and Technology. National Vulnerability Database. Technical report, 2019. URL: https://nvd.nist.gov/ (cited on pages 10, 25).

[PBC+18] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen, Cesare Alippi, and Gianluca Bontempi. Credit card fraud detection: A realistic modeling and a novel learning strategy. *IEEE Trans. Neural Networks Learn. Syst.*, 29(8):3784–3797, 2018. DOI: 10.1109/TNNLS.2017.2736643. URL: https://doi.org/10.1109/TNNLS.2017.2736643 (cited on page 20).

[PCT18] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 32–39. IEEE Computer Society, 2018. DOI: 10.1109/ICSTW.2018.00025. URL: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00025 (cited on pages 7, 17, 18, 43, 71).

[PDT14] Mike Papadakis, Márcio Eduardo Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.*, 95:298–319, 2014. DOI: 10.1016/j.scico.2014.05.012. URL: https://doi.org/10.1016/j.scico.2014.05.012 (cited on page 33).

[PHH+16] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on*

Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pages 354–365. ACM, 2016. DOI: 10.1145/2931037.2931040. URL: https://doi.org/10.1145/2931037.2931040 (cited on pages 7, 18, 19, 43, 55, 59, 71, 72, 88, 95, 109).

[PHT14] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: going beyond combinatorial interaction testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 1–10. IEEE Computer Society, 2014. DOI: 10.1109/ICST.2014.11. URL: https://doi.org/10.1109/ICST.2014.11 (cited on page 4).

[PI18a] Goran Petrovic and Marko Ivankovic. State of mutation testing at google. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 201–210. IEEE, 2018. DOI: 10.1145/3183519.3183525 (cited on pages 4, 5).

[PI18b] Goran Petrovic and Marko Ivankovic. State of mutation testing at google. In *40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2018, May 27 - 3 June 2018, Gothenburg, Sweden*, 2018 (cited on page 33).

[PJH+15] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 936–946. IEEE Computer Society, 2015. DOI: 10.1109/ICSE.2015.103. URL: https://doi.org/10.1109/ICSE.2015.103 (cited on page 17).

[PKL+09] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 87–102. ACM, 2009. DOI: 10.1145/1629575.1629585. URL: https://doi.org/10.1145/1629575.1629585 (cited on page 81).

[PKZ+19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: an analysis and survey. *Adv. Comput.*, 112:275–378, 2019. DOI: 10.1016/bs.adcom.2018.03.015. URL: https://doi.org/10.1016/bs.adcom.2018.03.015 (cited on pages 5–7, 17, 22, 43, 72, 81, 102).

[PLE+07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007. DOI: 10.1109/ICSE.2007.37. URL: https://doi.org/10.1109/ICSE.2007.37 (cited on pages 81, 90, 94).

[PLS+19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 329–340. ACM, 2019. DOI: 10.1145/3293882.3330576. URL: https://doi.org/10.1145/3293882.3330576 (cited on page 3).

[PM04] B. Potter and G. McGraw. Software security testing. *IEEE Security Privacy*, 2(5):81–85, 2004 (cited on page 121).

[PM10a] Mike Papadakis and Nicos Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 90–99. IEEE Computer Society, 2010. DOI: 10.1109/ICSTW.2010.50. URL: https://doi.org/10.1109/ICSTW.2010.50 (cited on page 33).

[PM10b] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 121–130. IEEE Computer Society, 2010. DOI: 10.1109/ISSRE.2010.38. URL: https://doi.org/10.1109/ISSRE.2010.38 (cited on pages 7, 18).

[PM11] Mike Papadakis and Nicos Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Softw. Qual. J.*, 19(4):691–723, 2011. DOI: 10.1007/s11219-011-9142-y. URL: https://doi.org/10.1007/s11219-011-9142-y (cited on page 7).

[PMD+20] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 492–502, Seoul, Republic of Korea. Association for Computing Machinery, 2020. ISBN: 9781450375177. DOI: 10.1145/3379597.3387482. URL: https://doi.org/10.1145/3379597.3387482 (cited on pages 88, 107).

[PP21] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE 2021*, pages 906–918. ACM, 2021. DOI: 10.1145/3468264.3468623. URL: https://doi.org/10.1145/3468264.3468623 (cited on page 101).

[PSY+18] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 537–548. ACM, 2018. DOI: 10.1145/3180155.3180183. URL: https://doi.org/10.1145/3180155.3180183 (cited on pages 43, 109).

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015 (cited on page 150).

[Rei99] Stuart Reid. Software fault injection: inoculating programs against errors, by jeffrey voas and gary mcgraw, wiley, 1998 (book review). *Softw. Test. Verification Reliab.*, 9(1):75–76, 1999. DOI: 10.1002/(SICI)1099-1689(199903)9:1\<75::AID-STVR174\>3.0.CO;2-T. URL: https://doi.org/10.1002/(SICI)1099-1689(199903)9:1%5C%3C75::AID-STVR174%5C%3E3.0.CO;2-T (cited on page 101).

[Row97] J. K. Rowling. *Harry Potter and the Philosopher's Stone*, volume 1. Bloomsbury Publishing, London, 1st edition, June 1997. ISBN: 978-0747532699 (cited on page 82).

[RSR+19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text

transformer. *CoRR*, abs/1910.10683, 2019. arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683 (cited on page 150).

[RW22] Cedric Richter and Heike Wehrheim. Learning realistic mutations: bug creation for neural bug detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173, 2022. DOI: 10.1109/ICST53961.2022.00027 (cited on pages 36, 101).

[SBH14a] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: the use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014 (cited on pages 22, 44).

[SBH14b] Martin J. Shepperd, David Bowes, and Tracy Hall. Researcher bias: the use of machine learning in software defect prediction. *IEEE Trans. Software Eng.*, 40(6):603–616, 2014. DOI: 10.1109/TSE.2014.2322358. URL: https://doi.org/10.1109/TSE.2014.2322358 (cited on page 82).

[SMM+17] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: the sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=B1ckMDqlg (cited on page 150).

[SMW+11] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.81. URL: https://doi.org/10.1109/TSE.2010.81 (cited on pages 38, 121, 131, 132, 142).

[SNL19] Apeksha Shewalkar, Deepika Nyavanandi, and Simone Ludwig. Performance evaluation of deep neural networks applied to speech recognition: rnn, lstm and gru. *Journal of Artificial Intelligence and Soft Computing Research*, 9(4):235–245, October 2019. DOI: 10.2478/jaiscr-2019-0006. URL: https://doi.org/10.2478/jaiscr-2019-0006 (cited on pages 56, 87, 107, 112, 130).

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, 2014. arXiv: 1409.3215 [cs.CL]. URL: https://proceedings.

neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html (cited on pages 20, 21, 87, 107).

[SW08] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 315–317, Kaiserslautern, Germany. Association for Computing Machinery, 2008. ISBN: 9781595939715. DOI: 10.1145/1414004.1414065. URL: https://doi.org/10.1145/1414004.1414065 (cited on page 131).

[SW09] Ben H Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.031 (cited on pages 4, 5).

[SW13] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? en. *Empirical Software Engineering*, 18(1):25–59, February 2013. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-011-9190-8. (Visited on 12/17/2015) (cited on pages 26, 38, 135, 136).

[SWH+14] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014 (cited on pages 26, 38, 132).

[SXL+17] Chang-ai Sun, Feifei Xue, Huai Liu, and Xiangyu Zhang. A path-aware approach to mutant reduction in mutation testing. *Information & Software Technology*, 81:65–81, 2017. DOI: 10.1016/j.infsof.2016.02.006. URL: https://doi.org/10.1016/j.infsof.2016.02.006 (cited on page 33).

[SZ08] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, pages 979–984. IEEE Computer Society, 2008. DOI: 10.1109/COMPSAC.2008.123. URL: https://doi.org/10.1109/COMPSAC.2008.123 (cited on page 37).

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in*

Computer Science, pages 134–153. Springer, 2008. DOI: 10.1007/978-3-540-79124-9_10. URL: https://doi.org/10.1007/978-3-540-79124-9%5C_10 (cited on page 81).

[TDS+22] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *IEEE/ACM AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*, pages 54–64. ACM/IEEE, 2022. DOI: 10.1145/3524481.3527220. URL: https://doi.org/10.1145/3524481.3527220 (cited on page 101).

[TJT+20] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1178–1189. ACM, 2020. DOI: 10.1145/3368089.3409758. URL: https://doi.org/10.1145/3368089.3409758 (cited on pages 9, 23, 35, 81, 82, 86, 89, 94, 101).

[TKW+20] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. Deepmutation: a neural mutation tool. In *ICSE: Companion Proceedings*, ICSE '20, pages 29–32, Seoul, South Korea. ACM, 2020. ISBN: 9781450371223. DOI: 10.1145/3377812.3382146. URL: https://doi.org/10.1145/3377812.3382146 (cited on page 101).

[TMR+18] Gongbo Tang, Mathias Müller, Annette Rios, and Rico Sennrich. Why self-attention? A targeted evaluation of neural machine translation architectures. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 4263–4272. Association for Computational Linguistics, 2018. DOI: 10.18653/v1/d18-1458. URL: https://doi.org/10.18653/v1/d18-1458 (cited on page 68).

[TPW+19] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 25–36. IEEE / ACM, 2019. DOI:

10.1109/ICSE.2019.00021. URL: https://doi.org/10.1109/ICSE.2019.00021 (cited on pages 86, 87, 94, 106).

[TW20] Christopher Theisen and Laurie A. Williams. Better together: comparing vulnerability prediction models. *Inf. Softw. Technol.*, 119, 2020. DOI: 10.1016/j.infsof.2019.106204. URL: https://doi.org/10.1016/j.infsof.2019.106204 (cited on pages 26, 38, 131, 132).

[TWB+19a] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*:301–312, September 2019. DOI: 10.1109/icsme.2019.00046. URL: http://dx.doi.org/10.1109/ICSME.2019.00046 (cited on pages 20, 21, 34, 39, 56, 71, 87, 106, 125, 130).

[TWB+19b] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019. DOI: 10.1145/3340544. URL: https://doi.org/10.1145/3340544 (cited on pages 20, 39, 56, 71, 86, 106, 130).

[TZY+15] Yaming Tang, Fei Zhao, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. Predicting vulnerable components via text mining or software metrics? an effort-aware perspective. In *QRS*, pages 27–36. IEEE, 2015. ISBN: 978-1-4673-7989-2. URL: http://dblp.uni-trier.de/db/conf/qrs/qrs2015.html#TangZYLZX15 (cited on page 121).

[VD00] András Vargha and Harold D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. ISSN: 10769986, 19351054. URL: http://www.jstor.org/stable/1165329 (cited on pages 61, 62, 135).

[Ver20] Veracode. State of Software Security. Technical report, 2020. URL: https://info.veracode.com/hubfs/2020-State-of-Software-Security-Report.pdf (cited on pages 10, 24).

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS 2017, December, 2017*, pages 5998–6008, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/

3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (cited on pages 107, 112).

[Wil45] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN: 00994987. URL: http://www.jstor.org/stable/3001968 (cited on page 135).

[WLT16] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, Austin, Texas. Association for Computing Machinery, 2016. ISBN: 9781450339001. DOI: 10.1145/2884781.2884804. URL: https://doi.org/10.1145/2884781.2884804 (cited on page 39).

[WQB+22] Matthew Watson, Chen Qian, Jonathan Bischof, François Chollet, et al. Kerasnlp. https://github.com/keras-team/keras-nlp, 2022 (cited on page 107).

[WTM+20] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1398–1409. ACM, 2020. DOI: 10.1145/3377811.3380429. URL: https://doi.org/10.1145/3377811.3380429 (cited on page 81).

[WTV+16] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016 (cited on page 39).

[WXS+17] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. Faster mutation analysis via equivalence modulo states. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 295–306. ACM, 2017. DOI: 10.1145/3092703.3092714. URL: https://doi.org/10.1145/3092703.3092714 (cited on page 72).

[YLX+15] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015 (cited on page 39).

[YS20] Jingxiu Yao and Martin J. Shepperd. Assessing software defection prediction performance: why using the matthews correlation coefficient matters. In Jingyue Li, Letizia Jaccheri, Torgeir Dingsøyr, and Ruzanna Chitchyan, editors, *EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020*, pages 120–129. ACM, 2020. DOI: 10.1145/3383219.3383232. URL: https://doi.org/10.1145/3383219.3383232 (cited on page 82).

[ZGM+13] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: better together. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 92–102. IEEE, 2013. DOI: 10.1109/ASE.2013.6693070. URL: https://doi.org/10.1109/ASE.2013.6693070 (cited on pages 17, 33, 43, 102).

[ZHH+10] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 435–444. ACM, 2010. DOI: 10.1145/1806799.1806863. URL: https://doi.org/10.1145/1806799.1806863 (cited on page 89).

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. DOI: 10.1145/267580.267590. URL: https://doi.org/10.1145/267580.267590 (cited on pages 4, 6).

[ZLS+19] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019. arXiv: 1909.03496 [cs.SE] (cited on pages 132, 133, 142).

[ZMK13] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 235–245. ACM, 2013. DOI: 10.1145/2483760.2483782. URL: https://doi.org/10.1145/2483760.2483782 (cited on page 72).

[ZNG+09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, Amsterdam, The Netherlands. Association for Computing Machinery, 2009. ISBN: 9781605580012. DOI: 10.1145/1595696.1595713. URL: https://doi.org/10.1145/1595696.1595713 (cited on page 121).

[ZZH+19] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Trans. Software Eng.*, 45(9):898–918, 2019. DOI: 10.1109/TSE.2018.2809496. URL: https://doi.org/10.1109/TSE.2018.2809496 (cited on page 34).