UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2018-62
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defence held on 02/10/2018 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN *INFORMATIQUE*

by

Matthieu JIMENEZ
Born on 6th September 1991 in Champigny Sur Marne, (Val de Marne, France)

EVALUATING VULNERABILITY PREDICTION MODELS

Dissertation defence committee

Dr Yves Le Traon, dissertation supervisor
Professor, Université du Luxembourg

Dr Federica Sarro,
Associate Professor, University College London, London, United Kingdom

Dr Pierre Kelsen, Chairman
Professor, Université du Luxembourg

Dr Mike Papadakis
Research Scientist, Université du Luxembourg

Dr Xavier Blanc, Vice Chairman
Professor, Université de Bordeaux, Bordeaux, France

Abstract

Today almost every device depends on a piece of software. As a result, our life increasingly depends on some software form such as smartphone apps, laundry machines, web applications, computers, transportation and many others, all of which rely on software. Inevitably, this dependence raises the issue of software vulnerabilities and their possible impact on our lifestyle. Over the years, researchers and industrialists suggested several approaches to detect such issues and vulnerabilities. A particular popular branch of such approaches, usually called Vulnerability Prediction Modelling (VPM) techniques, leverage prediction modelling techniques that flag suspicious (likely vulnerable) code components. These techniques rely on source code features as indicators of vulnerabilities to build the prediction models. However, the emerging question is how effective such methods are and how they can be used in practice.

The present dissertation studies vulnerability prediction models and evaluates them on real and reliable playground. To this end, it suggests a toolset that automatically collects real vulnerable code instances, from major open source systems, suitable for applying VPM. These code instances are then used to analyze, replicate, compare and develop new VPMs. Specifically, the dissertation has 3 main axes:

The first regards the analysis of vulnerabilities. Indeed, to build VPMs accurately, numerous data are required. However, by their nature, vulnerabilities are scarce and the information about them is spread over different sources (NVD, Git, Bug Trackers). Thus, the suggested toolset (develops an automatic way to build a large dataset) enables the reliable and relevant analysis of VPMs.

The second axis focuses on the empirical comparison and analysis of existing Vulnerability Prediction Models. It thus develops and replicates existing VPMs. To this end, the thesis introduces a framework that builds, analyse and compares existing prediction models (using the already proposed sets of features) using the dataset developed on the first axis.

The third axis explores the use of cross-entropy (metric used by natural language processing) as a potential feature for developing new VPMs. Cross-entropy, usually referred to as the naturalness of code, is a recent approach that measures the repetitiveness of code (relying on statistical models). Using cross-entropy, the thesis investigates different ways of building and using VPMs.

Overall, this thesis provides a fully-fledge study on Vulnerability Prediction Models aiming at assessing and improving their performance.

Acknowledgments

Writing this dissertation, alike the whole PhD was a challenging life experience with its many ups but also sometimes downs. The outcomes presented all along the present document would probably not have been possible if it were not for the support and help of a few people for whom I will always be grateful.

First of all, I would like to thank Pr. Yves Le Traon that gave me the opportunity to pursue my PhD studies in his group and under his supervision. He has always been benevolent and supportive throughout these 4 years. I'm especially grateful for the opportunity to teach at the university that he offered to me, as this experience taught me a lot.

My special thanks go to my daily supervisor, Dr. Mike Papadakis, for his patience (especially regarding my English writing), advices and flawless guidance through this process. He taught me how to conduct research, write scientific papers, and was always there for me. I am very happy about the friendship we have built up during the years.

I am grateful to the members of my dissertation committee, Pr. Pierre Kelsen, Pr. Xavier Blanc and Dr. Federica Sarro, whom I had the chance to encounter and discuss with during my thesis, for their time to review my work and for providing interesting and precious feedback.

I would also like to express my warm thanks to all the people I had the chance to work with. In particular, I want to thank Dr. Thomas Hartmann and Dr. François Fouquet for their valuable technical inputs and friendship. I would also like to thank Dr. Maxime Cordy, who helped developing many of the presented contributions.

I would also like to thank all current and former members of the SerVal team, with a special thought for Ludovic Mouline and Mederic Hurier, for the plenty good coffee breaks, passionate talks, and the great times we had.

Finally, and more personally, I would like to express my deepest heartfelt thanks to my family and my friends. Starting by my parents Valérie and Laurent and grandparents Josette, Nicole, Gérard and José that supported me since the very beginning always pushing me further. I would like to thank Dr Cyril Cecchinel, my old friend, who started his PhD at the same time and with whom I shared the experience of the PhD student journey even though he managed to finish first. My final thanks going to my siblings, my brothers Louis and Rafael who contributed in ways they couldn't possibly understand, and my one and only Sister Lisa whose long discussion helped me get through this long journey and who brought into my life, a life-long companion, my well-named cat, Thesis.

The role they all played in this journey is more important than they can possibly know.

Contents

List of abbreviations and acronyms

APC Attack Prone Components. 24

ASA Automated Static Analyses. 11, 24, 25, 27, 29, 30, 76

ASAp Attack Surface Approximation. 20, 26, 27, 30

AST Abstract Syntaxic Tree. 33, 69, 80, 85, 122, 130, 131, 133–135, 137, 138, 148, 180

BN Bayesian Network. 21, 24, 29, 31

CART Classification and Regression Tree. 24, 31

CART Control Flow Graph. 69

CSV Comma Separated Value. 95

CVE Common Vulnerabilities and Exposures. 7, 40, 51, 52, 62

CVSS Common Vulnerability Scoring System. 8, 51, 60, 61, 65

CWE Common Weakness Enumeration. xiii, 8, 43, 51, 53, 58, 60–63

DPM Defect Prediction Model. 5, 10, 22, 23, 29, 177, 179

GED Graph Edit Distance. 60, 63, 70

LoC Lines of Codes. 22, 24, 29, 30, 69, 134, 135, 176

LR Logistic Regression. 22–24, 27–29, 31, 81, 104, 107, 114

MCC MacCabe Cyclomatic Complexity. 22, 41

ML Machine Learning. xiii, 9, 12, 20–22, 24, 25, 27, 29, 31, 91, 92, 95, 101, 102

NB Naïve Bayes. 21, 24, 25, 27, 29, 31

NLP Natural Language Processing. 16, 122

NN Neural Network. 21, 25, 31

NVD National Vulnerability Database. 7, 8, 34, 38–40, 43, 47, 50, 52, 93, 177

RF Random Forest. 13, 21, 24–27, 29, 31, 81, 103, 104, 106, 107, 114

RP Recursive Partitioning. 29, 31

SVM Suport Vector Machine. 20, 21, 25–29, 31, 80, 114

TB Tree Bagging. 29, 31

VCS Version Control System. 26, 27, 50

VPM Vulnerability Prediction Model. v–vii, xi, xiii, 5, 10–29, 31, 33, 34, 37, 39, 50, 58–62, 70, 71, 75–80, 82, 84, 86, 88, 90–96, 98, 100–102, 104, 106, 108, 110, 112, 114, 116, 117, 121, 122, 143, 145, 146, 148–150, 152, 153, 171, 176–179

List of figures

List of tables

List of algorithms and listings

Part I

Introduction and State of the Art

1

Introduction

This chapter starts by exposing the context of this dissertation, followed by an introduction to software vulnerabilities. The technical background related to vulnerability prediction modelling is then presented. Next the challenges related to the use, evaluation and comparison of those models are described. Finally, an overview of the contributions of the thesis and its structure is presented.

Contents

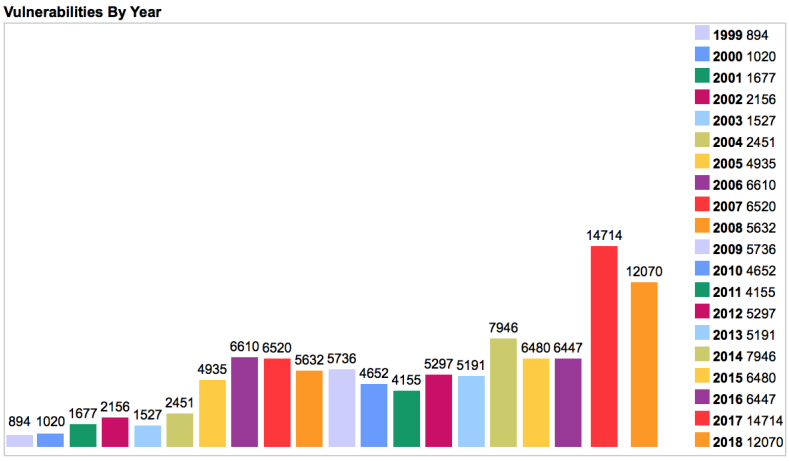


Figure 1.1: Vulnerabilities reported by year in the CVE Database since 1999

Source: https://www.cvedetails.com/browse-by-date.php (August 2018)

1.1 Context

Nowadays software can be found everywhere as it is not limited to the traditional computers sphere. Software can take various forms, like apps on smartphones, embedded code in laundry machines, web applications or even distributed software. As our dependence to software grows, the threat posed by vulnerabilities is becoming increasingly more important.

Vulnerabilities are described by the Mitre organization in charge of the Common Vulnerability and Exposure dictionary as a:

> *"weakness in the computational logic, e.g., code, found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability."* [6].

Recent history is full of examples, *e.g.,* Heartbleed [14], ShellShock [18], Apache Commons [3] where a single vulnerability in software installed in billions of devices had tremendous effects. Still, the threat level of a software vulnerability greatly depends on factors such as exploit complexity, the nature of the impact and the attack surface. Among the several types of vulnerabilities, code-based ones are responsible for the majority of the exploits [121] (software, data or commands that can lead to a security policy violation [106]).

The problem posed by vulnerabilities in software systems is not recent and can be traced back to the origin of computers. Guidelines and good practices of software development have indeed been suggested since the 70s [113]. Still, the globalization of software and the potential threats from ill-intentioned people brought more attention to the problem. This phenomenon can be observed in Figure 1.1 presenting the number of vulnerabilities publicly reported each year since 1999. An increasing trend can be observed with an explosion of reported vulnerabilities in the last two years.

To ensure secure development, industrialists adopt dedicated security life cycle processes which aim at identifying and fixing vulnerabilities [80]. Yet, due to business pressure and lack of experience in security, developers and companies tend to overlook these issues. To tackle these issues, various solutions have been suggested. Among those, some aim at uncovering vulnerabilities to warn software vendors as early as possible, while others aim at helping developers at performing security inspection efforts by pointing where to focus. In this dissertation, we are interested in this second category, which is vulnerability prediction. This prediction category of techniques involves methods aiming at guiding software security assurance (code reviews) and security testing. The same techniques can also be used to guide testing tools, such as static analysis tools, based on static rules such as Find Security Bugs [8] and the Fortify Static Code Analyser [10], and dynamic analysis tools, such as fuzzing tools.

Prediction techniques rely on data mining and machine learning in order to guess where vulnerabilities are. In other words, these techniques determine the software components that are likely to be vulnerable. Those models are commonly referred as Vulnerability Prediction Models and are the focus of this thesis. Yet there are many other prediction models, not considered, like models that predict the number of vulnerabilities remaining in a software system [24] ad models determining whether vulnerabilities are exploitable [39].

VPMs first appeared in the study of Neuhaus *et al.,* [133]. In this study, the authors suggested that includes and function calls of a file could be used as features by a classifier to build a model predicting the likelihood of a file to be vulnerable. This idea was previously introduced in a study of Schröter *et al.,* [162] but targeting defects instead of vulnerability. This last study falls in the category of Defect Prediction Models (DPMs) which is related to VPMs, as DPMs have the same goal but target likely to be defective elements. In fact most of the approaches that have been introduced in the context of VPMs can be linked to one introduced for DPMs. This link can be explained by the fact that vulnerabilities can be seen as a special kind of defect. Yet, VPMs face different problems than DPMs. First and quite (un)fortunately (depending on the point of view), vulnerabilities unlike defects are hard to find and fewer in numbers which hinder the creation of such models. As they rely on examples to determine the likelihood of an element to belong to one category instead of another. Second it requires quite a different identification process than defects. For instance, seeking for vulnerabilities requires an attacker's mindset [122] to understand and exploit the weaknesses of a system, whereas, defects are usually (easily) noticed by users or developers during the "normal" operation of the system while vulnerabilities pass unnoticed. Furthermore, vulnerabilities are critical [80, 121], while many bugs are not, *i.e.,* they are never fixed. Finally, most developers have a better understanding of how to identify and deal with defects than with vulnerabilities. All of this points make the creation of such models far more difficult as the number of examples available to train a model are limited and developers will have trouble to interpret the model outputs.

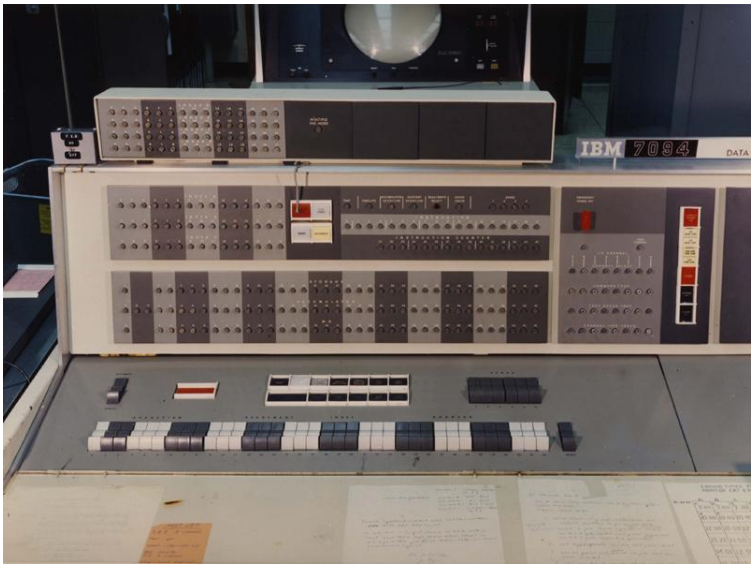


Figure 1.2: IBM 7094

Source: www.computer-history.info

1.2 Software Vulnerabilities

In this section, we detail the notion of software vulnerability and its related glossary.

Software vulnerabilities exist since the birth of computers and their history is tightly related to the one of hacking. First reference to hacking was made in 1963 when some people after analysing the frequencies used for the digit routing code of phones, managed to replicate the signal to make free phone calls [13]. Whereas, the first reported software vulnerability only occurred two years later, *i.e.,* in 1965. The vulnerability was in the text editor of the CTSS operating system running on IBM7090 (see 1.2). Back then text editors were designed to be used by one user at a time, on a given resource. Thus, when two system programmers tried to edit the same file simultaneously, the text editor swapped the welcome message file with the one containing all passwords, resulting in passwords being printed to every user when they were logging to the system.

In software security, such problem is considered as a vulnerability when one or more of the following principle is impacted: Confidentiality, Integrity or Availability. In this specific case, the password of all users being printed clearly corresponds to a breach in Confidentiality Since then, software vulnerabilities often hit the news especially in the past years where example such as the Heartbleed vulnerability in OpenSSL [14] that affected two third of the world servers are still fresh in the mind of people.

Yet, despite constant remainder, the definition of what is a vulnerability is still debated. Krsul in his PhD thesis [106] defined it back in 1998 as:

> *"An instance of an error in the specification, development or configuration of a software such that its execution can violate the security policy."*

Table 1.1: Terms used throughout the thesis (CVE, NVD, CWE and CVSS).

Term & Acronym	Explanation
Common Vulnerability Exposures (CVE)	Unique vulnerability identifier for publicly disclosed vulnerabilities.
National Vulnerability Database (NVD)	Database recording all vulnerabilities in the CVE list and enhancing it with information like CVSS and CWE.
Common Weakness Enumeration (CWE)	Unique vulnerability type identifier that links to a type list (used by NVD to categorize vulnerabilities).
Common Vulnerability Scoring System (CVSS)	Score (between 0 and 10) representing the severity of a vulnerability based on its impact, attack vector and complexity.

While Ozment [141], almost a decade later, suggested using the term "mistake" instead of "error" to better relate to the IEEE Standard Glossary of Software Terminology. Similarly Dowd *et al.,* [56] defined a software vulnerability as:

> *"Specific flaws or oversights in a piece of software that allows attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, take control of a computer system or program."*

In these three definitions, the terms "error", "mistake" and "flaw" are used to define a vulnerability, while they refer to the same global idea, they do not hold the same meaning. In this regard, Ghaffarian *et al.,* [68] suggested favouring the use of "fault" to refer to the cause of a vulnerability and suggested the following definition:

> *"A software vulnerability is an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy."*

Interestingly among those definitions, the one provided by Dowd *et al.,* is the only one clearly stating an impact on one of the security principle, *i.e.,* Confidentiality, Integrity and/or Availability, while the other refers to a violation of security policy. This is less significant as those impacts are commonly acknowledged and thus implied when referring to security policy violations.

Over the past two decades, the understanding and management of vulnerabilities were greatly improved. This is mainly due to the apparition of databases referencing publicly exposed vulnerabilities. Among those, the most famous one is the Common Vulnerability Identifier (CVE) List. The CVE system grants to each publicly disclosed vulnerability a unique identifier that can be used to retrieve information on it. The CVE List is the dictionary mapping all those identifiers to a brief description of their corresponding vulnerability and any relevant references to it.

While the CVE system is quite famous, it turns out that most people when mentioning it are in fact referring to the National Vulnerability Database (NVD). The National Vulnerability Database (NVD) is a database built upon the CVE List established by the National Institute of Standards and Technology (NIST) and the U.S. government in order to encourage secure software development, public disclosure and management of vulnerabilities. NVD enriches each CVE entry with information such as severity (named as CVSS) and type (named as CWE) of a vulnerability. Moreover data is continuously updated by the NVD staff [15].

Among the information found in the NVD, two are of particular interest in this manuscript, the Common Vulnerability Scoring System (CVSS) and the Common Weakness Enumeration (CWE). The Common Vulnerability Scoring System (CVSS) system captures the principal characteristics of a vulnerability and produces a numerical score ranging from 0 to 10 reflecting its severity. This score can be translated into qualitative representation, such as *low* (0.1-3.9), *medium* (4.0-6.9), *high* (7.0-8.9), and *critical* (9.0-10.0) to assist in the prioritization of vulnerability. The current version of the CVSS is the 3.0 which was released in June 2015[22]. The CWE is a categorization system for software weaknesses and vulnerabilities. It is driven by a community project with the goals of understanding flaws in software. It is used by the NVD to categorize vulnerabilities and is currently in version 3.1 [7].

In the remaining part of this manuscript, we rely on the NVD as a source of information and use the CVSS and CWE systems as means to measure the severity and category of a vulnerability. Table 1.1 summarizes those terms.

1.3 Background on Vulnerability Prediction

1.3.1 Vulnerability Discovery

Finding vulnerabilities manually is a daunting time consuming task requiring specific skills that most software vendors can't afford. Due to the criticality of the matter, a large amount of effort has been invested over the years by both academic and industrial actors to suggest and develop automatic vulnerability discovery techniques. Yet, finding a perfect approach that will detect vulnerabilities in a sound and complete way *i.e.,* detect all vulnerabilities (complete) without false alarms (sound), is impossible. Thus most of the current research is focussed on suggesting approaches that perform better than previous ones wrt. some criteria. Those methods can usually be classified into three categories, static analysis, dynamic analysis and hybrid analysis.

Static analysis: directly analyses the source code of the program without executing it. The analysis is performed using diverse abstraction of the program. This type of approach is likely to find a high number of vulnerabilities but has the default of often raising too many false alerts. Rule-based static analysis tools typically falls into this category. Another well-known type of approach falling in this category is "Tainted Data-flow Analysis" where the flow to sensitive program statements known as "sinks" of untrusted data input is tracked.

Dynamic analysis: monitors the behaviour of a program when executed with a specific set of inputs. As the number of possible input to a program is infinite, these approaches focused on providing the best possible set of input wrt. to criteria such as coverage. The drawback is the opposite problematic of static analysis approaches, fewer but real vulnerabilities will be reported. The most famous type of approach falling into this category is perhaps fuzz testing, where valid inputs are mutated and fed to the program to test it.

Hybrid analysis: combines static and dynamic analysis to obtain the best of the two worlds. Unfortunately, such techniques often suffer from the drawbacks of both. This combination can happen both ways, either using dynamic analysis as a way to prune the results of a static analysis or using static analysis to guide the dynamic analysis.

More recently, a fourth category of techniques, relying on the application of data mining and machine learning techniques to discover vulnerabilities, have appeared.

1.3.2 Leveraging Machine Learning and Data Mining techniques

Data mining consists in the extraction of knowledge from large amount of data. According to Han *et al.,* [72], it can be broken into in the following steps: *(1)* data extraction *(2)* data cleaning and integration *(3)* data selection and transformation *(4)* knowledge mining and *(5)* visualization and communication. Machine learning in this context refers to techniques building complex models that can be used, among other things, to make predictions. ML techniques can be divided in three types: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning: learns a function that maps an input to an output based on training set. The training set is composed of labelled example, where each example is composed of an input vector and a label, *i.e.,* the desired output. This type of algorithm is traditionally used for classification purpose.

Unsupervised learning: functions similarly except that the example from the training set are unlabelled. The algorithm has to identify patterns and structures in the training set to create its own group. It is usually used for clustering purposes.

Reinforcement learning: learns to take actions in a dynamic environment with a system of rewards and penalties in order to achieve certain goals. It has numerous applications ranging from robotics to recommendation systems.

In the context of vulnerability discovery, according to Ghaffarian *et al.,* [68], most of the approaches that use data mining and machine learning techniques can be categorized into three main categories: anomaly detection approaches, vulnerable code pattern recognition and vulnerability prediction models.

Anomaly detection: uses unsupervised learning approaches to automatically extract a model of normality or mine rule from the source code and detect vulnerabilities as deviant behaviour. A recent example is the work from Yamaguchi *et al.,* [194] that suggests an approach to automatically detect missing checks in code.

Vulnerable Code Pattern Recognition: uses mostly supervise learning to extract patterns from large samples of vulnerable code and then use pattern matching techniques to detect vulnerabilities. A good example is the work targeting SQL injection performed by Shar *et al.,* [164].

More examples of this two types of approaches are presented in Section 2.2.

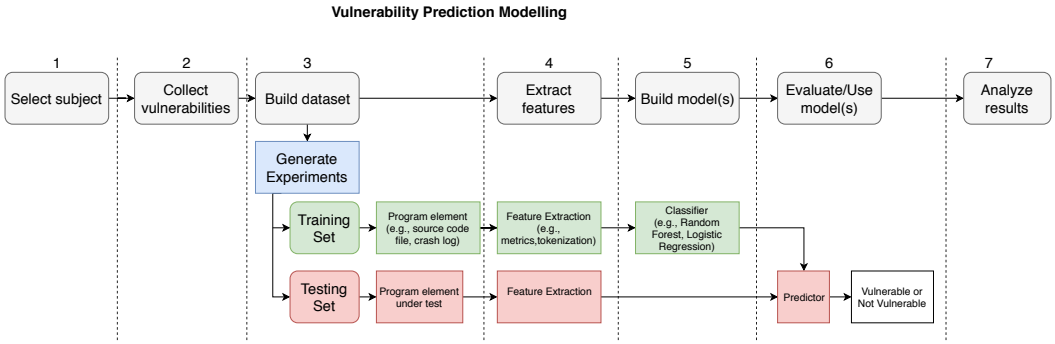


Figure 1.3: Vulnerability Prediction Modelling pipeline

Vulnerability Prediction Modelling: which is the topic of this dissertation, uses (mostly) supervised learning techniques to build prediction models based on a set of features and then used the model to evaluate the likelihood of a software component to be vulnerable. The study performed by Shin *et al.,* [167] is a good example of it.

1.3.3 Vulnerability Prediction Modelling

VPM is one of the numerous applications of prediction modelling. It aims at classifying software components as likely vulnerable with the ultimate goal to support testing and code review. Another instance of prediction modelling, as mentioned before, is Defect Prediction Modelling (DPM), which goal is to classify software components as likely buggy instead of vulnerable. More broadly, prediction modelling in the context of software engineering refers to the guidance that a model trained on historical data and/or part of a software project can offer by categorizing the software component according to specific properties of interest.

VPM, unlike previously introduced categories of approaches doesn't aim at directly finding vulnerabilities. Its goal is instead to help prioritize the security testing efforts. This makes VPM closer to other predictions modelling approaches (especially those regarding DPM) than of the other vulnerabilities discovery approaches.

Still, VPM faces some very specific problems starting by the data availability. The base of prediction modelling is to learn a model out of a large quantity of data, which in the case of vulnerabilities is problematic as these, by their nature, are few. Additionally, this raises the issue of unbalanced datasets as the vulnerability will always be in the minority in the training dataset.

Though, the pace of research in the area (presented in Chapter 2) has not decreased since the seminal work of Neuhaus *et al.,* [133] in 2007. Works in the area, aside of some minor differences, all follow a similar process, as demonstrated in Figure 1.3. It is clear when observing this process that VPM is a data mining application as each step presented in Figure 1.3 can be mapped to one of the steps from the definition of Han *et al.,* [72].

This process can be summarized as follows: *(1)* The subject(s), *i.e.,* the software project(s), on which the evaluation will be performed is chosen. *(2)* Vulnerabilities of the selected subject(s) are collected and *(3)* transformed into a dataset. *(3)* The dataset is split into training and testing set(s). *(4)* Features used by the evaluated approach are extracted for all components of the training set. *(5)* Those extracted features are used to train a chosen classifier. *(6)* The model obtained through the classifier training is then used to predict whether components from the testing set are likely to be vulnerable or not. *(7)* In case the correct label of the components present in the testing set is known, the overall performance of the model is measured. Those steps are further details in the next paragraphs.

1) subject selection is the first step of any VPM study. Software projects have different properties that are likely to affect the outcome of a study. Additionally, low quality and quantity of vulnerability data to train also have the potential to impact the performance of the models. Thus, a candidate subject should have a long and well-reported history of vulnerability. This explains that projects like Mozilla Firefox and the Linux Kernel have often been used to perform VPM studies as they are (1) security-sensitive (2) open source and (3) have a long history of reported vulnerability. Still, as this is not the case of a vast majority of projects, some researcher found a way around this issue that will be presented in the next paragraph.

2) vulnerability collection is perhaps the most critical part of any VPM studies and more generally any vulnerability related studies. Once the subject chosen, the next step is to collect the largest number of "real" vulnerabilities for this subject. This is usually done by collecting patches that fixed vulnerability, then considering as vulnerable every component that needed to be modification.

All these limitations on the subjects led some researchers to rely on a different ground truth to obtain vulnerability data. They suggested using the output of security oriented Automated Static Analyses (ASA) tools as indicators of vulnerability. Thus any component containing at least one warning returned by the tool would be considered as vulnerable. This idea was first suggested by Scandariato *et al.,* [161] on the premises of the result obtained by Walden *et al.,* [183] which shown a strong correlation between vulnerabilities and warning returned by ASA tools. While this enables a broader use of VPM, such ground truth remains questionable as the model obtained after training will likely return the likelihood of the ASA tools returning a warning instead of the likelihood to have a vulnerability. This is especially problematic as those tools tend to produce a lot of false alarms.

3) dataset creation transforms the set of collected vulnerabilities into a dataset containing both vulnerable and non-vulnerable examples. It can be considered as the first step really proper to VPM, as previous steps are common with most of vulnerabilities studies.

To create a dataset to train and/or evaluate a VPM on, the granularity level must first be chosen. In a sense the granularity level is the entity (code parts) that will be presented to developers. Evidently, different granularity levels offer different advantages [126]. For instance, the line level granularity can be direct but produce many false warnings and be too fine grained for the developers to identify the issue.

Most of the study adopt the file-component granularity level following the findings of Morrison *et al.,* [126], who found that the file (component) level was sufficient for Microsoft developers to work with.

Once the granularity selected, the source of the non-vulnerable components, *i.e.,* the type of dataset must be determined. Almost all studies opt for release based datasets. In release based datasets, one or several releases of the subject are downloaded and every component affected by a vulnerability that impacted a given release is considered as vulnerable for the said release while all the others are declared non-vulnerable. Another possibility suggested by Perl *et al.,* [150] while opting for commit-level granularity is to fill the dataset with random commits not related to vulnerabilities.

3) evaluation method is the part used to validate approaches. In the context of prediction modelling, it corresponds to the way to split the dataset into training and testing set(s).

The most common evaluation method is random splitting also called *leave one out* where a percentage of the dataset (usually 66%) is selected at random to be used as training and the remaining part, also named *holdout*, as testing. The percentage of each label is kept in both set at the same level as in the original dataset. The splitting can be repeated (usually 50-100 times) to ensure the generalization of the results.

Another common methodology is stratified k-fold cross validation, which can be considered as a specific application of random sampling. This method consists in splitting at random the dataset into k subsets with the same proportion between vulnerable and non-vulnerable components as the dataset (stratified). Each fold is then used as testing set while the others are used for training. This leads to the creation and evaluation of k models, where usually k equals to 5 or 10. This type of experiment can also be repeated several times (usually 10) to validate the results.

Another way when several releases of the subject are available is to train on one or several releases and evaluate on another one (usually more recent). This provides a more realistic setting of experimentation, *i.e.,* as a company would use it, but has the drawback of being sensitive to the problem of vulnerabilities spanning over multiple releases, *i.e.,* present in both training and testing set.

Finally, in case the dataset possesses some temporal property, it can be split according to some chosen time point, *i.e.,* everything before a given point in time is used as training while everything after is used as testing.

4) feature extraction is the central part of any VPM approach. When suggesting an approach, researchers are in fact suggesting a set of properties of the component to use as features for the ML algorithms. These properties can be take various forms and even use results from dynamic and/or static analysis.

The feature extraction phase then consists in computing for each component of the dataset those features. This phase can be realized before or after the split into training and testing set.

Table 1.2: Confusion Matrix in VPM

		Reality		
		Vulnerable	Non-vulnerable	Total
Prediction	Vulnerable	TP	FP	$TP + FP$
	Non-vulnerable	FN	TN	$FN + TN$
	Total	$TP + FN$	$FP + TN$	N

5) model building results in the creation of the model. First, a matrix M that use the feature elements of the feature list as columns and the component to train on values as rows is created. A last column is then added to M indicating the label of the component *i.e.,* whether the component is vulnerable or not. Then, this matrix is fed to a classifier that will return a model. The most common type of classifier found in VPM studies is by far Decision Trees. Decision Trees have the advantage of being human readable and quite easy to interpret. The most famous algorithm of this category probably being Random Forest (RF).

6) component classification is the phase where elements from the testing set are given to the model for prediction. For each element to test, a vector containing all the features values is first created then used as input for the model. The model will then return its prediction. Traditionally in VPM the classes are *vulnerable* or *non-vulnerable* which makes it a binary classification problem.

In case of evaluation, labels are known which makes it possible to evaluate the prediction. A model can either predicts that a component is vulnerable while it is not (False Positives - FPs), non-vulnerable while it is vulnerable (False Negatives - FNs), vulnerable while it is (True Positives - TPs) and non-vulnerable while it is (True Negatives - TNs). Note that as we are mainly interested in vulnerabilities they are considered as the positive case.

Once all predictions evaluated, the overall performance of the model is usually reported in a confusion matrix (see Table 1.2) that can then be used to measure the performance.

7) performance measurement is the final step of any empirical study on VPM and is used to evaluate the model.
The most common way to evaluate consists in computing precision and recall metrics from the confusion matrix. These are defined as:

Precision $= TP \ / \ (TP \ + \ FP)$
Recall $= TP \ / \ (TP \ + \ FN)$

Precision represents the probability of a component to be correctly classified as vulnerable. While, recall represents the probability of a vulnerable component to be correctly classified. Yet, it is hard to compare models using these metrics as they represent different aspects (often competitive) of the classification problem. A good general metric used in defect prediction studies [166, 38] and some of the VPM ones [93] is the Matthews Correlation Coefficient (MCC) [118].

MCC is related to chi-square statistic for a binary classification and is defined as:

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

As such, MCC returns a coefficient between -1 and +1. A value of 1 indicates a perfect prediction whereas a value of -1 a perfect inverse prediction. A coefficient of 0 indicates that the classifier performance is equivalent to random.

Another commonly encountered metric in VPM studies is the $F1$ score metric which is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

$$F_1 = \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Finally, as the output of VPM is supposed to guide the inspection effort, some authors [171] focused their effort on metrics called Component Inspection Ratio, noted as CI and Component Inspection Reduction, noted as CIR. CI is the ratio of components predicted as vulnerable (that is, the number of components to inspect) to the total number of components for the reported recall:

$$\text{CI} = (TP + FP) \,/\, (TP + FP + FN + TN)$$

Whereas CIR is the ratio of the reduced number of components to inspect by using the model compared to a random selection to achieve the same recall:

$$\text{CIR} = (Recall - CI) \,/\, Recall$$

1.4 Challenges

1.4.1 Overview

In today's connected world, software systems are pervasive. Thus, a vulnerability found in one of those systems has the potential to impact billions of users. Unfortunately, it is not possible for every software vendor to continuously test their entire codebase. First of all, uncovering vulnerabilities unlike bugs require specific skills and knowledge that developers don't necessarily possess. Then, the cost of searching the entire code base remains which represents millions of lines of code for the largest projects.

To help reduce this cost, researchers came up with various methods to guide the security testing effort. Among those, vulnerability prediction modelling has shown some promising results and research in this area has been pretty active over the past few years.

Most of the studies suggesting new approaches for vulnerability prediction modelling fail to compare their results with previously introduces ones. Furthermore, the way approaches are evaluated often differs from one study to another. This difference can originate from the choice of the dataset, ground truth, evaluation methodology, performance metrics or a combination of those. This makes impossible the comparison of results across studies, which can give an impression of fuzziness and instability. In order to pursue its evolution, research on vulnerability prediction modelling need more comparison and replication studies acknowledging what has already been proposed. Still, novel approaches should continue to emerge but they should always be compared to previous ones. One possibility for a novel approach could be the use of "naturalness of code" that has been successfully applied for defect prediction [156].

In the following, an overview of the main challenges addressed in this thesis is presented.

1.4.2 Challenges addressed in this thesis

Collecting Vulnerabilities: One of the many factors explaining the lack of empirical studies in this line of research is the absence of standard vulnerability datasets, which could be used to evaluate VPM. Indeed most of researchers starts by creating their own datasets. This hinders research in the area as creating a dataset is hard. Indeed, information required is usually scattered in different places. Additionally more than one project should be used to build such a dataset in order to verfify the generalization. Ideally, those projects should in addition have a large number of reported vulnerabilities, be security-sensitive and open source, which reduces the candidates.

In fact, from the researcher point of view, the creation of a dataset is interesting as it ensures a full control over and understanding of the data. Yet this is counterproductive at a larger scale. Moreover, as new vulnerabilities are found on a daily basis, a dataset can quickly be outdated and bias the results.

Finally, vulnerability datasets are not only useful for vulnerability prediction modelling, but can also be used for a large variety of analysis. Among the possible analyses, the ones analysing vulnerabilities properties are of special interest as when put together with the results of VPM, they can help developers understand the output of the model.

Challenge #1:
The first challenges addressed in this thesis is the automatic collection of vulnerable code instances. The collection process can be used in order to make a reliable, evolutive, multi-project and large dataset that is suitable for vulnerability analysis.

Evaluating the Existing VPM Approach: Previous studies on VPM often do not compare their results with previous works. One explanation, aside from the use of different datasets, lies in the fact that the target and evaluation methodology of the studies are usually different. Hence, even if the metrics used for measuring the performance of the models are the same (which is not always the case) it is unwise to compare the result obtained following a 10-fold cross validation to ones obtained performing next release validation.

The only remaining solution for researcher willing to compare their result with other approaches is thus to replicate them using their dataset, evaluation methodology and target. Unfortunately, only a couple of studies provide replication framework and researchers have thus to recreate the approach based on the information available on the papers. Replicating approaches is time consuming which explains the lack of comparison.

Challenge #2:
The second major challenge addressed in this dissertation is the replication and comparison of existing approaches.

Suggesting a new approach based on naturalness: The application of Natural Language Processing (NLP) to software engineering has received a growing interest in the recent years [25]. In particular, the study of software naturalness [77] has given birth to many approaches for generating source code, *e.g.,* code completion [77], synthesis [155], review [76], obfuscation [115] and repairs [154]) and performing static analyses [81, 104, 140].

The naturalness of software is the measure of how surprising is a software component to a statistical language model trained on other software components. Intuitively, one might think that a "surprising" component might be suspicious. Yet, interestingly, the naturalness of software has never been used for security-related tasks. Additionally, the fact that it has been successfully applied to the area of defect prediction [156] makes it an even more interesting candidate for building a VPM approach upon. History of VPM has shown that approaches working for defect prediction are usually good candidate for vulnerability prediction. Still, the diverse ways to compute naturalness, *e.g.,* tokenization, n, smoothing techniques,..., require careful thinking on which settings to use as features.

Challenge #3:
The third major challenge addressed in this dissertation is the development of a new VPM approaches based on the notion of naturalness.

Using Naturalness of software for Software Engineering: As stated before the study of software naturalness led to the development of many approaches. Still, the spectrum of it possible usages is large and has not been fully explored and some fields of research could benefit from it, like mutation testing.

Mutation Testing mutates part of a program to evaluate its test suite. To test a mutant, it needs to be executed against the test suite. One of the main issues of mutation testing is that generated mutants are numerous and testing them all can take time, while not all of them are of interest. Thus, some preliminary analysis to reduce their number is required and naturalness could be a good indicator to select mutants.

Challenge #4:
The last challenge addressed in this thesis is how naturalness could be used for other software engineering tasks.

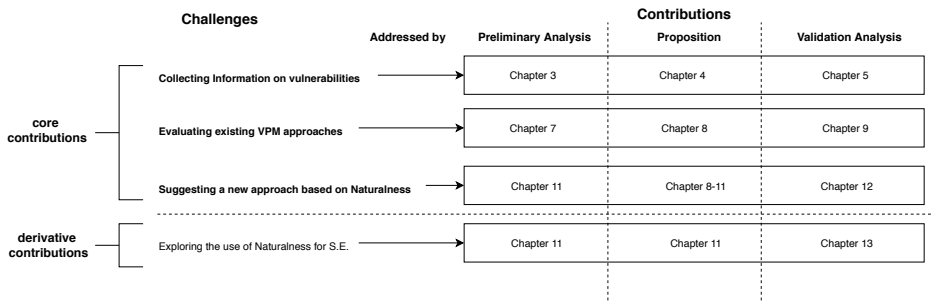


Figure 1.4: Thesis structure

1.5 Contributions and Thesis structure

This thesis is composed of five parts. The first part introduces the thesis, technical background and the state of the art. Then, the next three parts propose solutions to address the main challenges presented in Section 1.4. Finally, the last part concludes the thesis.

Each challenge is addressed using the methodology shown in Figure 1.4. At first, a preliminary study motivates the challenge, then a proposition to tackle the challenge is made under the form of a publicly available framework enabling further analysis. Finally, a validation study is presented. While Parts II and III address respectively the challenges 1 and 2, Part IV focuses on the naturalness of software and addresses the last two challenges (3 and 4).

Part I: Introduction and state of the art. is composed of the present Chapter (1), which introduces the context of this thesis and Chapter 2 which presents the state of the art regarding vulnerability prediction modelling.

Part II: Analysing and collecting vulnerabilities. addresses the challenge of collecting information on vulnerabilities to build a dataset. Chapter 3 presents a manual analysis of Android vulnerabilities. This analysis highlights the difficulties faced by researchers when trying to collect and analyse a large number of vulnerabilities. Then, Chapter 4 presents Data7, a publicly available extensible framework that automatically collects vulnerability fixes and information. This framework can be used as base for a VPM evaluation but also to support empirical analysis of vulnerabilities like the one presented in Chapter 5. This chapter presents an analysis of vulnerability fixes according to their types and severities and gives pointers for specific VPM analysis.

The contributions presented in this part are based on work that has been presented in the following papers:

- Matthieu Jimenez, Mike Papadakis, Tegawende F. Bissyandé, and Jacques Klein. Profiling android vulnerabilities. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 222–229, Aug 2016

- Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, pages 105–112, 2016

- Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the continous analysis of security vulnerabilities with vuldata7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2018

Part III: Investigating Vulnerability Prediction Models is dedicated to the second of four challenges of this thesis, the replication and comparison of existing VPM approaches. Chapter 7 starts by presenting an exact independent replication study of three of the main VPM approaches on a dataset of Linux Kernel vulnerabilities built using the framework of Chapter 4. Following this experience, chapter 8 presents an extensible framework that allows practitioners to replicate, evaluate and compare VPM approaches. This framework is then used in Chapter 9 to perform the largest empirical study on VPM comparing the three approaches replicated before using different settings and evaluation criteria.

The contributions presented in this part are based on work that has been presented in the following papers:

- Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the continous analysis of security vulnerabilities with vuldata7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2018

- An Empirical Study on Vulnerability Prediction of Open-Source Software Releases (under review)

Part IV: Naturalness of Software addresses the third and fourth challenges that are related by their use of naturalness. Chapter 11 presents a study on the use of naturalness for software engineering. The focus is put on the effect of code representation and language models parameters on naturalness. The chapter also introduces a framework to compute naturalness of software, which is used to create a VPM approach studied and evaluated in Chapter 12in an effort to address the third challenge. Finally, Chapter 13 address the last challenge by presenting an empirical study on the use of naturalness for the selection of "fault-revealing" mutants.

The contributions presented in this part are based on work that has been presented in the following papers:

- Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. On the impact of tokenizer and parameters on n-gram based code analysis. In *34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29 2018*, 2018

- Matthieu Jimenez, Thierry Titcheu Checkham, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural? a study on how "naturalness" helps mutant selection. In *12th International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, 11-13 October 2018*, 2018

Finally, this dissertation is concluded in Chapter 15, where possible future research directions are discussed.

2

State of the Art

This chapter presents a list of works and studies on VPMs and provides an overview of works on related research topics introduced in Section 1.3. Special care was taken to exhaustively cover all the published studies until the time of writing.

Contents

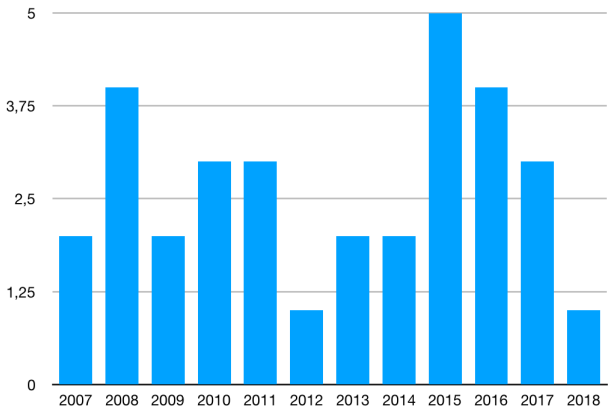


Figure 2.1: Numbers of published papers related to VPMs per year since 2007

2.1 VPMs: Over a decade of studies

The first paper we found related to VPMs was published in 2017 and is that of Neuhaus *et al.,* [133], entitled "Predicting Vulnerable Software Components". Since then, the field receives increasing attention from the research community.

Figure 2.1 presents the number of papers related to VPMs, categorized according to their publication year. The figure shows a constant publication trend as no year passed without any related publication. Interestingly, most of the studies we analyse, differ one another by either the features that are used to build the models, or by the granularity level on which the trained model is working, *e.g.,* file, modules, function,...

In this section, we present all of these works as follows: First we present works relying on dependency analysis. Next, we present all studies using source code metrics. Then, papers using text mining techniques are discussed. After that, a category of works relying on Attack Surface Approximation (ASAp)s is presented. Finally, we present studies that are combinations of other approaches.

2.1.1 Dependencies Analysis

The paper of Neuhaus *et al.,* [133] can be considered as the first attempt to build a VPM. The authors suggest using the include and the function call contains in a file as features to a ML algorithm. This approach is based on the intuition that vulnerable files are likely to share similar sets of imports and function calls which could be used to identify them. The idea is in accordance to the work performed by Schröter *et al.,* [162], which aims at predicting defects. The authors begin this study by performing a correlation analysis of import/function calls and vulnerabilities on a dataset of Mozilla Firefox vulnerabilities to validate the intuition. In their evaluation, the authors perform random splitting and experiment two approaches, one using the includes of the file as features while the other use its function calls. To build the models, the authors opt for the Support Vector Machine (SVM) algorithm.

Their results show that a recall of 45% and precision of 70% can be achieved. These results, although not great, were considered encouraging. The authors also find out that against the common belief, files that are vulnerable once are unlikely to be vulnerable again.

Neuhaus *et al.,* [132] extend this idea to the analysis of dependencies between packages in Red Hat. Indeed, most packages in Red Hat require the installation of other packages before starting their installation, this process can lead to long chains of dependencies that might be conflicting and propagate vulnerabilities. Similarly, the authors start by providing evidence that vulnerabilities indeed correlate with dependencies using a dataset containing more than 3,000 packages. Then, they perform an analysis to find out the dependencies that increase the chances of a package to be vulnerable ("beasts") and those that reduce it ("beauties"). Next, they use their dataset to build and evaluate VPMs using dependencies of binaries as features. To build the models, two different ML algorithms are considered, the first one SVM that has already been used in the previous study has the benefit of being less prone to overfitting. The second one, C4.5 is a decision tree based algorithm and thus has as major benefit to be understandable by humans. The authors find out that models using SVM algorithms are performing better and reach a median precision of 83% and a median recall of 65%. Finally, the authors explore the possibility to use a linear regression model to predict packages that might turn out to be vulnerable in the upcoming months. The use of this technique on their dataset led to the flagging of 25 packages, from which 9 turned out to be vulnerable.

Nguyen *et al.,* [137] further investigate the power of dependencies as features by analysing the relation between software elements at various granularity, *e.g.,* components, class, functions, variable instead of sticking to a single one. In the previous studies, the dependencies were either at the file level, or the binary one, which means that they can be represented as a simple text elements. However, dependencies in lower granularity cannot be represented as such and are traditionally presented under the form of graphs, which are not convenient to use as features. To tackle this issue, the authors suggest extracting metrics from dependency graphs and use those as features. To compute those metrics, the authors rely on two kinds of graphs, Member Dependency Graphs (MDG) which are direct graphs built from data items and method of a software system and focus on member levels and Component Dependancy Graphs (CDG) which can be generated from the MDG by gathering all member nodes of a same component node into a component node. In total, 22 metrics based on these graphs are presented. The authors evaluate models built on those metrics on two versions of the JavaScript engine of Mozilla Firefox with 5 different ML algorithms and find out that in terms of recall Bayesian Network (BN) performs best with score values reaching 75%, while SVM and Neural Network (NN) outmatch all others in terms of precision by reaching score values up to 85%.

Table 2.1: Summary of 2.1.1

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Neuhaus *et al.,* [133]	Include, Function calls	Mozilla Firefox	File	2/3 split (40 times)	SVM	Precision 70%, Recall 45%
Neuhaus *et al.,* [132]	Dependencies	3000 Red Hat Packages	Binary (Package)	2/3 split (50 times)	SVM, C4.5	Precision 83%, Recall 65%
Nguyen *et al.,* [137]	Dependancy graph metrics	Mozilla Firefox	File	10 fold cross validation	BN, Naïve Bayes (NB), NN, RF, SVM	Precision 68%, Recall 60%

2.1.2 Code Metrics

Extracting code metrics from a graph model has the advantage of having a prediction model with a fixed number of features. This guarantees that the dimension of the model will not depend on the training set. This leads to another category of VPMs approaches presented in this section that use metrics computed from the source code to build their models. The initial studies in this line of works are those carried out by the studies of Shin *et al.,* .

The seminal study of Shin *et al.,* [169] is the starting point of all studies that use complexity metrics to build VPMs. Shin *et al.,* [169] investigated whether complex code correlates with vulnerabilities and whether there are significant differences in code complexity between vulnerable and faulty code. To verify this, the authors compute a set of 9 complexity metrics for each function present in the 4 releases of Mozilla Firefox. Among those metrics, we can cite MacCabe Cyclomatic Complexity (MCC), essential cyclomatic complexity, nesting and Lines of Code (LoC). Overall, they found a weak correlation between vulnerable code and complexity and that vulnerable code tend to be more complex than the faulty one.

The results of the previous study are then used by the same authors [168] to build VPMs. The idea is directly inspired by the work of Nagappan *et al.,* [130] for defect prediction. In this work, the authors explore 3 different scenarios (1) predict faulty functions from all functions which is a typical case of DPMs (2) predict vulnerable functions from all functions and (3) predict vulnerable functions from faulty functions. For each scenario they build a model using the 9 metrics and binary Logistic Regression (LR) as a ML algorithm. The evaluation is performed on a dataset composed of 6 versions of the JavaScript engine of Mozilla Firefox. The author found that in the first two scenarios, a low false positive rate (high precision) was achievable but at the price of a high false negative rate (low recall), which can be problematic if aiming at reliability. Regarding the third scenario, the results turned out to be inconclusive as no common trends between any of the experiments could be observed.

Shin *et al.,* [170] extend their work to the use of execution complexity metrics as opposed to the previous ones that were static. The intuition is that metrics collected during software execution may be effective for identifying vulnerable code locations. The idea is borrowed from the work of Khoshgoftaar *et al.,* [99] applied to DPMs. The new metrics are number of calls, inclusive execution time and exclusive execution time. Those metrics can be computed while performing usual tasks on the software with tools such as Callgrind. In addition to static and execution metrics, the authors also look into dependency metrics similar to those of Nguyen *et al.,* [137] but based on the work of Zimmermann *et al.,* [200]. In total, 23 metrics are studied. The authors first evaluate the discriminative power of the metrics on Firefox and Wireshark. Results suggest that almost all of the metrics show significant discriminative power on the Firefox dataset while it's not the case for Wireshark. Regarding execution metrics, 2 of them show discriminative power for Firefox but none for Wireshark. Then, the authors build 4 VPMs models using LRs, one per type of metrics and one combining them and find out that models using dependency metrics perform best in terms of precision while the execution metrics ones are interesting for File Inspection Reduction.

Shin *et al.,* [167] perform an even more extensive study on the use of Complexity, Code Churn and Developer Activity (CCD) metrics for vulnerability prediction. If the use of complexity metrics is a logical extension of the previous works, the use of Code Churn and Developer Metrics can be considered as novel. Similarly to the previous works, the idea to use those metrics can be traced back to the work of resp. Nagappan *et al.,* [129] and Meneely *et al.,* [123] in the context of DPMs. In total, the authors suggest the use of 28 metrics (14 complexity, 3 code churn and 11 developer activity) and test their discriminative and predictive power against two open source projects Mozilla Firefox and Red Hat Linux. More information on those metrics can be found in chapter 7. Among the 28 metrics, the discriminative power of 24 metrics was supported for both projects. Regarding the evaluation of the predictive power of the metrics, the authors consider both univariate and multivariate model built using several classifier but only present the result of LR as the result were similar. Regarding the univariate model, *i.e.,* model using only one metric, only the number of changes of a file and its number of developers provided result acceptable by the authors chosen threshold (over 70% recall and below 25% of false alarm) for both projects. As for multivariate models, 4 kinds were considered, one per type of metrics and one combining them all, the model using Code Churn metrics performed the best in Firefox, while the one using developer activity performed the best for Red Hat, the model using all metrics being the most stable between the two projects. The authors conclude that model based on development history are stronger than ones based on complexity.

In a last study Shin *et al.,* [171] investigate the use of traditional DPM based on Complexity Code Churn and Fault history metrics for vulnerability prediction. The authors consider three scenarios (1) a model is built on faulty data and evaluate whether a file is faulty (2) a model is built on faulty data and evaluate whether a file is vulnerable and (3) a model is built on vulnerable data and evaluate whether a file is vulnerable. They evaluate those scenarios on 2 releases of Mozilla Firefox using different classifiers but only presenting the result of LR as they claim that result are similar for all classifiers. They find that result varies greatly depending on the chosen threshold for the classifier, *i.e.,* classifier usually returns a probability of the element under test belonging to a given class and the threshold corresponds to the minimum probability required for this element to be considered of this class. Hence depending on the chosen threshold, the recall and precision value can be inverted. Yet, results suggest that traditional DPMs could replace specialized VPMs. This is explained by the authors as the "needle effect", *i.e.,* the lack of vulnerable example to build the model, indeed the authors observe as well that if models built on faulty data are given the same amount of faulty example than their vulnerable counterpart, then results are similar. The consequence of this effect are investigated in Chapter 9.

Chowdhury *et al.,* [48, 49] introduce the idea of using coupling and cohesion metrics in addition to the complexity ones as indicators of vulnerabilities especially for Object Oriented Program. For each category of metrics, the authors suggest metrics that can be computed at code level or design level. All code level metrics are similar to the ones of Shin *et al.,* [167] while design level ones are novel to the area. In a first study [48], they investigate whether these metrics correlate with vulnerability using Firefox as a case study and found that complexity and coupling metrics are indeed strongly correlating with vulnerability while cohesion metrics have a medium correlation.

Then in an extension of this study [49], they use those metrics to build models and test 4 different classifiers (C4.5 Decision Tree, RF, LR and NB) on 52 releases of Firefox. They found out that algorithm based on a decision tree (RF and C4.5) outperforms the other algorithms and could reach 75% of recall and 28% of false positive rate.

Table 2.2: Summary of 2.1.2

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Shin *et al.,* [168]	Complexity	JSE Mozilla Firefox	File	-	LR	False Positive 1%, & False Negative 90%
Shin *et al.,* [170]	Static and Execution complexity, dependencies	Mozilla Firefox, Wireshark	File	10*10 fold cross validation	LR	Precision 67-80%, Recall 12%
Shin *et al.,* [167]	Complexity, Code Churn Developer activity	Mozilla Firefox, Red Hat	File	10*10 fold cross validation next release validation	LR, J48, RF, NB, BN	Recall 85% & False Alarm 24%
Shin *et al.,* [171]	Complexity, Code Churn Fault history	Mozilla Firefox	File	10*10 fold cross validation	BN, J48, RFLR	Threshold 0.5: Recall 70% Precision 49% Threshold 0.84: Recall 36% Precision: 76%
Chowdhury *et al.,* [49]	Complexity, Coupling Cohesion	Mozilla Firefox	File	10*10 fold cross validation	NB, RF, C4.5, LR	Recall 75% False Positive: 28%

2.1.3 Attack Prone Components

All these works based on metrics focus on file level prediction which is the most used granularity for VPMs, yet it is possible to work at different granularity. This part presents a series of works related to the prediction of Attack Prone Components (APC)s performed at the module level by Gegick *et al.,* .

APCs are a subset of Vulnerability Prone Components that are likely to be exploited. Indeed the existence of a vulnerability is not always paired to an exploit. In fact, most of vulnerabilities haven't reported exploit.

The vision paper of Gegick *et al.,* [187] is the first of this series. It introduces the different definitions, ideas and experimental processes that are used in all of their studies. Among the various suggestions, the most important one is to use results of ASAs tools to identify vulnerabilities and more precisely compute metrics from the warning. In a following technical report, Gegick *et al.,* [67] present their use case which is a large Cisco project of over 1.2 million lines of code along with the chosen metrics (1) ASAs tool internal metrics (2) LoCs and code churn (3) Count and density of pre-and post-release failure (security and non-security-related). They also describe the Classification and Regression Tree (CART) ML algorithm selected to build models. Next, the authors present the results obtained while testing on 25 components out of the 38 that the project is composed of[66]. They build one model per category of metrics and one combining them all and find that the model using all metrics performs best achieving 8% of false positive and 0% of false negative.

In a follow-up study [65], the authors investigate the use of unit tests and static inspection in addition to the formerly introduced metrics as features. The authors achieve a precision of 80% and a recall of 48.8% in the best case.

Table 2.3: Summary of 2.1.3

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Gegick *et al.,* [66]	ASAs, LoCs code churn, failure count	Cisco	Module	5 fold cross validation	CART	False Positive 8% & False Negative 0%
Gegick *et al.,* [65]	ASAs, LoCs, code churn unit tests, static inspection	Cisco	Module	5 fold cross validation	CART	Precision 80%, Recall 48.8%

2.1.4 Text Mining

This part is devoted to studies suggesting using techniques from the text analysis field to build models. Similarly to previously introduced approaches, the idea of reusing text analysis techniques for vulnerability prediction can be traced back to works on defect prediction such as the one performed by Hata *et al.,* [75].

In a position paper, Hovsepyan *et al.,* [79] introduce a novel approach based on technique from the text analysis area called bag of words. The idea according to the authors is to consider raw source code as text and apply text analysing techniques on it instead of using "cooked" features, *e.g.,* code metrics. Hence, they suggest to tokenize the code as any regular English text and create a bag of words out of it. The bag of words can then be used as a feature by a ML algorithm. The authors evaluated the idea on a dataset composed of 19 releases of K9 mail Android applications. In this study, vulnerabilities (obtained through the use of an ASAs dedicated to security) present in the first release are used to train a model and which is then applied on all following releases. Results obtained were considered as encouraging result with an average recall of 88% and precision of 85%.

In a follow-up work, Scandariato *et al.,* [161] evaluated the same approach on 20 Android open source applications and Drumel a php project. In a similar manner, the authors rely on a static analysis tool to build their dataset of vulnerabilities. The choice of relying on this kind of ground truth instead of one based on vulnerability reports is justified by the authors on the basis of results obtained by Walden *et al.,* [183] showing that there is a strong correlation between warning from security static analysis tool and vulnerabilities. In this study, the authors evaluate their models using 3 different means: (1) next release prediction (2) cross validation and (3) cross project predictions. They found interesting results, reaching in average a recall of 77% and a precision of 90%. More information on this study can be found in chapter 7.

Pang *et al.,* [143] suggest using a Bag of N-Grams instead of a Bag of Words as feature to create the model. To evaluate this idea, they use the data of 4 of the Android applications from the study of Scandariato *et al.,* [161]. The authors perform both cross validation and cross project prediction and obtained a precision of 95% and a recall of 87% for cross validation and a precision of 67% and a recall 63% in the case of cross project prediction. The same authors [144] extend their study to the use of a deep NN instead of SVMs use in the first study and found slightly better result with both precision and recall close to 95%. However, in both papers the authors emphasize on the need for more experiments to validate those results.

All works on text mining based approach share the advantage of their simplicity, in a similar manner another kind of approach consists in the analysis of crash report instead of source code.

Table 2.4: Summary of 2.1.4

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Hovsepyan *et al.,* [79]	Bag of Words	K9 Mail Android	File	Next Release	-	Accuracy 87%, Precision 85%and Recall 88%
Scandariato *et al.,* [161]	Bag of Words	20 Android Applications, 3 PHP	File	10 fold cross validation, next release and cross Project	NB, RF	Precision 90%and Recall 77%
Pang *et al.,* [143]	N-Grams	4 Android Applications	File	5 fold Cross Validation, Cross Project	SVMs	Precision 95%and Recall 87%
Pang *et al.,* [144]	N-Grams	4 Android Applications	File	5 fold Cross Validation, Cross Project	Deep NN	Precision 95%and Recall 95%

2.1.5 Attack Surface Approximation

In this part, approaches suggested by Theisen *et al.,* leveraging crash dumps to identify part of system that might be vulnerable are presented.

Crash dumps are frequently used to localized defects in a program [185], thus Theisen *et al.,* [177] came up with the idea of applying it to vulnerabilities. In particular, they introduce the notion of ASAp, an automated approach to identify parts of a system that are contained on the Attack Surface through the analysis of the stack trace of crash reports. ASAp has two major interests, first it can be used as a replacement of traditional VPMs to find potential vulnerable components, second it can be used to narrow down the space VPMs should investigate. To investigate the usefulness of ASAp, the authors evaluate using a dataset of Windows 8 vulnerabilities [201] and use crashes from 3 different sources (kernels, user and fuzzing tools) as features. The authors in a first time compared the results obtained by ASAp against a traditional VPMs based on code metrics and found better precision and recall for both binary and file level. Then the authors look whether ASAps could be used to improve results of traditional VPM and found that it was the case even if the gain were marginal. They also found that crash from fuzzing tool are redundant with kernels and user crash.

In a follow-up study, the same authors [178] investigate how to reduce the number of traces required to perform ASAps while maintaining acceptable results. Indeed, the analysis of all stack traces can be really expensive. While performing the same experiments as in their previous paper on Firefox and Windows datasets, they found that simply performing a 10-fold random sampling at 10% just was only reducing the result by 3% while significantly reducing the cost of analysis.

In the previous sections, studies introducing new approaches were presented, the next one introduced those that tried to combine them.

Table 2.5: Summary of 2.1.5

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Theisen *et al.,* [177]	Code Metrics & ASAps	Windows 8	Binary & File	100* 2/3 split	RF	Precision 69%and Recall 10%

2.1.6 Combined Approaches

Perl *et al.,* [150] study the effect of using metadata contained in source code repositories in addition to code metrics to flag vulnerability contributing commits (VCC). Software evolves over time and most of the time developers rely on a Version Control System (VCS) to keep track of the changes, making commits (changes) a natural unit of evolution worth studying. The authors thus gather a dataset of 170,000 commits from 66 Open Source Github projects from which 640 are flagged as vulnerable. Among the metrics selected by the authors to use as features for VPM identifying VCC, most are previously introduced metrics like code churn and developer activity, but some are novel and extracted straight from the VCS such as commit messages (converted into a bag of words). The authors then build a model using the SVM algorithm and data up to the end of 2010 as training and the one following as testing. In the end, the model reached a precision of 60%, while obtaining a recall of 25%.

Zhang *et al.,* [199] propose, on the other hand, to combine code metrics and text mining techniques. To do so, the authors present an approach combining the result from 6 different classifiers, three train on code metrics, and three on text mining. To combine the different predictions into a single one, the authors suggest using another classifier (RF) on top that use the confidence score of each classifier for each training example as training. The authors evaluate this approach on the same dataset as Walden *et al.,* [184] (see 2.1.7) composed of 3 PHP applications. Overall, the authors managed to improve the precision in every case while only improving the recall in one case.

Moshtari *et al.,* [127] investigate the power of complexity and coupling metrics on cross project prediction especially the case of projects programmed in different languages. The authors built their model with the same features as in Shin *et al.,* [169] study but also add what they refer as included vulnerable header metrics (IVH). This last category of metrics corresponds to the number of headers related to sensitive calls like IO, network, databases, memory and systems and is in fact inspired by the work of Neuhaus *et al.,* [133]. To evaluate the possibility to use models trained on those features for cross prediction, the authors build a dataset based on 5 Open Source Software (Apache Tomcat, Eclipse, OpnScada, Mozilla Firefox and Linux kernel) using the output of a security ASA tool as ground truth. The authors found that the combination of complexity metrics and IVH was greatly improving the results with recall reaching up to 87%.

Yu *et al.,* [196] propose a tool named "Harmless" to guide security review process by building an active learning model based on code metrics, text mining and crash log. On the opposite of the previously presented studies that predict whether files are vulnerable in batch, the idea of the tool is to order the file to review according to their chance of being vulnerable. Hence, the goal is not to have an accurate binary prediction but to have most of the vulnerable file highly ranked, *i.e.,* to obtain a recall of 100 for the least cost. To evaluate their tool, they compare it to other approaches on the same dataset as Theisen *et al.,* [177] and found better result than models built using only one kind of approach .

In this section, works combining different approaches to build better ones have been introduced, yet that is not the only way to improve the results. In the next section, an overview of studies that tries to improve the results using different ML techniques is given, along with the one that replicate and evaluate the impact of existing VPMs.

Table 2.6: Summary of 2.1.6

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Perl *et al.,* [150]	Code Metrics, VCS info	66 Open Source Projects	Commits	Time Split	SVM	Precision 60%and Recall 25%
Zhang *et al.,* [199]	Code Metrics, Text Mining	3 PHP Applications	File	10 fold cross validation	RF,NB, C4.5	Precision 25-67%and Recall 4-69%
Moshtari *et al.,* [127]	Complexity, Coupling & Dependency	Tomcat, Eclipse, OpnScada Firefox, Linux kernel	File	Cross Project	NB, LR, RF, clustering	ROC 80% and Recall 87%
Yu *et al.,* [196]	Code Metrics, Text Mining, ASAp	Mozilla Firefox	File	Cost	Active Learning	53% of files for 100% Recall

2.1.7 Replication, Improvement and Impact studies

Finally, in this section, studies that replicate and evaluate the practicality of existing approaches as well as those exploring techniques to improve the results like dimensionality reduction.

Following their work on text mining [161], Walden *et al.,* [184] compare the result obtain by models built using bags of words against one's using code metrics from Shin *et al.,* [167] study. To perform this comparison, they build a dataset of 3 PHP Applications Drummel, Moodle and PHPmyAdmin containing 223 vulnerabilities and cross validate the results. In the end, the authors found that text mining models were always performing better than their code metrics counterparts.

Hovsepyan *et al.,* [78] extend this study to the use of those models for future prediction instead of classical cross validation from one release to the other on a dataset containing 9 releases of Firefox and 15 releases of Google Chrome. They find that when training on one release and keeping the model is more stable than training a new model based on the previous release each time a new release is made available. Models using previous release perform in average 5-10 % worse than models trained on an old release.

Stuckman *et al.,* [173] investigate the effect of dimensionality reduction techniques on VPMs. They consider 5 techniques, Feature Subset Selection (FSS), Entropy Reduction (FS), Principal Component Analysis (PCA), Sparse Principal Component Analysis (SPCA) and CFA and evaluate them on two approaches one based on code metrics and one based on bag of words. The authors reuse the PHP dataset presented in the work of Walden *et al.,* [184] to evaluate the performance of the different techniques. The authors find that CFA technique can slightly improve code metrics models while no techniques can improve the bag of words ones. Still, the use of dimensionality reduction techniques reduces the gap in computation time between code metrics-based models and text mining ones, while just slightly worsening the results.

Zimmermann *et al.,* [201] were the first to investigate the use of VPM in an industrial context and to replicate studies. The authors test models using complexity, churn, coverage, dependency and organizational metrics and their combination as well as models based on dependency and evaluate it on Windows Vista at the binary level. First, the authors confirm the existence of a weak correlation between the metrics and vulnerabilities. Then they build metrics models using LR and dependency models using SVM and find that combination models are performing better among the metrics ones, while ones based on dependencies have a better recall for the same precision.

Morrison *et al.,* [126] further investigate the reproducibility of the results on Windows 7 and 8 at two granularities binary and file level. Indeed, results at the binary level tends to be good, but they are less actionable, while results of file level are insufficient but actionable. The authors evaluate different models based on 29 metrics from 6 different categories of metrics, churn, complexity, dependency, legacy, size and vulnerability history and a large set of classifiers. After analysing the results, the authors conclude that in the current state, results are not good enough to be of actual use. Indeed at best, a precision of 76% or a recall of 42% is reached at binary level, *resp.* 47% or 14% at file level, which is far from the initial goal of 70% recall and precision.

Moshtari *et al.,* [128] replicate the study of Shin *et al.,* [167] but only using complexity and coupling metrics. They start by performing an experimentation similar to the one of Shin *et al.,* , *i.e.,* performing next release validation, on a dataset of 35 releases of Mozilla Firefox and find precision and recall above 90%. Then the authors investigate whether the same result can be achieved in a cross project prediction context but find at best a precision of 20% and a recall of 31%.

Alves *et al.,* [26] replicate 18 experimentation from state-of-the-art studies using software metrics on a dataset of 2875 vulnerability patches from 5 software Firefox, Linux kernel, httpd, glibc, XenHV [27]. Overall, the authors find that the accuracy metric is not relevant and that other metrics such as informedness and markedness are more interesting. They also observe that models using RF are performing better in all cases.

Table 2.7: Summary of 2.1.7

Study	Features	Dataset	Granularity	Evaluation Method	Algorithm	Result
Walden *et al.,* [184]	Code Metrics, Text Mining	Drummel, Moodle, PHPmyAdmin	File	5 fold cross validation	RF	Precision +0.5-5%, Recall +4-10%
Hovsepyan *et al.,* [78]	Code Metrics, Text Mining	Mozilla Firefox, Google Chrome	File	next release validation	RF	Recall -5-10%
Stuckman *et al.,* [173]	Code Metrics, Text Mining	Drummel, Moodle, PHPmyAdmin	File	10 fold cross validation, next release validation	RF	-
Zimmermann *et al.,* [201]	Code, Dependancy, Organizational, Coverage Metrics, Dependencies	Windows Vista	Binary	2/3 random split	LR, SVM	Precision 66%, Recall 20%
Morrison *et al.,* [126]	Code, Dependancy, Legacy and Vulnerability History	Windows 7 & 8	Binary File	100 * 2/3 random split	Recursive Partitioning (RP),LR, SVM, NB, RF ,Tree Bagging (TB)	Precision 76%, Recall 42%
Moshtari *et al.,* [128]	Complexity, Coupling	Apache Tomcat, Eclipse, OpnScada, Firefox, Linux kernel	File	next release cross project	BN, J48,NB, LR, RF, clustering	Precision 97% and Recall 91%
Alves *et al.,* [26]	Code Metrics	Firefox, Linux kernel, httpd, glibc, XenHV	File	-	-	-

2.1.8 Summary

VPMs have received an increasing amount of attention over the past few years. This chapter gathered and introduced 32 papers directly related to VPM. In Table 2.8, we can observe that models based on code metrics-based features have been used in most of the studies. Hence, the LoC metrics have been used in at least one model in 80 % of the presented studies. This trend is somehow similar to the one observed in DPM, where those models that are easy to build and evaluate are often used as a comparison basis. In terms of results, text mining based techniques seem to outperform the others according to the works considering it. Yet its high computation demand is clearly a downside, that dimensionality reduction can just slightly temper.

The use of the different ML algorithms is summarized in Table 2.9, where we see that many different possibilities have been tested. RF is the most used technique, present in more than 50% of the related studies, and seems to be the best performing. LR which easy to interpret outputs have been largely used as well with less success.

The projects used to build the evaluation datasets of the studies are presented in Table 2.10. Firefox have been extensively used, this can be explained by the high number of vulnerabilities reported and the ease to collection information on them through the MFSA. Still the use of a project remains tightly linked to the researchers. In most cases, a new group working on VPMs lead to the introduction of a new dataset, which hinders replication and comparison between studies. The explanation of this multiplication of datasets, which sometimes doesn't even share the same ground truth, *i.e.,* based on vulnerability reports, or the output of an ASA, lies in the fact that most are not publicly available.

Table 2.8: Table of features used in the different studies

Paper \ Feature	Dependancy	Calls	LoC	Complexity	Coupling	Dep. Metrics	Nb Function	Comments	Runtime	Churn	Devs	Bug/vul Hist	Cohesion	ASA	Unit Test	Bag of Word	N-Gram	ASAp	Commit
Neuhaus *et al.*, [133]	X	X																	
Neuhaus *et al.*, [132]	X																		
Nguyen *et al.*, [137]			X	X	X	X													
Shin *et al.*, [168]			X	X	X		X	X											
Shin *et al.*, [170]			X	X	X			X	X										
Shin *et al.*, [167]			X	X	X			X		X	X								
Shin *et al.*, [171]			X	X	X			X		X		X							
Chowdhury *et al.*, [48]			X	X	X		X	X					X						
Gegick *et al.*, [66]			X							X		X		X					
Gegick *et al.*, [65]			X							X		X		X	X				
Scandariato *et al.*, [161]																X			
Pang *et al.*, [143, 144]																	X		
Theisen *et al.*, [177, 178]			X	X	X	X		X		X		X		X	X			X	
Perl *et al.*, [150]							X			X	X								X
Zhang *et al.*, [199]			X	X	X	X										X			
Moshtari *et al.*, [127]	X		X	X	X		X	X											
Walden *et al.*, [184]			X	X	X	X	X									X			
Howsepyan *et al.*, [78]			X	X	X	X	X	X					X			X			
Stuckman *et al.*, [173]			X	X	X	X	X	X					X			X			
Zimmermann *et al.*, [201]	X		X	X	X	X	X			X	X				X				
Morrison *et al.*, [126]			X	X	X	X	X	X		X	X	X							
Moshtari *et al.*, [128]			X	X	X			X											
Alves *et al.*, [26]			X	X	X	X	X	X		X	X		X						
Total	4	1	19	17	17	10	11	13	1	10	5	6	4	2	2	4	1	1	1

Table 2.9: Table of ML algorithms used in the different studies

Paper \ ML	SVM	C4.5	NB	BN	RF	NN	LR	J48	CART	Clustering	TB	RP
Neuhaus *et al.,* [133]	X											
Neuhaus *et al.,* [132]	X	X										
Nguyen *et al.,* [137]	X		X	X	X	X						
Shin *et al.,* [168]							X					
Shin *et al.,* [170]							X					
Shin *et al.,* [167]			X	X	X		X	X				
Shin *et al.,* [171]				X	X		X	X				
Chowdhury *et al.,* [48]		X	X		X		X					
Gegick *et al.,* [66]									X			
Gegick *et al.,* [65]									X			
Scandariato *et al.,* [161]			X		X							
Pang *et al.,* [143, 144]	X					X						
Theisen *et al.,* [177, 178]					X							
Perl *et al.,* [150]	X											
Zhang *et al.,* [199]		X	X		X							
Moshtari *et al.,* [127]			X		X		X			X		
Walden *et al.,* [184]					X							
Hovsepyan *et al.,* [78]					X							
Stuckman *et al.,* [173]					X							
Zimmermann *et al.,* [201]	X						X					
Morrison *et al.,* [126]	X		X		X		X				X	X
Moshtari *et al.,* [128]			X	X	X		X	X		X		
Total	7	3	8	4	13	2	9	3	2	2	1	1

Table 2.10: Table of the project used in the different studies

Paper \ Project	Mozilla Firefox	Red Hat Packages	Wireshark	Linux Kernel	Cisco	Android Apps	PHP Project	Windows	Apache Project	Eclipse	OpnScada	Chrome	httpd	glibc	XenHV
Neuhaus *et al.*, [133]	X														
Neuhaus *et al.*, [132]		X													
Nguyen *et al.*, [137]	X														
Shin *et al.*, [168]	X														
Shin *et al.*, [170]	X		X												
Shin *et al.*, [167]	X			X											
Shin *et al.*, [171]	X														
Chowdhury *et al.*, [48]	X														
Gegick *et al.*, [66]					X										
Gegick *et al.*, [65]					X										
Scandariato *et al.*, [161]						X	X								
Pang *et al.*, [143, 144]						X									
Theisen *et al.*, [177, 178]	X							X							
Zhang *et al.*, [199]							X								
Moshtari *et al.*, [127]	X			X					X	X	X				
Walden *et al.*, [184]							X								
Hovsepyan *et al.*, [78]	X											X			
Stuckman *et al.*, [173]							X		X						
Zimmermann *et al.*, [201]								X							
Morrison *et al.*, [126]															
Moshtari *et al.*, [128]	X			X					X	X	X				
Alves *et al.*, [26]	X			X									X	X	X
Total	13	1	1	4	2	2	4	3	2	2	2	1	1	1	1

2.2 Analysing and detecting vulnerabilities

In this section, an overview of works on related research topics such as vulnerability analysis and discovery is presented.

2.2.1 Vulnerability discovery

As presented in Section 1.3, when trying to discover vulnerabilities in a software, different techniques can be used other than VPMs which focus more on narrowing down the search space for vulnerabilities. Hence we can cite, Security oriented Static Analysis tools, Fuzzing tools, Penetration testing, ...

Static Application Security Testing (SAST) Tools [20] which falls under the category of static analysis approaches, analyse the source code or a compiled version of it to find vulnerabilities. This type of tool usually relies on an existing set of rules that will highlight the vulnerabilities. One of their main benefits is their scaleability and their understandable output. They are particularly well fit for finding vulnerabilities like buffer overflow and SQL injection. However they are neither sound, nor fitted to detect authentication problem or access control issues. Examples of such tools are FindSecBugs [9] and Fortify [10].

Fuzzing Tools [11] belong to dynamic analysis approaches and are black box software testing tools trying to find bugs in implementation by injecting malformed/semi-malformed data in an automated way. Fuzzers tend to find simple bugs. Yet if the bug provokes a crash, it can be considered as a vulnerability as there is an impact on the availability. Interestingly, the less a fuzzer is knowing about the expected input, the more error it will return. A real downside of fuzzers is their hard to interpret output.

Another possibility is to use penetration testing (pentest) either in an automated way or manual one. Pentest is generally used to find whether the system is exposed to known vulnerabilities, but can as well lead to the discovery of new vulnerabilities.

Austin *et al.,* [31] investigate those techniques and their complementarity on Tolven, CCHR and Open EMR, *i.e.,* exploratory manual penetration testing, static analysis, automated penetration testing and systematic manual penetration testing, the latter finding the most vulnerabilities while observing that vulnerabilities found by static analysis tools were of different kinds.

DaCosta *et al.,* [52] study is one of the first on automated vulnerability retrieval. The authors suggest that function near a source of input are the likeliest to contain a vulnerability and hence build a tool named FLF (Front Line Functions) detecting such functions and evaluate it on 31 vulnerabilities from MICQ, elm, dhcpd and openSSH.

Yamaguchi *et al.,* [192] introduce an approach that extract api call from software to create a vector space, which then by computing similarity can be used to investigate the calls close to a reported vulnerability. They evaluate it on a dataset based on the Linux kernel and ffmpeg and found 2 new vulnerabilities and one exploit. They then extend this approach [193] to the analysis of Abstract Syntaxic Tree (AST). They evaluate it on a new dataset and found several 0-day vulnerabilities.

2.2.2 Extrapolating information on Vulnerabilities

VPMs are used to predict components that are likely vulnerable, but it is not the only possible usage of prediction models for vulnerabilities.

As an example, Alzhami *et al.,* [24] suggest an approach to guess the number of vulnerabilities in a system. To do this, the authors rely on two metrics (1) the vulnerability density and (2)the vulnerability discovery rate and use them to build a model. They perform an evaluation on 5 version of Windows and 2 of Red Hat Linux and observe that most of the vulnerabilities discovered lately in the life of a system are linked to the release of a new version. While, Pokhrel *et al.,* [153] suggest on the other side to rely on the use of time series predictive model to predict the number of vulnerabilities present in a system and found interesting result with a non-linear model.

With a close concern, Zhang *et al.,* [197] try to determine whether using metadata from the NVD, the time to the next vulnerability discovery could be predicted but found negative result. Using the same source of information, Bozorgi *et al.,* [40] try to create models to determine whether a vulnerability is in fact exploitable based on the field of CVE.

2.2.3 Vulnerability Analysis

To design VPMs, it is important to first understand vulnerabilities. In this regard, researchers have conducted various studies to deepen their knowledge on vulnerabilities.

Bosu *et al.,* [37] investigate whether code reviews that are led by humans can indeed identify potential vulnerabilities. To do so, they analyse 267,046 code review from 10 major Open Source Software and found positive results. They also discover that in agreement with common belief less experienced developers are 1.8 to 24 times more likely to introduce a vulnerability. Another logical discovery is that the likelihood of introducing a vulnerability increase with the number of lines that is modified. This can be directly linked to the result of Perl *et al.,* [150] which try to identify Vulnerability Contributing commits. A less obvious result is that the risk to introduce a vulnerability is higher in the case of a file modification than in a file creation, meaning that files are unlikely to be vulnerable from the start.

Pianco *et al.,* [151] analyse the change history of vulnerable function of software with the goal to determine if some change metrics could be of use as indicators. The authors investigate this on 17,000 function from Mozilla Firefox and the Linux Kernel and found some correlation.

Ferreira *et al.,* [58] research whether the configuration complexity has an impact on the occurrence of vulnerability in a program. Configuration complexity represents the specific configurations the program will handle. Typically in C program, it can be linked to the presence of if def. After the analysis of the configuration complexity of the Linux Kernel, the authors found that vulnerable function had indeed higher variability than non-vulnerable ones.

Part II

Analysing and Collecting Vulnerabilities

3

A Manual Analysis of Android Vulnerabilities

To be of real use, VPMs require their users to understand vulnerabilities, their origins and consequences. If the latter is often clearly described in vulnerability reports, the former is not always as clear and might need additional crawling. Moreover, such information is necessary to build a VPM. This chapter by presenting an analysis of the origins of all Android Vulnerabilities reported between 2008-2014 introduces to the first challenge of this dissertation, while providing an interesting insight on vulnerabilities affecting the Android operating system.
This chapter is based on work that has been published in the following paper:

- *Profiling android vulnerabilities (QRS'16)*
  M Jimenez, M Papadakis, TF Bissyandé, J Klein

Contents

3.1 Introduction

As presented in Chapter 2, different solutions have been suggested to predict and discover vulnerabilities, yet before even relying on such approaches, it is important for practitioners to get accustomed with vulnerabilities. This was demonstrated by Baca *et al.,* [32] that shown that the experience of developers greatly impacts the benefits of the use of static analysis tool and/or a prediction models. This is also important for researchers, as it can inspire them or simply deepen their understanding. Yet, studies on the nature and origins of vulnerabilities are fewer than ones on vulnerability discovery. A major reason for this is the extensive manual work required to gather and analyse data on vulnerabilities. This chapter presents such an analysis performed on Android vulnerabilities focusing on the root causes of vulnerabilities.

Android is currently the most used operating system for hand-held devices such as smartphones, and is trending in other embedded system products, such as TV sets and Internet boxes. In the first quarter of 2018 alone, 329 million Android devices were sold, which represents 85% of the market [83]. These sales also exceed, by far, the number of personal computers that were sold in the entire year 2016 (269 million [84]). A direct consequence of this widespread adoption is that a single vulnerability within Android, or one of its embedded libraries, can impact a huge number of users.

Such vulnerabilities should thus be corrected as soon as possible. However, Android suffers from a huge fragmentation of the actively used versions. Indeed, by the end of 2017, 28% of the devices connecting to the Google Play Store are still running a "Lollipop" Android Version that was released in 2014 and only 15% of the devices use the latest version of Android released end of 2016. Thus, patching vulnerabilities that are present in more than one version turns out to be a tremendous task.

To find those likely vulnerable parts, we first need to understand what is a vulnerability in the current context. In other words, we need to provide an answer to the following questions:

From what kind of defects are the vulnerabilities coming from? Are they hard to fix? What are the kind and the role of the vulnerable components?

To answer these questions, we manually investigate the code changes that were made to fix known Android vulnerabilities from 2008 to 2014. This investigation focus on three properties of vulnerabilities: root cause, complexity of the vulnerable part and location. The root cause helps to understand the vulnerability's origin, whereas the complexity suggests the easiness or difficulty of removing it, while the location gives indications about the most vulnerable parts of the system under investigation. In the remainder of the chapter, we refer to those root causes as issues.

To study likely true and exploitable Android vulnerabilities, we need a reliable ground truth and hence chose to use information from the NVD. The experience in crawling this data highlighted the difficulty to use the NVD for performing comprehensive and large-scale study of vulnerabilities.

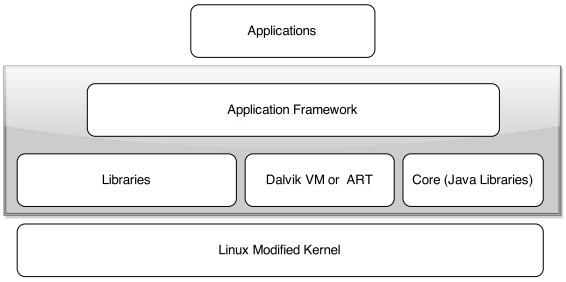


Figure 3.1: The four layers of the Android operating system.

This chapter involves intensive manual work to collect missing/hidden information on Android vulnerabilities recorded in the NVD. This information includes the often missing links to the relevant patches, which are necessary to the creation of any VPM, as well as some valuable information that is hidden in the description or in the natural language, of the vulnerability.

Based on the complete dataset, A taxonomy of the issues that are reported in CVEs is built. A characterization of the fixes made to correct the vulnerabilities is also made by summarizing (1) their origin (2) the complexity of the code where the vulnerabilities are found (3) the complexity of the fixes themselves and (4) the different actions that were necessary to fix the vulnerabilities.

To sum up, the contributions of this chapter are:

- A collection of Android vulnerabilities present in the NVD enriched with information mined outside of it is introduced. This collection is available at :`https://github.com/electricalwind/ReasearchData/tree/master/AndroidVulnerabilities/Android`
- Findings of an empirical study on Android vulnerabilities are presented.
- The difficulty of crawling the NVD are described.

3.2 Background on Android

The Android Operating System is composed of four layers, which are depicted by Figure 3.1. The first layer is a modified Linux kernel. The layer on top of the Linux kernel encompasses C/C++ Libraries, Java Core Libraries and the Dalvik Virtual Machine (DVM), which has been renamed as Android Runtime (ART). The DVM and the ART are used to run user applications and embedded applications written in the Java programming language. Finally, the third layer contains built-in Android applications while the last layer is for user applications.

Since Android is Open Source (available on Github[1] and on GoogleSource[2]), practitioners and researchers can adapt and use it for their specific needs. In this study, we focus on vulnerabilities related to the Android system, i.e., the second (libraries and DVM) and third (built-in applications) layers (highlighted in gray on Figure 3.1). The vulnerabilities affecting the two other layers, i.e., the kernel or the applications, are not considered since these vulnerabilities are respectively reported as Linux or application-specific.

3.3 Methodology

This chapter's analysis is based on all vulnerabilities related to Android reported between 2008 and 2014 in the NVDs (42). The analysis can be summarized by the following steps (1) mining the NVD to find the actual reported issues and their corresponding patches (2) classifying the vulnerabilities (3) Analysing the vulnerabilities.

The described process is designed to answer the three following Research Questions (RQs):

RQ1. What kinds of issues cause vulnerabilities? Can we categorize them?

RQ2. In which components the Android vulnerabilities are located?

RQ3. How vulnerabilities are fixed?

3.3.1 Vulnerability & Issue Mining

The first step is to retrieve Android Vulnerabilities from the NVDs. To this end, we manually mine the database to look for vulnerabilities related to Android. Unfortunately and despite the fact that NVDs provides a lot of useful information, this is not enough. In particular, information on the following three elements are lacking : Vulnerable components, Bug reports, Patches. As a result, an exploration of all the external resources provided via links within CVE is needed.

While crawling all the links present in our set of CVE, we found out that about 20% of the links are dead and 72% in the case of links pointing to patches or bug reports. In fact, this issue is not specific to Android as it turns out that only 30% of the external links declare in CVE are still valid and even less when restricting to solely patch links.

This hinders the data mining process as it implies additional searches to find the new location of the resource. This issue can only be solved through an extensive manual search of the related repositories, *i.e.,* browsing of the git commit history, looking for keywords present in the vulnerability description. In the end, patches for 31 out of the 42 vulnerabilities were found.

3.3.2 Taxonomy of the Issues Related to Android Vulnerabilities

Once all the sought patches and bug reports collected, the classification of vulnerabilities according to their origin can start. Since classifying a vulnerability requires a deep understanding of it, we only consider vulnerabilities for which we are able to find related patch(es).

The goal of this classification is to obtain a taxonomy of the issues related to Android vulnerabilities. Many researches have been conducted to categorize bugs, *e.g.,* [21, 19].

However, these categories turn out to be quite impractical to our set of vulnerabilities as they either A) don't fit in a single category or B) are fitting in too many. To tackle this problem, we use mind mapping as a way to extend existing bug taxonomies. Mind mapping is a technique suggested by Vijayaraghavan *et al.,* [69] that constructs a taxonomy by incrementally considering one issue after the other. This process allows the taxonomy to evolve over time, *i.e.,* to improve its accuracy with each newly considered issue.

To correctly classify the considered issues, the following methodology is applied. First a complete analysis the commits and/or patches along with their related source code is made. Then bug and vulnerability reports and their available comments are investigated. Next, the information retrieved from the first two steps are crossed to try answering the following question: What was the problem, what kind of mistake led to the vulnerability and at which state of the project the issue emanates from. Finally, the category that fits best for the issue at hand is selected. Note that when an issue doesn't fit in an existing category, a new category is created.

3.3.3 Analysis

After classifying all vulnerabilities, we proceed with the analysis of the components they originate from and the changes that were needed to patch them.

To better understand the origin of the vulnerabilities, two aspects of the vulnerable components are investigated (1) their purpose and (2) their complexity.

Purpose of the Component: This information indicates which part of the Android system tends to have the most vulnerabilities. To this end, we design top level categories of purposes and classify the vulnerable components, according to them.

Complexity Analysis of a Component: Shin *et al.,* [169] in their study show that there is a correlation between code complexity and the appearance of vulnerabilities. Thus, we measure the MCC [119] of the functions that were changed to fix the vulnerability and compare it with the complexity of the other functions located in the same file to see if a similar pattern can be observed.

Once the origin of the vulnerability established, we investigate the complexity of the vulnerability itself. First an analysis of the patches and a synthesization of their contents is performed. Then the following metrics to measure the complexity of the vulnerabilities are used:

1. *Number of commits:* Number of commits required to correct the vulnerability.

2. *Number of modified files:* Number of modified, deleted or created files.

3. *Number of Changed Lines of Executed Code (CLoEC):* Number of modified, added or deleted lines of code. Empty lines and bracket lines are ignored and commands spanning over multiple lines are measured as a single line. This metric can also be referred as Code Churn in the literature with the difference that we don't take into account modification on commented lines.

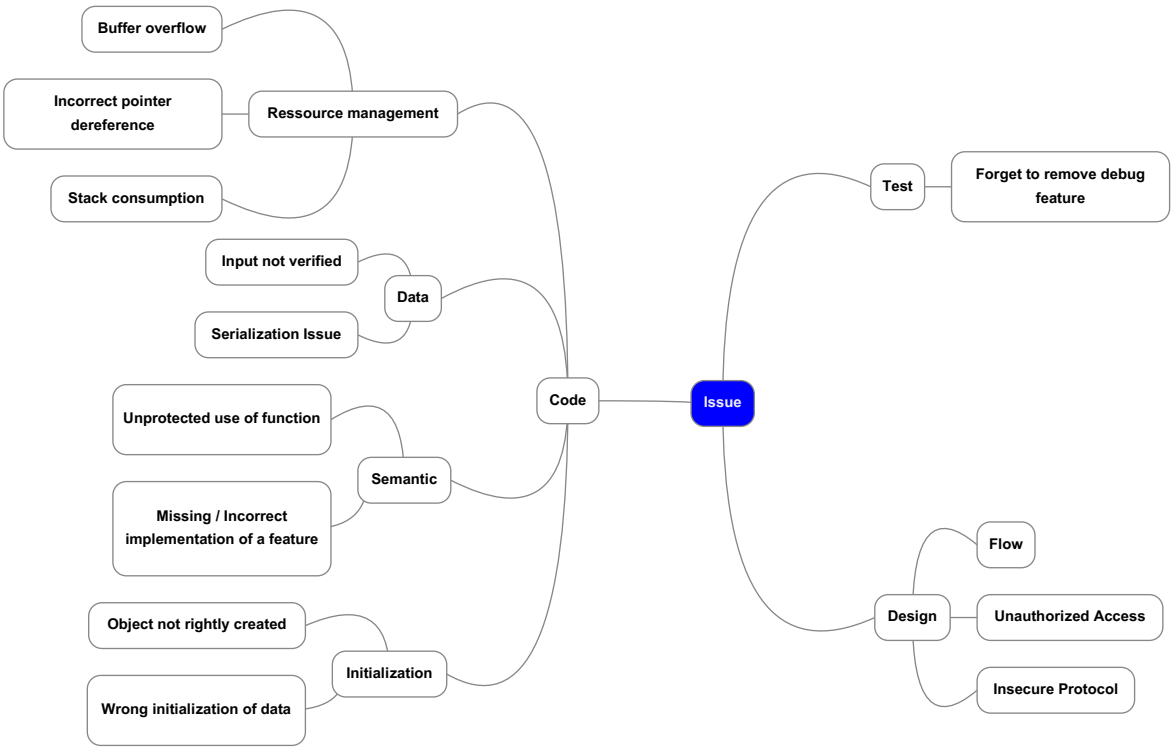


Figure 3.2: The issue taxonomy

Mind map of issues related to Android vulnerabilities.

4. *Cyclomatic Complexity of Change (CyCC)[36]* The CyCC metric measures the number of linearly independent sequences of changed statements from entry to exit in a changed program. It represents the least complex changes needed to remove the studied defect [36]. It is computed the same way that the Cyclomatic Complexity except that it uses the Change Sequence Graph (CSG [35]) instead of the Control Flow graph. The CSG is a control flow graph that only contains basic blocks that were changed. A Cycc superior to 2 indicates that the change was complex as it involves the addition or modification of more than two independent linearly path.

CLoEC metric represents a raw measure of the number of code changes. While, CyCC quantifies the complexity of changes by considering only the independent code places that the developer has to consider. Thus, CLoEC and CyCC are complementary indicators of the difficulty of fixing a vulnerability.

3.4 Results

This section presents the results obtained while applying the methodology presented in Section 3.3 on the dataset of Android vulnerabilities. First the causes of the vulnerabilities and introduced a taxonomy (3.4.1) are identified. Then information regarding the nature of the vulnerable components (3.4.2) are reported. Finally, in Section 3.4.3, the findings related to the changes necessary to patch Android vulnerabilities are detailed.

3.4.1 RQ1: Categorizing the Causes of Vulnerabilities

The analysis revealed a total of 43 different issues. Those were processed according to the methodology presented in Section 3.3.2 to build the mind map presented in Figure 3.2.

The suggested mind map is composed of three main nodes *Design*, *Code* and *Test*. They refer to the moment of the phase of development the issue likely occurred. The second layer of nodes corresponds to the cause of the issue. 8 types of nodes were identified, *i.e.,* Resource Management, Data, Semantic, Initialization Bug, Forget to remove debug features, Flow, Unauthorized Access, and Insecure Protocol. The third layer, which appears only in the Code part, enable a finer grained categorization of the issues. This layer is composed of 9 nodes, *i.e.,* Buffer overflow, Incorrect pointer dereference, Stack consumption, Input not verified, Serialization issues, Unprotected use of a function, Missing / Incorrect implementation of a feature, Object not rightly created, and Wrong initialization of data.

Table 3.1 shows the frequency of the categorized issues extracted from the mind map. Most of the issues are in fact originating from the coding part (about 70%), which is not surprising as other studies on faults and failures, *e.g.,* Hamill *et al.,* [71], showed that failures are heavily associated with coding faults. However, the number of issues related to design is surprisingly high (about 28 %). This can be explained by the number of issues related to permission handling. A deeper analysis shows that a great number of issues are in fact a missing or incorrect implementation of a feature, which occurs when developers misunderstand or misinterpret requirements.

The taxonomy allows to distinguish 13 kinds of issues that cause Android vulnerabilities. Those categories can be helpful during a code review to determine what to look for. Yet, this new taxonomy raises the question of what are the actual differences between the vulnerability origins and consequences ? In other words, do the introduced categories differ from the vulnerability types as reported in the NVD, i.e., CWE. After performing a comparison, only two of the suggested categories are overlapping with the CWE: (1) Input validation (CWE category) and input not verified (taxonomy) (2) Buffer errors (CWE category) and buffer overflow (taxonomy).

Except those categories, no overlaps have been observed. Yet, this overlapping is understandable as in these cases the CWE categories of vulnerabilities are in fact describing the origin of the vulnerabilities.

From those results, it is clear that the suggested *issue taxonomy* provides different information than the CWE field of vulnerability reports.

Table 3.1: Distribution of the issues in the taxonomy.

Origin	Kind	Number	Total (%)
	Flow	4	
Design	Insecure protocol	1	12 (27.90%)
	Unauthorized access	7	
	Resource Management		
	Buffer overflow	4	
	Incorrect pointer dereference	1	
	Stack consumption	1	
	Data		
	Input not verified	7	
Code	Serialization issue	1	30 (69.77%)
	Semantic		
	Unprotected use of a function	3	
	Missing/incorrect implem. of a feature	11	
	Initialization		
	Object not rightly created	1	
	Wrong initialization of data	1	
Test	Forgot to remove debug feature	1	1 (2.32%)

3.4.2 RQ2: Vulnerable Components

Role of the vulnerable component After analysing all the vulnerabilities, 9 kinds of top level components can be distinguished, Driver, Library, Messaging, Networking, Access Control, Browsing, Cryptography, Dalvik and Debug. The result of this analysis is presented in Figure 3.3. Hence, 9 vulnerabilities are originating from components related to web browsing, while 7 from components in charge of access control, which is directly linked to the numerous issues related to the handling of permission. Cryptography related components turn out to be as well quite vulnerable with 6 vulnerabilities. Interestingly those vulnerabilities are directly linked to a lack of understanding on how cryptography works and how it should be implemented.

Complexity of the vulnerable components

In the second part of the component analysis, the granularity is reduced to the file level. Hence, the cyclomatic complexity of all vulnerable functions and comparing it to the average complexity of all the other functions that are present in the same files is computed, for a total of 40 vulnerable functions from 3 programming languages, *i.e.,* Java, C and C++. As it is not possible to compute the complexity for vulnerabilities caused by errors within XML or RC files.

Figure 3.4a presents the results of this computation on a logarithmic scale. The horizontal axis corresponds to the average complexity of the vulnerable file and the vertical axis corresponds to the complexity of the vulnerable function. The gray line represents the $y = x$ function. Thus, points above this line indicates that the vulnerability was located in a function that was more complex than the other present in the file.

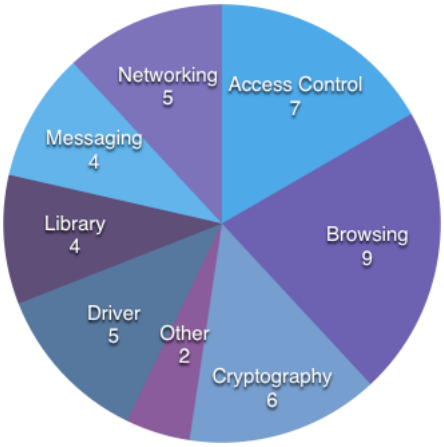
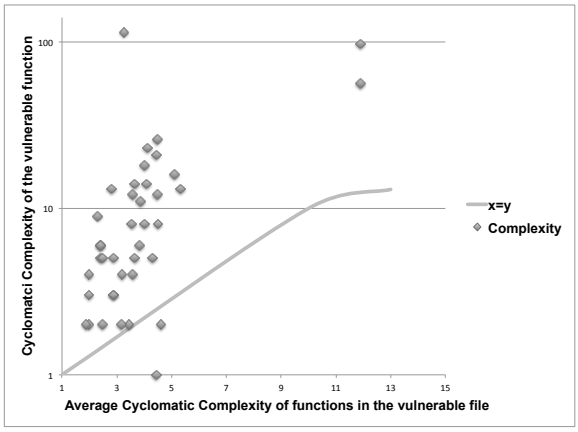
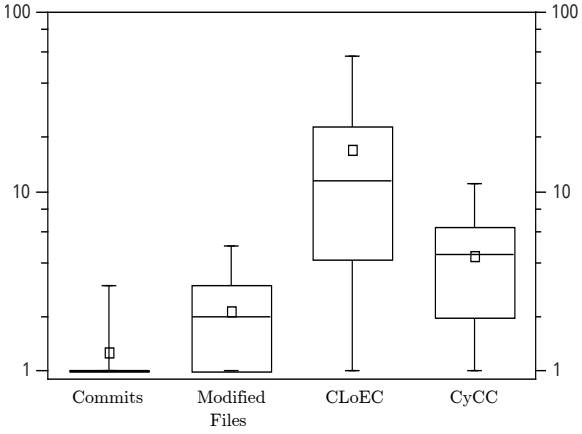


Figure 3.3: Android vulnerable components.



(a) Complexity of vulnerabilities.



(b) Complexity of the changes made.

Figure 3.4: Complexity of the vulnerable components and of the fixes.

Almost all vulnerable function turns out to have a higher cyclomatic complexity than the rest of the functions. In fact, only two cases are less complex. However, these represent functions that needed to be updated in combination with the one that actually contain the fix and were indeed complex. This fact confirms that vulnerabilities tend to appear in functions that have a higher complexity than the average.

3.4.3 RQ3: Fix Analysis

The last part of this study of Android vulnerability investigate the changes required to fix a vulnerability. These are described in terms of change types and complexity.

Kind of Change Table 3.2 shows the type of changes that were required to fix the vulnerabilities and their distribution. Among the patches that we were able to retrieve, 60% of them consisted in the addition of one or more lines of codes, 23% were modifications of existing codes and 17% were solely removing code.

Table 3.2: Distribution of the code changes needed to fix vulnerabilities.

Type	Kind	Description	Number	Total
Add	Condition(s)	If then else	19	36 (60.0%)
	Authorization	Permission in android manifest	2	
	Function + Use of it	Creation of a new function	5	
	Class + Use of It	Creation of a new class	2	
	Exception raise	Try catch or throw	3	
	Define or initialization	Variable	5	
Remove	Condition(s)	If then else	2	10 (16.7%)
	Use to a function	If the function was vulnerable	5	
	File	Deprecated and vulnerable files	2	
	Authorization	Permission in the Android manifest	1	
Modify	Call to a method	Changing arguments	8	14 (23.3%)
	Condition	Modifying expression	1	
	Function	Rewrite of a function	4	
	Complete rewrite	Rewrite of the all component	1	

In fact, one third of those changes are additions of one or more conditions, meaning that the developer's mistake was mostly failing to adequately check something in the code. Removing actions are frequently used to remove some "resources" that are not needed. The modification changes are rather simple ones like a call to a method, changing the arguments that are passed to it in order to avoid triggering a vulnerability. In only one case the issue required a complete rewrite. This means that in most cases vulnerability fixing doesn't imply large and deep refactoring of the code and can be done within the vulnerable function.

Complexity of the Changes If vulnerability fixes are usually simple, this doesn't mean that the fix actions are neither easy nor don't impact the code complexity. Figure 3.4b's boxplots presents the results of the 4 metrics introduced in 3.3.3 allowing us to investigate the impact of fixes. Regarding commits, in almost all cases only one commit was required to patch an Android vulnerability. On the side of file modification, on average, it involved the modification of 2 different files and 17 lines of codes. As of the CyCC metric, we found an average value of 4, which indicates that on average four linearly independent paths had to be changed. This can be interpreted as very complex according to the authors of the metric Böhme *et al.,* [36]. Thus if Android vulnerabilities can be solved at the function level, the efforts required to fix this function remain quite complex.

3.5 Discussion

The study presented in this chapter can be summarized by the following findings. Android vulnerabilities are always located among the most complex functions of the system, which confirm the hypothesis of Shin *et al.,* [169]. Their removing is complex and requires changes on code parts located on (in average) 4 linearly independent paths accounting for an average of 17 lines of code. In 50% of the cases, they are located in the Browsing, Cryptographic and Access Control components.

Moreover one out of four vulnerabilities (i.e., 25%) are due to a missing or an incorrect implementation of a feature, while 60% of the effort needed for removing a vulnerability is due to conditions adding.

3.6 Threats to Validity

A first threat to validity of this study is that required a lot of manual processing. Thus, it is not possible to ensure that all the produced results are fully accurate. As an example, a misinterpretation of the nature of a patch or a wrong categorization of an issue remains possible. This is reduced by reproducing the different analysis three times. Additionally, the data are made publicly available, thus, enabling replication and independent validation of the results.

Another threat to the validity lies in the use of the CyCC metrics. This metric is used as it has been used by the literature for the study of regression bugs [36]. However, it might not be appropriate for describing vulnerability complexity. To reduce this threat, it is combined with other metrics like CLoEC.

Finally, the biggest threat to validity is the fact that the only source of vulnerability used is the NVD and is limited to the vulnerabilities reported before 2014. Even if the set is consequent, its size remains small. It is not certain that the result would be the same with vulnerabilities reported after this date.

3.7 Conclusion

This chapter presented a deep analysis of the issues, components and patches related to Android vulnerabilities. Our study was performed on all Android vulnerabilities reported in the CVE-NVD database up to 2014.

This study introduces the challenges faced by practitioners faced when investigating vulnerabilities and that will be addressed in the next chapters. First of all, gathering information on vulnerabilities is requiring intensive manual work, which is impractical for any large-scale studies such as a vulnerability prediction model one. An automated way of collecting them is thus necessary. Yet, the development of an automated tool to gather vulnerability information might be considerably hindered by the problem of broken links in the NVD, especially patch ones. In addition, the two functions from Figure 3.4a which complexity were lower than the other are the perfect example of noise in a dataset, *i.e.,* functions that are linked to the vulnerable parts, but not mandatorily vulnerable on their own, which can disturb results.

4

Data7: A Vulnerability Fixes Collection Framework

Studies on security vulnerabilities require the analysis, investigation and comprehension of real vulnerable code instances. However, as introduced in the previous chapter collecting and experimenting with a sufficient number of such instances are challenging. To cope with this issue and address the first challenge of this thesis, we introduced in this chapter an extensible framework and dataset of real vulnerabilities, automatically collected from software archives called Data7. This framework supports the collection of information all reported vulnerabilities from 4 security critical open-source systems, i.e., Linux Kernel, WireShark, OpenSSL, SystemD including fixes for 1,600 out of the 2,800 vulnerabilities. The framework also supports the collection of additional software defects and is used as the basis of all subsequent studies.
This chapter is based on work that has been published in the following paper:

- *Enabling the Continuous Analysis of Security Vulnerabilities with VulData7 (SCAM 18)*
  M Jimenez, Y Le Traon, M Papadakis

Contents

4.1 Introduction

In section 2.1.8, we observed that there is no standard dataset for studies on VPMs, and more generally for the analysis of vulnerability. One of the reason lies in the fact that most authors are not releasing their dataset along with their study to enable replication, hence forcing any newcomers in the area to build their own dataset. Still, some publicly available datasets exist but they encompass their own issues. Perl *et al.,* [150] made their dataset available but this one is only containing suspected vulnerability contributing commits and not actual vulnerability fixes which reduces the field of its possible use. In addition, the link to this dataset present in the paper is not available anymore. Walden *et al.,* [184] released their dataset on PHP application which was reused by Zhang *et al.,* [199] but this dataset turns out to be relatively small. Alves *et al.,* [27] and Gkortzis *et al.,* [70] both released vulnerability datasets but those are processed datasets, *i.e.,* they only contain the metrics of the vulnerable files, not the actual vulnerable files, which hinder the use of new approaches. Moreover as most of the dataset, these datasets once released are never updated anymore or at the price of additional analysis. Hence, there is a need for an up-to-date, publicly available, large dataset of vulnerabilities providing information on the vulnerabilities as well as the patches that corrected them in their integrality.

In this chapter, we tackle the first challenge by introducing an extensible framework and dataset of real vulnerabilities, automatically collected from software archives called Data7 answering those 4 requirements.

Information on vulnerabilities: Chapter 3 highlighted the challenges faced by practitioners when trying to gather such information, *e.g.,* broken links. In this framework, we gather all required information on vulnerabilities of a given project by crossing information available in the NVD and git history. In particular the problem of dead patch link is tackled through the extraction of information on the patch directly from the link, *i.e.,* commit hash.

Large Dataset : To be useful, datasets require a large number of vulnerabilities. This reduces the number of subjects that datasets can be built upon to those having a high number of reported vulnerabilities. Additionally, as the use of few subjects threatens the external validity of empirical studies [189], the framework requires to work on more than one subjects. Ideally these subjects should share versioning technology and vulnerabilities reporting systems, so the collection of information can be performed in a similar fashion for all of them. With all of this in mind, we picked Linux Kernel, WireShark, OpenSSL and SystemD as initial subject for Data7 as they all rely on NVD and Git, but Data7 can be extended to any project using both. Mozilla Firefox that has been used as the subject by most of the studies on VPMs was ineligible to this last criterion as it still relies on SVN as VCS.

Updatable: Most datasets are never updated once released, yet new vulnerability emerged on a daily basis and files that were once considered as non-vulnerable might turn out to be in fact vulnerable. Hence, it is important to update datasets with the latest information available. Thus, Data7 is capable of updating with little cost its information, *i.e.,* only new and modified vulnerabilities are investigated.

Publicly Available: Data7 is available at `https://github.com/electricalwind/data7`

In summary, this chapter introduces Data7 which:

- provides a set of real vulnerabilities for 4 security critical systems. The current version includes 2,800 vulnerabilities and 1,600 patches.
- brings together related code, its commits and all related information, i.e., CVE number, vulnerability description, CWE number (if applicable), time of creation, time of last modification, CVSS severity score, bug ids (if existing), list of impacted versions.
- is automated. Once configured, it extracts and links information from the related software archives (through Git and NVD reports) to create a dataset that is continuously updated with the latest information available.
- includes all reported and mentioned (in the software archives) vulnerabilities. Special care was taken in order to make the dataset as complete as possible (with respect to what can be mined) by searching both sources of information (links on NVD and Git messages). As a result we managed to mine vulnerabilities with assigned CVE that have not yet been recorded in NVD.
- is flexible and easily extensible. Our framework links NVD with Git and thus, it involves little effort in configuring and importing data from additional projects. It includes all available processed and "raw" information (commit hashes, commit timestamps, commit messages and fixes - files in their states before and after fix), in order to be useful and extensible for research purposes. It also retrieves the complete related code bases to ease analysis.
- provides a friendly interface for retrieving the related information. It includes utilities (such as XML exports, Git utilities, CWE Importer and others) for common analysis tasks which eases the access and analysis of the set.

4.2 Information and Collecting Process

Data7 is a framework that brings together vulnerability reports, vulnerable files and their patches for a given project. The framework is automated, it retrieves, stores and updates the sought data with the latest available information. In short the main information that can be retrieved is the following: (1) CVE (2) Vulnerability description (3) CWE (if applicable) (4) time of creation (5) time of last modification (6) CVSS (7) Bug ids (if existing) (8) list of impacted versions (9) list of commits that fixed the vulnerability. These contain the *commit hash, timestamp and message*, and the *commit fixes* (files in their states before and after fix).

Table 4.1: Dataset Statistics

Systems	No Vulnerabilities	No Fixed Vulnerabilities	Average CVSS	Unique Vulnerable Files
Linux Kernel	2,082	1,202	5.41	1,508
Wireshark	531	265	4.99	221
OpenSSL	187	126	5.34	164
SystemD	9	5	5.76	5
Total	**2,809**	**1,598**	**5.38**	**1,898**

A high-level view of the Data7 architecture and process can be described as follows: For a given project P, Data7 clones in a local folder the git repository, connects to the NVD database and downloads all the available XML feeds for vulnerabilities for a given period of time (normally this should be in the range from 2002 to the current year). Then, Data7 parses the XML feeds and retrieves all vulnerabilities reported for the specified period of time. For each vulnerability, it retrieves and saves all declared links (links mentioning bug reports or direct links to fixing commits). Then, Data7 follows these links and retrieves the related commit information (for each vulnerability that had a link to a fixing commit). To account for missing links, the framework searches the version history of the project to identify (in the related commit messages) for a CVE Identifier or a bug id that was mentioned in the vulnerability report. Based on this information Data7 retrieves vulnerable and fixed versions.

To support continuous analysis, the framework can be automatically updated. Thus, it checks NVD for the latest information and updates its data. In case new data are there, Data7 will pull vulnerabilities reported for P and create a new entry if there is a new vulnerability or update as necessary. In a nutshell, for each vulnerability entry, the framework checks for new links or commit fixes and/or bug id. It then pulls the repository, retrieves the new (vulnerable) commits, checks for bug ids and CVE Identifiers and updates the dataset.

4.3 Subjects

At the time of writing Data7 contains data for 4 major security critical projects. While the framework is completely automated, we restrict the analysis to these projects because we are interested in collecting a large number of instances (in a per project basis) and the tool is working only with Git. Thus, we mined C programs, which tend to have more reported vulnerabilities. Nevertheless, the purpose of Data7 is to provide a framework to support the collection and mining of vulnerabilities.

We collected data for the following four projects:

- **Linux kernel**: started in 1991 as a hobby by Linus Torvalds. The Linux kernel is now shipped in billions of devices (embedded in all Android devices). It is the biggest OSS with more than 19.5 million lines of code and more than 14,000 contributing developers.

Table 4.2: Top-10 most frequent vulnerabilities

Each entry represents a pair of the form CWE id (frequency).

Rank	Linux	Wireshark	OpenSSL	SystemD
1	264 (318)	20 (136)	310 (32)	20 (3)
2	200 (219)	399 (108)	399 (28)	264 (2)
3	399 (212)	119 (98)	116 (17)	362 (2)
4	119 (204)	189 (51)	200 (15)	787 (1)
5	20 (161)	400 (14)	20 (12)	119 (1)
6	189 (106)	74 (9)	189 (11)	-
7	362 (89)	476 (8)	362 (5)	-
8	476 (45)	134 (5)		-
9	284 (45)	200 (4)	-	-
10	416 (28)	-	-	-

Table 4.3: Most frequent CWE definitions

CWE id	Description
264	Permissions, Privileges, and Access Controls
200	Information Exposure
399	Resource Management Errors
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
20	Improper Input Validation
189	Numeric Errors
400	Uncontrolled Resource Consumption ('Resource Exhaustion')
310	Cryptographic Issues
116	Improper Encoding or Escaping of Output
362	Concurrent Execution with Shared Resource and Improper Synchronization

While these numbers are important, they are not the main reason for choosing the Linux kernel as a candidate for our study. As mentioned earlier, the Linux kernel has to deal with many security aspects, it is the software with the second highest number of reported vulnerabilities (according to CVE). Another main reason behind this choice is the fact that the community behind the Linux kernel is well organized. This makes it relatively easy to get relevant and reliable information on vulnerabilities. A last criterion worth mentioning is the stability of the version control system, which is used in our study to gather the vulnerable files. In the past few years, many OSS adopted git as version control system, making links to a previous version control system as reported in the vulnerability reports invalid. As the Linux kernel community created git in 2005, the Linux kernel was the first to adopt it. This gives us access to over ten years of history to study. According to our data, the project has 2082 reported vulnerabilities.

- **Wireshark**: is packet analyser enabling the analysis of network traffic, protocols and interface controllers. It is mainly used for troubleshooting of related network issues and to support development. It is available on all operating systems and is open source. The project firstly named ethereal was renamed following a fork as Wireshark in 2006. Since then 531 vulnerabilities were reported for the project.

- **OpenSSL**: is a widely used library that provides implementations of the SSL and TLS protocols (used extensively in communications). The project code is not that big (it contains approximately 650k lines of codes) but due to its criticality [16] it is often subject to attacks. The project started in 1998 and migrated to Git in 2013. The migration was successful and no significant loss of information occurred making possible to access all the versioning information directly from the git history. Currently, the project involves 187 vulnerabilities.

- **SystemD**: is the service manager for the Linux operating system. As such, its main goal is to unify the services and configurations of the Linux systems. It is used in the Data7 tool as an example project. So far, 5 vulnerabilities have been reported for this system.

Overall, the descriptive statistics of our data are shown in Table 4.1. The table records details about the number of vulnerabilities (column “No Vulnerabilities”), number of vulnerabilities with available patches (column “No Fixed Vulnerabilities’), the average severity score of the collected vulnerabilities (column “Average CVSS’), the average score of the collected vulnerabilities with patches (column “Average CVSS Fixed”) and the number of unique vulnerable files involved (column “Unique Vulnerable Files”).

In total our data contain 2,809 vulnerabilities and for 1,598 of them we retrieved a patch. These account for a collection of 1,898 vulnerable files with an average severity of 5.34.

Table 4.2 records the 10 most frequent types of vulnerabilities (according to CWE categorization) per project. Each entry on this table represents a pair of the form vulnerability type (CWE id) and the frequency it appeared in the project. The list (description) of the vulnerability types (CWE id) is given in Table 4.3.

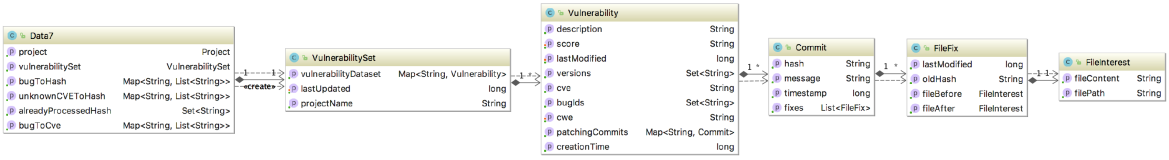


Figure 4.1: VulData7 API

4.4 Accessing the Generated Dataset

The generated dataset can be access in two ways, either through a JAVA API, or through a generated XML file.

4.4.1 Java API

Upon the creation (or the update) of the dataset, the user will receive a Data7 Java object. This object contains information about the project, some required data that are used by the tool to optimize its update action and the dataset in the form of a VulnerabilitySet Object.

Among the data used by the tool to optimize its update, two can be used for other purposes (i) a mapping of bug ids to commit hashes and (ii) a mapping of all CVE identifiers that were found in commit messages and are not present in the CVE database. While the first one (i) can be used to create a Bug Dataset (see Section 7.7 for details), the second one (ii) offers the possibility to observe vulnerable fixes before the release of the vulnerability report. As an example, in its latest run, our tool found that the commit [5] was made to address the CVE-2018-10840 which is not yet public. A glimpse at the commit message informed us that there was an issue with ext4 extended attributes. This list can thus be seen as a way to retrieve data related to all new vulnerabilities.

The VulnerabilitySet object is the dataset itself and is composed of the following elements: a list of every vulnerability ever reported for the chosen project (list of Vulnerability objects) and the time of the last update. A Vulnerability object contains all of the information mentioned in Section 4.2, which in essence includes the information that was found in the vulnerability reports. The Vulnerability object also includes a list of all the commits (Commit object) that were reported as fixing commits. A Commit object includes the hash, the message, the timestamp of the commit and a list of files that were modified by it (FileFix Object). A FileFix object records information on the time of last modification before the given commit and its corresponding previous hash as well as the file in its state before and after commit (FileInterest object). A FileInterest object contains the text of the file and its fullPath in the project.

All information related to the fields of each object that is accessible through the API is presented (as a UML diagram) in Figure 4.1.

4.4.2 XML Exporter

Once the dataset is created, the user is offered the possibility to export the data to an XML file. The generated XML file contains only vulnerabilities for which fixes were found. The file also includes all relevant information from the NVD database.

The schema of the generated file is presented in Listing 1.

Listing 1 XML schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <data7 last_updated="YYYY-MM-DD HH:mm:ss CEST"
       project="project name">
      <cve id="CVE-YYYY-XXXXXX" last_modified="timestamp">
        <cwe></cwe>
        <score></score>
        <description></description>
        <affectedVersions>
          <version></version>
        </affectedVersions>
        <bugs/>
        <patches>
          <commit hash="aaaaaaa" timestamp="xxxxxxx">
            <message></message>
            <files>
              <file>
                <before hash="aaaaaaaa" path="src/file.c">
                Content of the file
                </before>
                <after path="src/file.c">
                Content of the file
                </after>
              </file>
            </files>
          </commit>
        </patches>
      </cve>
    </data7>
```

4.5 Using the Framework

4.5.1 Installation, Generation and Export

The tool can be downloaded from GitHub and the latest version can be found at `https://github.com/electricalwind/data7/releases`. Data7 requires maven and Java (version 8 or higher). To proceed with the installation, the user should type mvn install in a terminal at the location of the project (where it was downloaded). The project can then be used from any maven project by adding the dependency presented in Listing 2.

Succeeding with the compilation, the user needs to define a path (to the place where the binary will be saved) and create an instance of an importer. These steps are quite simple as demonstrated on the following Java code (Listing 3).

As shown in Listing 4, to export to XML format, an instance of an exporter must be created before calling the XML export function (Listing 4).

4.5.2 Integrating other tool or database

The Data7 framework offers the possibility to export the data to another tool through listeners (which should be provided by the user). Indeed, when creating or updating a dataset, the user has the possibility to declare Listeners. These listeners should use the DatasetUpdateListener interface. Thus, a potential listener could use the notifications to populate an SQL database.

Listing 2 Dependency to add in pom.xml

```
<dependency>
    <groupId>lu.jimenez.research</groupId>
    <artifactId>data7</artifactId>
    <version>1.3</version>
</dependency>
```

Listing 3 Generating a dataset

```
ResourcesPath path = new ResourcesPath("Path To Save Tour Data
   into");
Importer importer = new Importer(path);
Data7 data7 = importer.updateOrCreateDatasetFor
                    (CProjects.LINUX_KERNEL);
```

Listing 4 Exporting to xml

```
Exporter exporter = new Exporter(path);
exporter.exportDatasetToXML(data7);
```

4.6 Additional Tooling

To ease analysis, Data7 includes three custom-made libraries. These include:

Git Utils: This library is coded in Kotlin and provides useful functions related to the mining of Git repositories. In short, the library provides methods related to the retrieval of files from specific commits, the retrieval of commits modifying a file, git-Blame, etc....

Misc Utils: This library has been coded in Kotlin and contains useful functions for common tasks such as downloading a file, unzipping a file, normalizing a (directory) path and getting the recursive list of directories.

CWE Importer: This library collects and reports data related to the CWE (types of vulnerabilities). The library can be invoked by calling *Importer.getListOfCWE();*. This call downloads (from NVD) the descriptions of the CWE, which are parsed and stored. Data related to the hierarchy of CWE types are also collected.

Further details about the utilities supported by the Data7 framework can be found in the website of the tool.

4.7 Limitations

Data7 is an ongoing project aiming at the automatic mining, analysis and evaluation of software defects and security vulnerabilities. As such, the current version has some potential limitations, which are discussed in this section.

The dataset is automatically constructed by mining software archives. This results in some noise in our data. We have no guarantee that what we retrieve is in fact vulnerability patches or that the vulnerabilities have been really fixed. Additionally, no attempt is made to prune or minimize the commits of the retrieved versions to the most likely causes. Another limitation regards the retrieved data, which may include some duplicated instances (due to commits residing in different branches). This fact depending on the performed analysis may be or may not be an important issue.

Process tackling these limitations are in fact introduced in the chapter presenting the VPM framework built upon this framework (8). Nevertheless, data provides by the Data7 framework is a good starting point to any Vulnerability analysis study.

4.8 Conclusion

In this chapter, we introduced Data7, a framework and dataset supporting the continuous collection and analysis of security vulnerabilities. Currently, the dataset involves 2,800 reported vulnerabilities, with 1,600 fixes, for 4 large security critical systems. A considerable effort was made in making this toolset automated, extensible and easy to use, with the goal of providing the community with the means to support research and analysis at a large scale. As the framework was built on top of Git and NVD it is simple to include additional open source projects. The addition of new projects requires a simple configuration (setting the appropriate links and paths), if the new projects use Git (with reported vulnerabilities on NVD) and a couple of extensions for other software archives. The dataset will be used as the basis of all following chapters, starting by the next one that proposed an analysis of the vulnerabilities of OpenSSL and Linux Kernel made on previous version of Data7.

Data7 is provided under the Apache Licence (Version 2.0) and is publicly available on GitHub: `https://github.com/electricalwind/data7`

5

An Automated Analysis of OpenSSL and Linux Vulnerabilities

In this chapter, a possible use of the data collected by the Data7 framework is presented. One of the intended use of the framework is indeed to better understand vulnerabilities, hence we investigate the characteristics of vulnerabilities from 2 of the framework subjects, i.e., the Linux Kernel and OpenSSL. In particular, we seek to analyse and build a profile for vulnerable code, which can ultimately help researchers in building automated approaches like VPM. Thus, in a similar but automated fashion as chapter 3, we examine the location, severity and category of vulnerable code along with its relation with traditional software metrics. Overall, the analysis of more than 2,200 vulnerable files accounting for 863 vulnerabilities is performed and reported.
This chapter is based on work that has been published in the following paper:

- *An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel(APSEC 16)*
  M Jimenez, M Papadakis, Y Le Traon

Contents

5.1 Introduction

Vulnerabilities remain nowadays a problematic issue that most developers fail to completely grasp. The difficulty lies in the fact that seeking for vulnerabilities requires an attacker's mindset [122] that can foresee and exploit weaknesses.

In Chapter 3 an analysis of the vulnerabilities of Android was presented and provided interesting insight on them. However, due to the considerable manual effort, the scale of the analysis remained relatively small. Chapter 4 introduced a dataset enabling to perform larger scale vulnerability analysis study. Thus, in the chapter, we exploit this dataset to broaden the understanding of vulnerabilities by performing an analysis of all vulnerabilities for which fixes exist from the 2 most security critical software of the set, *i.e.,* Linux Kernel and OpenSSL.

In this context, we investigate the metric profile of vulnerabilities according to their types, *i.e.,* CWE focusing on 4 properties: i) location, ii) severity, iii) code metric profile and iv) impact on this profile when the vulnerability is fixed. Overall, this study is based on data gathered by the data7 framework as of June 2016 and involves more than 2,200 vulnerable files accounting for 862 distinct vulnerabilities. In total 35 software metrics are computed and reported building the profiles. These profiles are then linked with the location, type and severity, *i.e.,* CVSS of the studied vulnerabilities.

After the analysis, it turns out that 20 different CWE types are used for Linux and OpenSSL vulnerabilities. Among those types, 9 are the most prevalent ones while only 3 and 2 are the most severe ones in the OpenSSL and Linux kernel, respectively. In addition, most of the vulnerabilities are located in 2 and 4 directories for the OpenSSL and Linux projects, respectively. Finally, the most interesting finding is that vulnerability types profiles differ depending on the studied project. This indicates that the conception of a generic VPM approach operating in a cross project setting would be rather hard, which is confirmed by several studies [161, 127]. In particular, our results suggest that future research should focus on building a "personalised" vulnerability prediction model for every type of vulnerability or by targeting the most critical categories.

Overall, the contributions of this chapter are the following:

- It presents the results of a study on more than 2,200 vulnerable files, which account for 862 different vulnerability reports from two securities critical systems, the Linux kernel and OpenSSL.
- It constructs a metric profile for every major type of vulnerabilities for both considered systems. This profile includes the location of the vulnerability, its severity, its code metrics and its impact when fixing the vulnerable code.
- It shows that metric profiles differ among the different types of vulnerabilities and projects.
- It introduces a new metric based on Graph Edit Distance (GED) to compute the impact of a fix the code.

5.2 Research Questions

Due to their importance, vulnerabilities are a popular research subject. The community has organized its efforts by creating a set of software weaknesses (namely CWE) and a scoring system for severity (namely CVSS), based on exploitability and impact metrics. An interesting starting point for this analysis is to identify the prevalence of the vulnerabilities according to their types and severity. Thus, we ask:

RQ1. Which are the types of vulnerabilities that are most prevalent in the Linux kernel and OpenSSL? Can we link categories with severity?

The answer to this question will provide better insight on data present in the dataset and reveals the types were vulnerabilities are more prevalent. It also reveals the categories with the most severe vulnerabilities.

The next step of regards the vulnerability location, *i.e.,* which part of the software system a vulnerability is originated from. Together with the categorization, this information will provide useful insights regarding the weaknesses of the different sections of the studied projects.

RQ2. Is the location of the vulnerable files linked to the vulnerability types and/or their severity?

After studying the vulnerabilities themselves, the next step is to analyse the vulnerable files, *i.e.,* files that needed to be modified to fix the vulnerability. As shown in chapter 2 this level of granularity for analysis is the most commonly used for VPM. Thus, the result of this analysis could help by assessing the prediction power of some characteristics of vulnerable files. This leads to the following question:

RQ3. What are the characteristics of vulnerable files per considered type?

Another interesting point to consider that was investigated in chapter 3 is the extent to which a vulnerability fix impacts a file. For example, are those fixes complex or simple? Can we observe different patterns when fixing same specific types of vulnerabilities? All these concerns guide us to our next research question:

RQ4. What is the impact of fixing a vulnerability on the metric profile we use?

An analysis of the impact could help to flag some commits as vulnerability fixes, as well as provide some insights on the vulnerability fixes.

Table 5.1: Vulnerability Dataset Statistics

	Linux kernel	OpenSSL
Num of Vulnerabilities CVE	768	95
Num of Commits	899	382
Num of vulnerability types (CWE)	20	11
Vulnerable Files/ Unique	1615/951	619/102

5.3 Methodology

5.3.1 Dataset

This study uses all the fixes retrieved by the data7 framework as of June 2016. Table 5.1 presents the statistics of this dataset. Overall, Data7 managed to collect fixes for 52% of the CVE from the Linux kernel, and 59% from OpenSSL. The table records the number of commits, the number of different CWE and the number of vulnerable files that were retrieved. Note that a vulnerability can be fixed by more than one commit and a file can be vulnerable several times.

5.3.2 Characterization of Vulnerable Files and Fixes

To build the profile, 35 metrics are considered. Those metrics are similar to the one suggested by Shin *et al.,* [167] to build their VPM and are categorized as follows:

Basic Metrics: lines of code, blank lines, commenting lines, comment density, preprocessor lines, number of variable, number of declared functions. These metrics provide information regarding the profile of the vulnerable files.

Code Metrics: all variants of cyclomatic complexity (strict, modified and standard), essential complexity, maximum nesting, fan in and fan out. Note that these are function-level metrics and we are working at the file level so we computed for each file the metrics on all the function and kept the maximum, average and sum values. These metrics characterize the structure of the code and their definition can be found at [17].

Code churn: number of changes and number of lines added, deleted, modified in the history of the file.

Developer history: number of developers currently working on the file (git blame), number of developers that have worked on the file.

These metrics are computed through an especially developed software named FileMetric presented in Section 5.6.

To measure the impact of a vulnerability fix, the delta of all the previously suggested metrics between the vulnerable file and the fixed one is measured. This provides information related to the nature of the patch that fixes the vulnerability. Yet, it does not give a general overview on the impact of the change on the code. In view of this, we introduce a new metric detailed in the following subsection.

Table 5.2: CWE appearing in this chapter

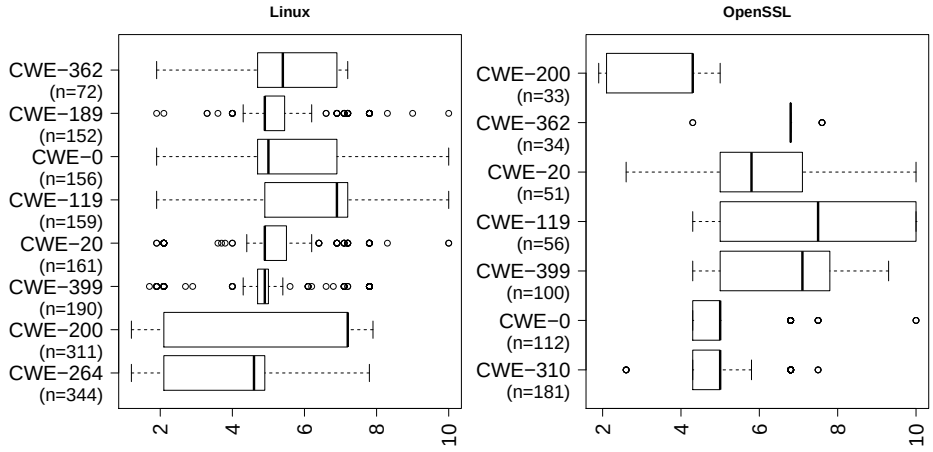
CWE	Definition
20	Improper Input Validation
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
189	Numeric Errors
200	Information Exposure
264	Permissions, Privileges, and Access Controls
310	Cryptographic Issues
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
399	Resource Management Errors
0	Vulnerabilities without a CWE Number

5.3.3 GED

Measuring the impact of a vulnerability fix is not an easy task. Making a 'diff' on the fixed and vulnerable files, which is a common practice, only reveals changes at the line level and does not consider the control flow. To measure the impact on the program flow, we compute the control flow graph (CFG) of the related functions before and after the modifications. Then, we compute a GED between the two graphs. Graph edit distance was first formalized by Sanfeliu *et al.,* [158] and is used to measure the similarity between two graphs. A survey on its possible use has been conducted by Gao *et al.,* [64]. The main idea here is to evaluate the minimum edit "cost" of going from the vulnerable CFG to the fixed CFG, by attributing different costs for an edge or a vertex of the graph replacement, deletion or insertion. For our study, we chose to attribute a value of 2 for replacement, 1 for insertion, 1 for deletion. This measure will indicate to what extent the patch modified the control flow of the function. As this measure is also a function level metric, we compute the GED for all functions of a file and sum them up after. The GED of an unmodified function being 0.

5.3.4 Experimental Process

Once all vulnerable files are collected by the Data7 framework, we compute the metrics both in the state before and after the vulnerability fixes. We then compute all deltas and the graph edit distances. Once in possession of all the metrics, the results are analysed by grouping the vulnerable files according to their CWE types.



(a) Linux Kernel (b) OpenSSL

Figure 5.1: Severity (CVSS score) per type of vulnerability (CWE)

n indicates the number of vulnerable files in the dataset for this category.

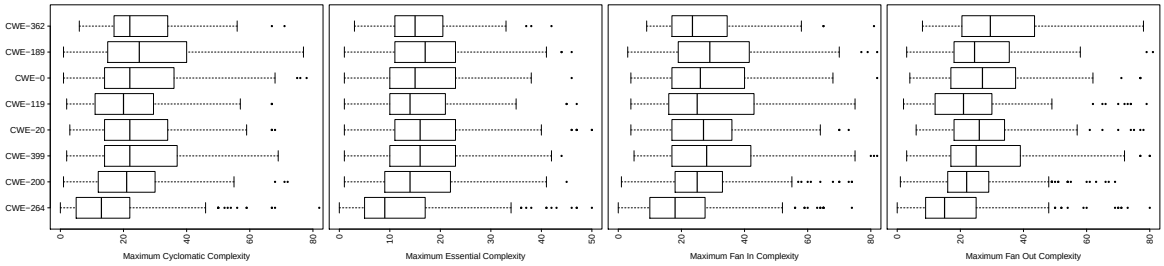


Figure 5.2: Maximum complexity of Linux Kernel vulnerable files

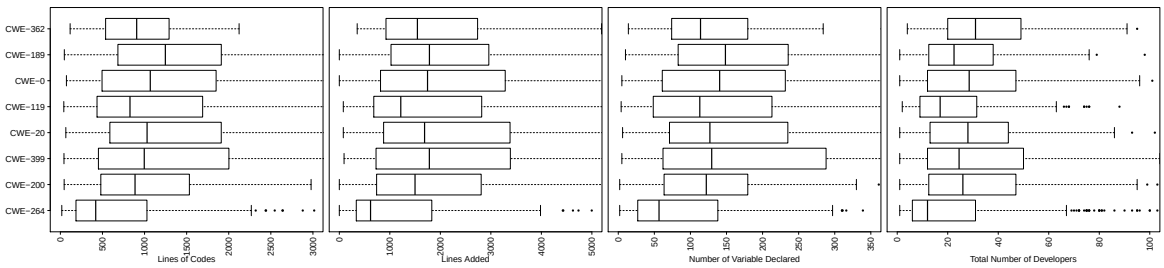


Figure 5.3: Metrics of Vulnerable Files for the Linux Kernel

Table 5.3: Location of the vulnerable files in OpenSSL

Directory	Number of Vulnerable Files	CVSS Score	most represented CWE (number of vulnerable files)
apps	9	6.77	CWE-20 /CWE-399 (4)
crypto	286	5.82	CWE-310 (90)
engines	9	6.67	CWE-399 (6)
ssl	314	5.52	CWE-310 (90)

5.4 Results

5.4.1 RQ.1: Types of Vulnerabilities and Severity

When analysing the data, it turns out that specific types of vulnerabilities are rather scarce. As the goal of this analysis is to identify trends, we filter them out and focus only on the most common ones. Thus, we set a threshold of 50 vulnerabilities for the Linux Kernel and 30 for OpenSSL. This leave us with 9 types of vulnerabilities reported in Table 5.2 that are the most prevalent in the studied systems.

After identifying the vulnerability types, the focus is brought upon severity. Figure 5.1 presents the severity per type, ordered (from top to bottom) by the number of vulnerable files present in each category. Quite logically, the most represented categories (order) of one system are not the ones of the other. This is explained by the functionality differences of the studied systems. With respect to CVSS score the most consistent category over the two software systems is the "Improper Restrictions Of operations within the bounds of a memory buffer" (CWE-119) which reach a high average severity of 7 for both systems.

Overall, the results show that out of the 20 types of vulnerabilities, 9 are prevalent and among them the most severe ones are the CWE-200 and CWE-119, for the Linux Kernel and CWE-119, CWE-399 and CWE-362, for OpenSSL.

5.4.2 RQ.2: Location of the Vulnerabilities

Table 5.3 and 5.4 present the results of the investigation on the whereabouts of vulnerable files. It is important to note that only the main directories appear here, *i.e.,* depth of 1. This choice was made in order to maintain consistency between OpenSSL and Linux due as OpenSSL file structure has fewer branches. An interesting result is that OpenSSL vulnerabilities are either emanating from the crypto directory or the ssl one. This is linked to the fact that the most represented category is related to the "Cryptographic" issues (Figure 5.1b). Interestingly, the most severe vulnerabilities are located on the "apps" directory, but it involves a much lower number of vulnerabilities. Overall, among the two most vulnerable directories (crypto and SSL), those in 'crypto" directory are more severe.

Table 5.4: Location of the Vulnerable Files in the Linux Kernel

Directory	Number of Vulnerable Files	CVSS Score	most represented CWE (number of vulnerable files)
arch	241	4.46	CWE-264 (120)
crypto	81	2.35	CWE-264 (66)
drivers	239	4.95	CWE-119 (55)
fs	339	5.60	CWE-200 (95)
kernel	117	5.92	CWE-200 (49)
mm	71	5.83	CWE-264 (22)
net	423	5.26	CWE-200 (76)
security	38	5.85	CWE-119 (12)

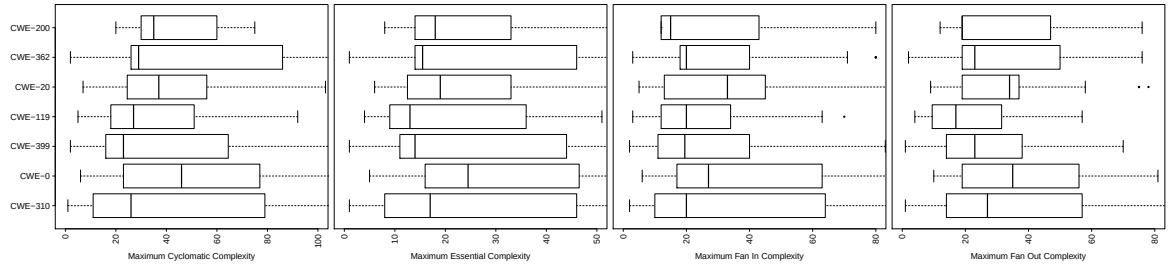


Figure 5.4: Maximum complexity of OpenSSL vulnerable files

Regarding the Linux kernel, the directory with the most vulnerable files is the "net" directory, which is in charge of the network functionalities and has half of its vulnerabilities categorized as "Information Exposure". Interestingly, the problem of permission and privileges seems to occur mostly in the "arch" and "crypto" directories, whereas buffer errors are mostly present in the "drivers" and "security" directories. Looking at the severity, the "kernel" directory seems to be the more problematic followed closely by the "mm" responsible for the memory management one.

Overall, the results suggest that the analysis of the vulnerability history of a project can provide interesting insight on the specific problem of a given part of software.

5.4.3 RQ.3: Characteristics of Vulnerable Files

Only the 8 software metrics possessing the strongest discrimination power are presented in this section to improve readability.

Linux Kernel Figures 5.2 and 5.3 present the results related to the 8 selected metrics for the Linux Kernel. We observe that vulnerable files related to "Permissions, Privileges, and Access Control" (CWE-264) contain less complex functions than any other type of vulnerabilities. They also have fewer lines of code, lines added, variables declared and a smaller group of developers working on it. This indicates that files dealing with permission rights do not require complex algorithms and are less likely to be modified once been written.

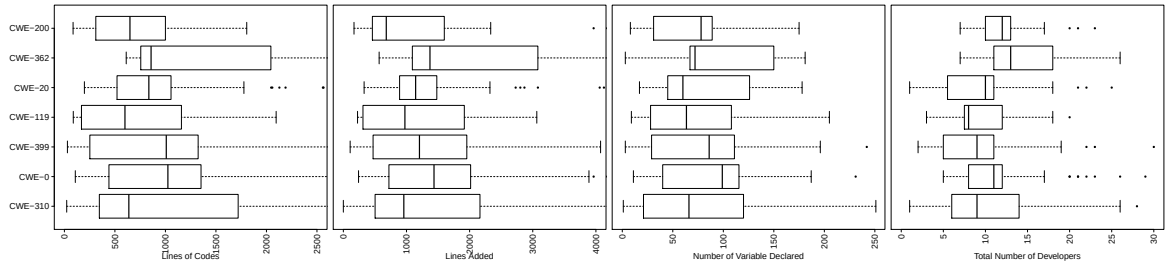


Figure 5.5: Metrics of Vulnerable Files for OpenSSL

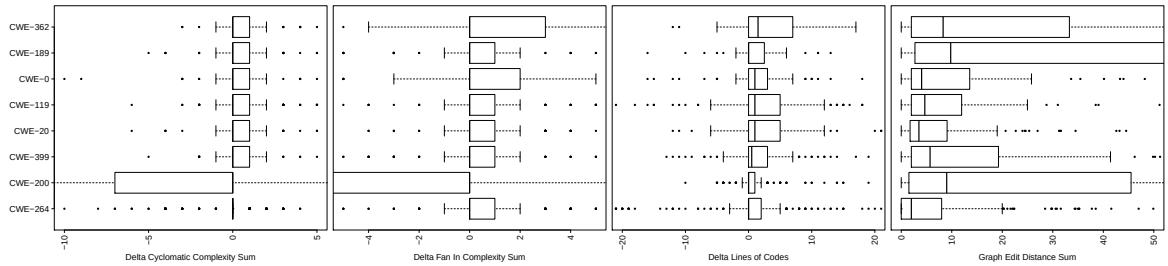


Figure 5.6: Impact of Fixes for Linux kernel

Vulnerable files related to "Numeric Errors" (CWE-189) appear in the most complex functions (according to "Cyclomatic Complexity", "Essential Complexity" and "Fan In"). The same files are also having the highest number of lines of codes and declared variables. This may suggest that numeric error, *i.e.,* improper calculation or conversion of numbers, are more likely to be found in complex functions than in simple ones.

High "Fan Out" values, *i.e.,* the number of called functions plus global variables, and high number of developers result in "Race Conditions" problems. This indicates that code parts related to concurrency also has high "Fan Out", which in term requires special attention. These files are also important and seem to have a central interest in the project as there is also a high number of developers working on them.

We also observe that vulnerable files without a category (CWE-0) have average values for all metrics, which may suggest that these vulnerabilities are indeed a mixture of categories waiting to be categorized.

OpenSSL Figures 5.4 and 5.5 present the results for OpenSSL. A quick comparison with the values obtained from the Linux Kernel indicates that there are differences between the maximum cyclomatic and essential complexity while interestingly lines of code remain in the same range of values.

The category of vulnerable files with the higher score in most metrics are those that are uncategorized vulnerabilities (CWE-0) which might indicate that in the case of OpenSSL uncategorized vulnerabilities are of another category than the existing ones.

Regarding "Race Conditions" (CWE-362) vulnerable files, we observe that they occur when many developers are involved (like in the Linux kernel). While in the case of "Cryptographic Issues" (CWE-310) which is the most represented type of vulnerabilities in OpenSSL, we observe a low maximum complexity compared to other types of vulnerabilities, except for "Fan Out" result.

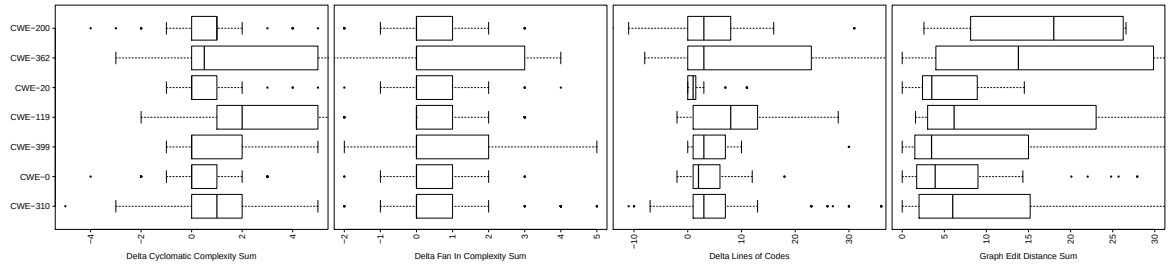


Figure 5.7: Impact of Fixes for OpenSSL

5.4.4 RQ.4: Impact of Vulnerability Fixes

Linux Kernel Figure 5.6 presents the results for 4 of the metrics. Regarding the complexity, the impact of a fix is similar for most of vulnerabilities except for "Information Exposure", which reduces the complexity. This result is surprising, as one could expect an increase of complexity from additional checks. Looking at graph edit distance metrics, fixes for CWE-264 are the ones that are less impacted whereas the fixes for " Numeric Errors" are the ones with the greatest impact. This was expected as those vulnerabilities were the ones with the higher complexity observed for most metrics.

OpenSSL Figure 5.7 shows the OpenSSL results. A first observation is that there are much larger variations from one category to another than in the case of the Linux kernel. Fixes of CWE-119 files seem to have a larger impact on the cylomatic complexity increasing it by a value of two, on average, and the highest delta in lines of codes. In the "Fan In", i.e., the number of function calls plus global variable reads, the highest variation is observed for CWE-362 fixes. Regarding graph edit distance, the fixes with the most impact on the CFG are the ones from "Information Exposure" (CWE-200) files followed by "Races Conditions" CWE-362. Interestingly, these two types are also among the top ones of the Linux kernel for this metric.

5.5 Threats to Validity

Regarding threats to construct validity, the first one is that the categorization of vulnerabilities might be inconsistent, since this is a manual process performed by different people. It is likely that different points of view on which category a vulnerability belongs to might exist. In addition the hierarchy of the category might be problematic as well. However, given the well-organized community behind these projects and the high attention they received, this concern should be limited to a small percentage of the vulnerabilities.

In this study, a vulnerable file is a file that had to be modified to fix a vulnerability. Thus, all the files from a fixing commit were added in the dataset. Yet, some commits might include fixes for some other things than the said vulnerability, hence adding noise to our dataset. As this is against common practices, it should not impact the results too greatly.

A third concern is the fact that data taken from the Data7 framework are automatically gathered and not curated. Hence if a vulnerability has two identical fixing commit due to branching, we might end up with duplicated data. After analysis, this problem only occurred a couple of times and can be solved by using some heuristics that will be presented in 8.

Regarding internal validity, potential bugs in the implementation may also influence the results by providing incorrect measures. To reduce this threat, the implementation was carefully tested and verified. Moreover, as metrics for all files are computed the same way, this should not have much influence on particular reported profiles as similar variations, in the measurements, should be observed with other tools.

Finally, regarding external validity, this study is limited to two open source software systems written in C. Thus, the results might not be generalizable to systems written in other languages. This is partly indicated by the results which show that different profiles exist, between the two projects, for the same types. However, these two projects are typical examples of safety-critical applications.

5.6 Additional Tooling

To perform this study a specific library named FileMetrics was developed. The FileMetrics library provides utilities to computing common metrics of a C file from Java programs. The library allows to compute all metrics presented in this study and more generally those from the study from Shin *et al.,* [167]. There are several reasons for the creation of this library. First, there is no other existing library to compute this metric in Java. Second, it is not possible to rely on the tool use by previous studies such as the ones from Shin *et al.,* [169, 167, 171], *i.e.,* the Understand tool, as it requires the full project and we are only in possession of the commit. In addition, this tool is proprietary and requires heavy tooling to run which prevents its good integration. Still, all metrics from the library that are common with the Understand tool are computed according to the definition found in the Understand tool to obtain similar result.

To perform such analysis on C files, the first step is to retrieve the AST from the file, we achieve this by using the Joern tool developed by Yamaguchi *et al.,* [192]. Then the metrics can be computed and in case Control Flow Graph (CFG) is required we use the corresponding utility from Joern.

The library is written in Kotlin and take the file, *i.e.,* its content as String, as an input.

Listing 5 Using File Metrics in Kotlin

```
val cm = CodeMetrics(fileContent)
cm.cyclomaticComplexity()
```

Listing 5 present how to use the library. Once the constructor the following metrics can be computed: *Blank Lines*, *LoC*, *Preprocessor Lines*, *Lines of comments*, *Comment/LOC ratio*, *Number of Function*, *Number of Variable declaration*, *Cyclomatic Complexity of all Function (Standard, Strict, Modified)*, *Essential Complexity of all Function*, *Fan in/out*.

The library can also be used to retrieve information in the file at different granularity. For example, at the function level, the library can return the list of parameters of the function, calls, assignment and declarations while at the file level, it can return the list of function name and global variable.

Regarding the computation of GED which is important in this study, the library integrates the code of the Hungarian Algorithm by Hakon Drolsum Rokenes available on Github [12] and offer to compute it with and without the Levenshtein distance.

FileMetrics is available at `https://github.com/electricalwind/FilesMetrics`.

5.7 Conclusion

In this chapter, we analysed the characteristics of the vulnerabilities from Linux kernel and OpenSSL for which the Data7 framework was able to retrieve fix. We found that the profile of vulnerabilities varies depending on their type. This suggests that VPM could be tuned to target specific types of vulnerabilities instead of considering just vulnerability. Yet this might be complicated due to the needle effect. Indeed breaking the already small group of vulnerability models can train on to even smaller subgroup would considerably impact the performance.

Overall, the analysis of the 2,200 vulnerable files, related with 862 vulnerabilities reveal that 20 types of vulnerabilities are used for those projects. 9 of them are the most prevalent ones and only few are really critical (3 for OpenSSL and 2 for the Linux kernel). These results suggest that building specialized models targeting these critical types of vulnerabilities would be of interest. We investigate this specific case in Chapter 9.

Another important finding is that the profiles are system-specific, which means that it is hard to draw conclusions on vulnerability types since vulnerabilities have different profiles in the two studied systems. This suggests that the use of prediction models that are trained on one project may fail on another and hence the creation of cross-project vulnerability models, as attempted by existing approaches [167, 161, 127], with reasonable performance seems to be quite hard if not impossible.

Additionally, the results show that the location of the vulnerable files can provide useful information on where and which type of vulnerabilities to look for. Also our results suggest that vulnerability criticality is related to its location. Interestingly, file location is not really considered by any of the existing vulnerability prediction methods. A possible reason for this is that location is a nominal property linked to the software system under analysis and can only be used for specific project prediction and not for cross system ones.

Finally, the results indicates that fixing vulnerabilities in Linux does not involve many changes (from source code editing point of view). While in OpenSSL the fix process is more complex than in Linux and requires many changes in different parts of the code.

6

Wrapping Up Part II

To create or understand the results of VPMs, the first important step is to better understand vulnerabilities and the reason why the suggested approaches could work. Thus, in this first part, the emphasis was put on the analysis of Software Vulnerabilities. In the first Chapter (3), an analysis of Android Vulnerabilities was presented. This analysis focused on specific properties of vulnerabilities such as the origin, the complexity of the vulnerable components and the location of the vulnerability. Performing this investigation highlighted one of the main challenges practitioners faced when attempting such tasks, i.e., gathering enough and relevant data. This challenge is not limited to the analysis of vulnerability but impact all fields of research targeting vulnerabilities, starting by VPM. To tackle it this, in the second Chapter (4), a framework that automatically collect vulnerability fixes was introduced. This framework is fully automated, extensible and publicly available. The dataset generated by this framework is among the largest of its kind and can be continuously updated with the newest data available. Then in the last chapter of this part (5), an analysis of the vulnerability found by the framework is performed. This analysis focused on the kind and severity of the vulnerability and led among other findings to the discovery that these properties are system-specific. In the next part, the emphasis will be put on the replication and analysis of existing VPM approaches in order to tackle the second challenge of this dissertation.

Part III

Investigating Vulnerability Prediction Models

7

A Replication Study of Existing VPMs Approaches on the Linux Kernel

Over the past decade, several VPM approaches have been suggested, such as ones based on Code Metrics or even Text Mining. Yet as observed in Chapter 2 no external study tried to compare the different approaches on a common ground. In order to address the second challenge, we aim in this chapter at making a reliable replication and comparison of the main approaches. Thus, we seek for determining their effectiveness under different scenarios. To achieve this, we rely on the data gathered by the Data7 framework for the Linux Kernel as of June 2016. Based on this, we then built and evaluate prediction models. Overall, we observe that an approach based on the import or function calls performs best when aiming at future vulnerabilities, while a text mining approach works best when aiming at random instances. We also find that models based on Code Metrics perform poorly.
This chapter is based on work that has been published in the following paper:

- *Vulnerability Prediction Models: A case study on the Linux Kernel (SCAM 16) M Jimenez, M Papadakis, Y Le Traon*

Contents

7.1 Introduction

In Chapter 2, a review of all studies on VPM was performed. One outcome of this review is the absence of a study that compared the different suggested approaches. Ideally, such a comparison between approaches should be performed by a third party, *i.e.,* researchers that are not involved in the development of one of the approaches. Yet, the only real comparison study is from Walden *et al.,* [184], which introduced the Text Mining approach. In addition, their study only compares two approaches, *i.e.,* Code Metrics and Text Mining. On the other hand, studies introducing new approaches tend not to make reliable and comprehensive comparison with previously introduced ones and are often not releasing their dataset.

Therefore, in this chapter we aim to replicate and compare different VPMs approaches. As we plan of using the data gathered by the Data7 framework to perform this study, we select among the possible approaches the three main ones, *i.e., Software Metrics* [167], *Text Mining* [161] and *Includes and Function Calls* [133] ,that are not requiring additional data such as crash logs [177] or result from ASA [65].

To perform this study, the choice is made to rely solely on data from the Linux Kernel as they are the most numerous. In addition, as shown by the results of Chapter 5 and previous studies it is rather unlikely that the models will work in a cross project prediction setting. Hence it is better to focus on a single project at a time. The data gathered by the Data7 framework is used to create two different settings one "experimental" emphasing on the ability of the tested models to distinguish defects from vulnerabilities, and one "realistic" exploring the performance of the models in a more realistic context, *i.e.,* the proportion of vulnerable files is approximately equal to the proportion reported on the Linux release.

In line with previous studies, we make the evaluation at the file granularity level as it was confirmed as actionable by Microsoft Windows developers [126].

The study aims at making a comprehensive evaluation of the studied approaches by reliably setting the training and evaluation datasets. Thus, the investigation is based on two scenarios, one that splits the training and evaluation sets at random and a "practical" one, based on time, where with respect to given reference time points (release times), we train on the past vulnerabilities and to predict (evaluate) the future ones. Overall, the study involved a set of 1,640 vulnerable files, 54,000 clear files and 870 experiments.

The results show the importance of the element selection when building a dataset. Indeed, models can perform better or similarly in a largely unbalanced dataset, which is closer to the reality, than in a balanced dataset composed of similar but distinct elements, *i.e.,* , bugs and vulnerability.

Overall, the data suggests that the variations on the studied application scenario can greatly influence the performance of the prediction approaches. In our two scenarios, we find that when aiming at future vulnerabilities, the approach based on includes and function calls is performing better, while when aiming at random instances of vulnerabilities irrespective to time, text mining outperforms the other approaches.

In summary this chapter makes the following contributions:

- It performs an exact independent replication of three of the main vulnerability prediction approaches in the context of the Linux kernel, using both "experimental" and "realistic" datasets.
- It investigates the predictability power, using past data to predict future ones, of the three main vulnerability prediction methods.
- It provides evidence that certain parameters and evaluation choices can have a major impact on the drawn conclusions. Therefore, future research needs to consider them in their evaluations.

7.2 Research Questions

The goal of the present study is to replicate and compare three of the main VPMs approaches in the literature, *i.e.,* software metrics [167], text mining [161] and includes and function calls [133]. Interestingly, these have never been neither replicated nor compared; each study has been evaluated on different contexts and custom datasets. Thus, the need to deal with their external validity and comparing them on a large and reliable "ground truth" dataset is evident.

Arguably one of the most important questions in predictive modelling is the ability of the developed models to identify, among several elements, the ones that it seeks for. In our context, VPMs should be able to distinguish between vulnerable and non-vulnerable files. Among non-vulnerable files some are closer to vulnerabilities, *i.e.,* buggy files. Indeed, vulnerabilities are often referred as security bugs and thus considered as a subgroup of bugs. Hence, it is interesting to determine if the approaches are able to determine this subgroup, *i.e.,* vulnerabilities, or are just pointing toward the upper group.

Thus, the first research question investigates whether the studied approaches can distinguish the vulnerable from buggy files.

RQ1. Are the vulnerability prediction models capable of distinguishing between vulnerable and buggy files?

A positive answer to this question will provide a good indication if the studied methods are of any value in our context.

Being able to distinguish vulnerable files does not imply that the model is actually useful for developers. This was investigated by Morrison *et al.,* [126] who found that when a tiny proportion of files is vulnerable, the usefulness of the prediction models is hindered. Therefore, we seek to investigate the effectiveness of the examined approaches under cases that are closer to reality, *i.e.,* when the proportion of vulnerable and non-vulnerable files in the studied data set are close to the ratios that are found in the Linux kernel (3% of files have a vulnerability history). Thus, we ask:

RQ2. What is the discriminative power of vulnerability prediction models in distinguishing between vulnerable and non-vulnerable files in a realistic environment?

This question is as an attempt to investigate the actual prediction power of the studied approaches. Evidently, a high (respectively low) accuracy indicates a relatively good (respectively bad) prediction. Also, the difference between the results of RQ2 from those of RQ1 demonstrates the impact of the datasets (proportion of vulnerable files) on the discriminative power of the models.

To address these two research questions, two distinct datasets named "experimental" and "realistic" are created. These datasets will be presented in Section 7.3.

Although in RQ2 we use what we call the "realistic" dataset, this does not accurately assess the predictability power of the developed models in identifying future vulnerabilities. In other words, we need to investigate the extent to which future vulnerabilities can be captured by the developed prediction models when trained on past data. Therefore, we investigate the ability of the models to capture the vulnerabilities of the different Linux kernel releases using the data of the past releases.

RQ3. How effective are the vulnerability prediction models in predicting future vulnerabilities when using past data?

The answer to this question provides a complete picture regarding the practicality of the approaches. Additionally, a relatively good accuracy on the predictions will indicate that vulnerabilities share the same characteristics over time, since they are captured by the prediction models.

Up to this point, the discussion is solely based on the effectiveness perspective. However, we have left aside any discussion regarding the cost of each method. This information can be useful for researchers or practitioners dealing with frequently changed data. Thus, our last research question evaluates the time and memory needed to train and develop each of our models.

RQ4. What is the cost in terms of memory and time consumption of building the studied vulnerability prediction models?

7.3 Dataset

As explained in Section 1.3.3, to perform any VPM study a large and reliable dataset is mandatory. Thanks to the Data7 framework this is not an issue. Still the Data7 framework only collect vulnerability fixes and is thus not directly usable for Vulnerability Prediction Modelling. This section review the procedure that was used to create suitable datasets.

To answer the different research questions, we built two distinct datasets: one corresponding to RQ1, *i.e.,* the "experimental" dataset, which is composed of a slightly unbalanced set of vulnerable and likely to be buggy files, and another one designed for RQ2, the "realistic dataset", which is close to the practical cases (fully unbalanced instead of slightly). To answer RQ3, we used both datasets.

7.3.1 Experimental Dataset

This first dataset is created to determine whether models are able to distinguish vulnerable files from closely related ones, *i.e.,* likely to be buggy but non-vulnerable.

To gather buggy files, we rely on the bug collector extension of the Data7 framework that is presented in Section 7.7. It is noted that even if the major part of these gathered files are buggy, some might not. Thus in the following we refer to those files as "likely to be buggy". Overall we collected about 4,900 likely to be buggy files, which put together with the 1,640 vulnerable files accounting for 743 vulnerabilities found by the Data7 framework, resulted in a proportion of the vulnerable files of 20%.

7.3.2 Realistic Dataset

This dataset is designed to investigate whether the models are able to flag vulnerable files under a realistic setting, *i.e.,* in an environment where vulnerable files would be uncommon. This implies the creation of a largely unbalanced dataset. After some preliminary analysis, it turns out that about 3% of the files in Linux have a history of being vulnerable, 47% are linked to bug patches and 50% were never impacted by a vulnerability or a bug.

Thus to reproduce these proportions, we randomly select for every vulnerable 31 files that were never declared as vulnerable. These 31 files are mined from the project according to the time that vulnerability was declared, *i.e.,* we ignore the changes that made after this point in time. To be as close to the reality as possible, among these 31 files we randomly select 15 of them from a pool of files that have a history of being linked to a bug patch and the remaining 16 from a pool of files that were never implicated in a patch. Thus, we constructed a set of approximately 52,500 files that reflect the actual ratios of vulnerable, buggy and non-vulnerable files in Linux, i.e., 3% vulnerable, 47% of being linked to bug patches and 50% clear files.

7.4 Studied Methods

Replicating a vulnerability prediction study is not always straightforward since several points have to be considered and some decisions taken by the authors of the initial study might be unknown. To avoid bias we use the same machine learning techniques and the same filters as those applied in the original study. Still, filters used are not always clearly indicated in all the studies.

While Section 7.3 presented, the choice made for the second (Data collection) and third step (Dataset building) of VPM evaluation process presented in Figure 1.3, this section describes how we proceed with the replication of the selected approaches regarding step 4 (feature extraction) and 5 (Model building).

7.4.1 Include and Function Calls

This approach is based on a simple intuition, which is that vulnerable files share similar sets of imports and function calls. Thus, Neuhaus *et al.,* [133] built two models using the profile of either imports or function calls to discriminate between vulnerable files and non-vulnerable ones.

Features Extraction To build such models, we first extract for each file under analysis its includes and its function calls. To do so, we use a simple regular expression for gathering the includes and use the AST of the files to retrieve the function calls. Similarly to Chapter 5, we rely on Joern to retrieve the AST.

Machine Learning Technique Once all the features extracted, we first proceed with a filtering phase. Indeed, the number of different Includes and Function Calls over the whole dataset might be too high, and causes dimensionality issue. Neuhaus *et al.,* suggest using minimum support, *i.e.,* a minimum amount of time a potential feature should appear in the training set to be kept as a feature, and set it to 5% for Function calls and 3% for Includes. Regarding the learning technique, Neuhaus *et al.,* [133] decided to use a SVM algorithm with a linear kernel for this task. We used the Weka[23] core library and its libSVM module for the replication.

7.4.2 Software Metrics

Shin *et al.,* conducted many studies using software metrics (see Section 2.1.2). We replicate their most comprehensive one based on complexity, code churn and developer activity metrics. These metrics were used as an indicator of software vulnerabilities [167], mainly due to their success in defect prediction.

Features Extraction In their work, Shin *et al.,* created 4 models: one for each kind of metrics, and one combining them. We report results for the combined one since this gave the best results. To build such models, we first have to make the metric measurements for the studied files.

Complexity. In the original study, 14 complexity metrics were suggested. These were broken down into three categories,*i.e.,* intra-file complexity, coupling and comments density. These metrics were computed using the Understand tool.

Code Churn. The original work was suggesting the use of 3 metrics, number of changes, number of changed lines and number of added lines. The way to retrieve these metrics is not described in the original work but is easy to assume.

Developer Activity Metrics. The initial study used a tool developed by its authors which was not available. Thus, we had to reimplement it. However, in their study Shin *et al.,* discovered that the only interesting metric of this category for vulnerability prediction was the "number of developers who have worked on every component". To minimize the risk of bugs in the implementation as well as the computation cost, we only focused on this one as it can be easily retrieved from the git history. Indeed, the cost of computing a whole developer network for the remaining metrics is important given the size of the Linux kernel.

To retrieve all the aforementioned metrics, we use the tools introduced in Section 5.6.

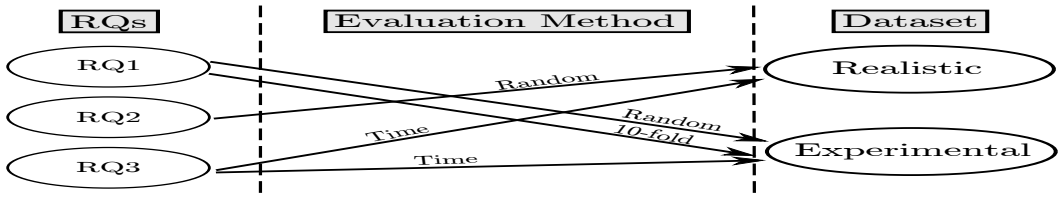


Figure 7.1: Evaluation methods and datasets that were used to answer our RQs.

Machine Learning Technique Shin *et al.,* hypothesized that a subset of either complexity, code churn and developer activity metrics or a combination of them could predict vulnerable files. To select the right subset for each case, we, as suggested, only keep the three best metrics to build the model using Information Gain as ranking. Regarding machine learning, we used as in the original study LR with Weka.

7.4.3 Text Mining

Text mining was suggested for vulnerability prediction by Scandariato *et al.,* [161]. The underlying idea of their study was to suggest a method capable of choosing features without any human intuition. This is in contrast to the other two approaches we replicated. As we stated, this approach creates a bag of words from the source code of the training files under and builds a model based on them.

Features Extraction The feature extraction of this approach is quite straight forward. The file's source code is split into tokens which are imported to a vector of unigrams. Then, the frequency of each unigram in the file is computed. The delimiters for the tokens are based on the language punctuation characters and the frequency is not normalized. In this replication, we reimplement the proposed tokenizer by adapting it for the C language.

Machine Learning Technique Once all the features have been extracted, we proceed in a similar manner as the include and function calls by creating a list of all unigrams that are present in all the files of the training set. Then, we built the feature matrix based on it.

Yet, instead of using minimum support, Scandariato *et al.,* suggested discretizing the count of each word and making it binary using the method of Kononenko [105]. The discretization is first computed on the training set and then apply on the testing set. The features rendered useless by the discretization (all binary values are the same in the training set) are then removed.This part required search to retrieve the right filters with the right options in Weka, similarly to what was used by the authors.

In the end, we figure out that the first filter to use is "Discretize" with the options "Kononeko", makes binary and use bin number activated and the second one was "RemoveUseless". The need for these two filters is to reduce the number of features which are exploding (up to 2 million in our dataset). Thus, we found out that these filters were indeed able to divide the number of features by 10. Finally, we use RF with 100 trees for the machine learning part as the authors found that this algorithm was performing better than the other algorithms.

7.5 Methodology

In this section we detail the evaluation methodology used to answers the different research questions, this corresponds to the third step of the vulnerability evaluation process presented in Figure 1.3.

For RQ1, we use the experimental dataset and two evaluation methods; stratified 10-fold-cross-validation and random splitting. We use cross validation since this is the main evaluation method used by the previous studies *i.e.,* [184, 133, 167, 161]. However, the use of two independent datasets, one for training and one for evaluation is important to get reliable results. Therefore, we also use random split to split the dataset into two sets of the same size. To check the generalization, we repeat the process 50 times to create 50 distinct experiment settings.

To answer RQ2, we use the realistic dataset and random splitting repeated 50 times. Regarding RQ3, we split our datasets according to 20 reference points, *i.e.,* time split. Each reference point corresponds to a release date of the Linux kernel, from the 2.6.28 (released 25 December 2008) to the 3.7 (released 10 December 2012). To evaluate the effectiveness of the studied classifiers, we rely on precision and recall and MCC metrics presented in Chapter 1.3.3.

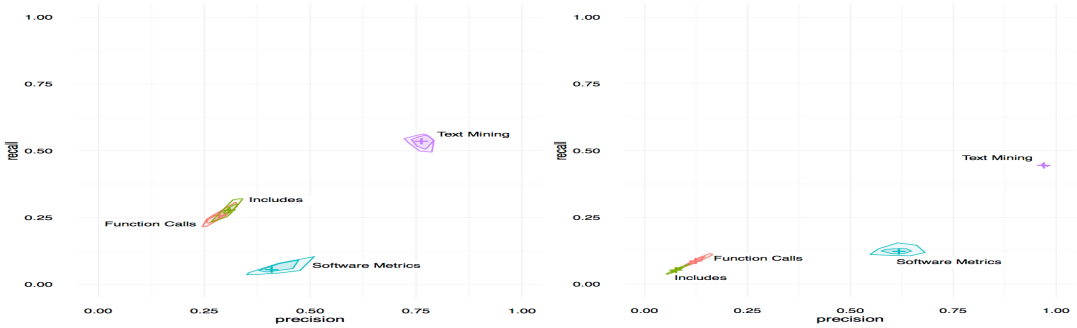
The evaluation methods and datasets are summarized in Figure 7.1.

7.6 Results

7.6.1 RQ1: Experimental Dataset

Table 7.1 records the average precision and recall values, when running 10 times a stratified 10 fold cross validation on the experimental dataset. It should be noted that median values are very close to the average ones. From these data we can observe that three methods are performing reasonably well with an MCC close or equal to 0.6: includes, function calls and text mining. These approaches have precision above 70% which is often mentioned as a "practical threshold". Recall values for these two methods are approximately 60%. The model of software metrics performed worse, with relatively low recall and MCC.

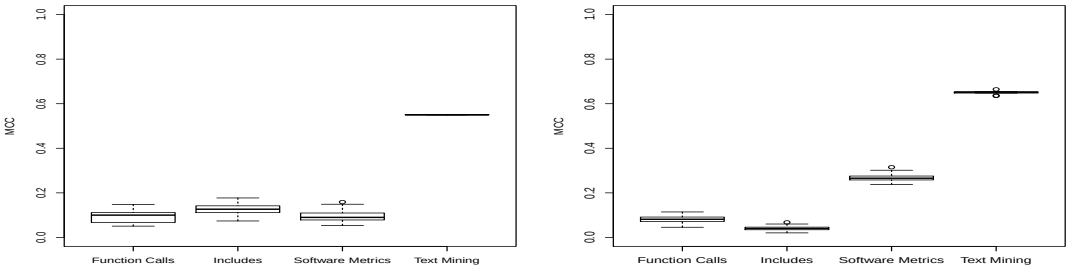
Figure 7.2a shows the precision and recall values of all the studied approaches. We present them under the form of a bag plot to visualize the variation between the performed repetitions of the random splits and ease the comparison. Evidently, the text mining approach is performing best in terms of precision and recall. It even manages to be above the "practical threshold" with respect to precision; with an average value of 76.3%, while its average recall is 53%. This impression is further confirmed by the result of the MCC presented in Figure 7.3b where the Text Mining approach outperforms by far the other approaches. Includes, Function Calls and Software Metrics while achieving similar MCC score around 0.15, performs differently in terms of precision and recall. Includes and Function Calls achieved in average 30% of both precision and recall, while software metrics approach provides an interesting precision ranging from 35% to 55% but at the price of a low recall, which is less than 10%.



(a) Experimental (RQ1) (b) Realistic (RQ2)

Figure 7.2: Bagplot of precision over recall

When using random split for the experimental and realistic datasets.



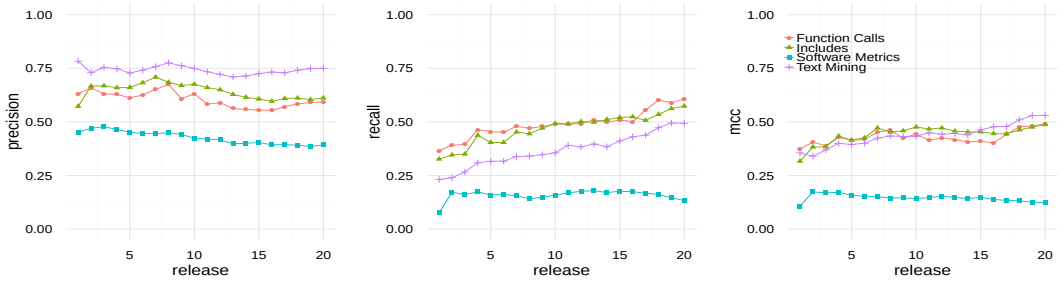
(a) Experimental (RQ1) (b) Realistic (RQ2)

Figure 7.3: MCC box plots

When using random split for the experimental and realistic datasets.

Table 7.1: Precision, recall and MCC for cross validation *experimental dataset RQ1.*

	Software Metrics	**Includes**	**Function Calls**	**Text Mining**
Precision	65%	70%	67%	76.5%
Recall	22%	63%	64%	58%
MCC	0.28	0.59	0.58	0.60



(a) Precision (b) Recall (c) MCC

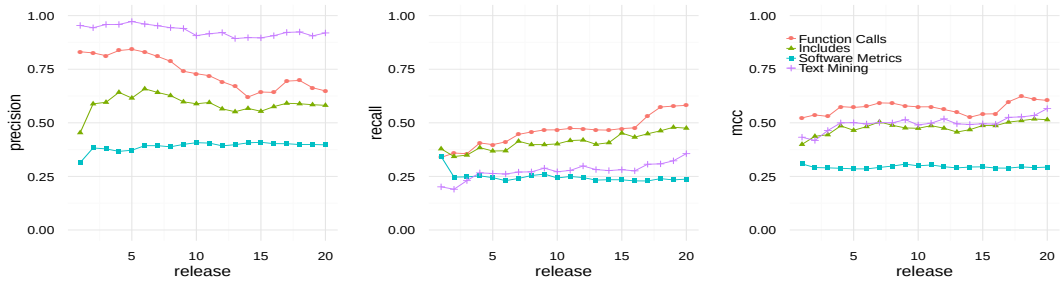
Figure 7.4: Time split for the experimental dataset (RQ3)

7.6.2 RQ2: Realistic Dataset

RQ2 regards the random split evaluation on the realistic dataset. Figure 7.3b displays the MCC results. Text mining clearly outperforms the other approach with an average MCC of 0.65. While observing the result in terms of precision and recall (Figure 7.2b), we can see that this is mostly driven by a great precision close to 100%. Interestingly, the software metrics model is the second best with an average MCC of 0.27 and an average precision of 60%. Include and Function calls are both performing poorly with MCC close to 0.1, meaning that these kinds of approaches in this context perform barely better than a random classification.

7.6.3 RQ3: Evaluation with respect to Time

RQ3 regards the evaluation of the studied methods when using past data to predict future vulnerabilities. Figure 7.4 depicts the results while applying on the experimental dataset. From this data we can observe that the most precise approach remains the Text Mining one. Its best precision, of approximately 78%, is achieved in the first studied release. Includes and Function Calls are following closely with a precision ranging from 60% on the last release to a maximum of 71%, on the 8th studied release. Precision of the Software Metrics is slowly decreasing from 48% to 40%. All the recall values have an opposing trend, *i.e.,* they improve over time, which is logical as the number of vulnerable case to train on, increases. Surprisingly, the best recall values are obtained by the Includes and Function Calls methods, with results ranging from 32% to 62%. Text mining performs slightly worse, with results ranging from 22% to 50%. Regarding MCC, we observe that Include, & Function calls and Text Mining are performing similarly.



(a) Precision (b) Recall (c) MCC

Figure 7.5: Time split for the realistic dataset (RQ3)

Figure 7.5 presents the results with respect to the realistic dataset. Interestingly, these are not so different from those we got for the experimental dataset. The Text Mining is still the most precise. Regarding the recall, Function calls is performing better managing to get more than half of the vulnerabilities for the latest releases. This can be observed on the MCC graph where Function calls is the best approach for all releases, while Includes and Text mining perform similarly.

7.6.4 RQ4: Memory and Time Cost

In the previous RQs, we focused on measuring the effectiveness of the models with respect to different settings. In these regards, the text mining approach seems to be the best in most of the cases. Yet, this comes at a price; time and in memory consumption.

Perhaps the biggest issue is the cost in memory. Indeed to run the experiments on the realistic dataset, 200 GB of memory was required in average by the Text Mining method, while half of it were required for Includes and Function Call with respectively 80 GB and 120 GB. In this regard and despite its poor performance, the Software Metrics is quite thrift in memory with only 4 GB required. Arguably, these numbers are partly due to the Weka library, which might not be the best choice for large datasets. Yet, the impact of dimensionality on memory, *i.e.,* the number of features used by the model is quite visible and model with an unfixed set of features are doomed to consume a lot of memory.

Regarding the time required to run the different parts of our study, we observed the following: Text mining is extremely fast for the extraction of features but requires some time to both train and test (up to 3h in the case of the realistic dataset). This is due to the explosion of features caused by the use of bag of words and due to the use of several filters in order to reduce them. The Software Metrics, due to its small number of features, is really fast with respect to training and testing (less than 10s). However, the feature extraction phase is time consuming as it requires for each file to build the AST and browsing its complete history, which is especially impacting in the case of the Linux kernel. The Includes and Function Calls model seems to offer a nice trade-off with a fast time to extract the needed features and a shorter time than text mining to build, learn and evaluate. It is noted that Function Calls usually have a higher number of features than Includes and thus require more preprocessing.

Table 7.2: Defect Statistics

defects not declared as vulnerabilities

Systems	No Defects	No Fixed Defects	Unique Defective Files
Linux Kernel	3,160	5,193	2,428
Wireshark	3,871	8,019	1,907
OpenSSL	2,442	7,741	1,733
SystemD	1,868	3,538	925
Total	**11,341**	**24,491**	**6,993**

7.7 Additional Tooling

In this section, a description of the BugCollector extension of the Data7 framework is presented. The Bug Collector extension supports the collection of software defects (other than vulnerabilities). The purpose of this extension is to collect data related to potential defects and their patches. Indeed, as explained in Chapter 4, to retrieve the highest possible number of vulnerabilities in a project, Data7 links the references of bug ids (mentioned in vulnerability reports) with commit messages. Using the same process, Data7 can also collect defects mentioning (in the commit messages) bug ids and bug fixes.

Bug Collector stores all the related information in a separate dataset. In particular, the Bug Collector includes a list of commits that fixed a bug which contains their hash, timestamp, message and patches (files in their states before and after fix). Table 7.2 records descriptive statistics about the collected bugs as of June 2018.

7.7.1 Collection Process

For a given dataset DV generated by the Data7 framework for a project P, two cases arise whether the project declares the bug id in its commit message or just mentions a bug correction. In the first case, the defect dataset DB can be generated from DV (see Listing 6) , in the other case a pass through all the commits is required (see Listing 7).

7.7.2 Using the tool

To create and update a bug dataset, only a configured installation of the data7 tool is required. Listing 8 present how to use the tool. More information on the tool is available on its Github page `https://github.com/electricalwind/Data7-BugCollector`.

Listing 6 Bug Tracker ID mentioned in commit message

```
for each (bugID found in DV){
        if (bugID not link to a vulnerability){
        for each (commit attached to bugID){
                add commit information to DB}}
        else {
                if(bugID exists in DB){
                        remove all commits link to BugID from
                           DB}}
}
```

Listing 7 Mention to a bug fix

```
for each (commit in P history)
        if (not yet processed)
                search mention to a fix in the commit message
                if (found)
                        add commit information to DB
for each (commit in DB)
        if (commit used in DV)
                remove commit from DB
```

Listing 8 Generating a Bug dataset

```
ResourcesPathExtended path = new ResourcesPathExtended("Path
   To Save Tour Data into");
Importer importer = new Importer(path);
BugCollector bugCollector = new BugCollector(path);
BugDataset bd =
   bugCollector.updateOrCreateBugDataset(projectName);
//simply loading a dataset
BugDataset bd = new
   ExporterExtended(resourcesPathExtended).loadBugDataset(projectName);
```

Table 7.3: Comparison with related work

Time-based results recorded as (Y / Z) refer to (Experimental dataset / Realistic Dataset). The mark 'X' denotes the absence of reported results by the previous study.

		Includes and Function calls		Software Metrics				Text Mining		
		Neuhaus *et al.*, [133]	This paper	Shin *et al.*, [167]	Shin *et al.*, [171]	Walden *et al.*, [184]	This paper	Scandariato *et al.*, [161]	Walden *et al.*, [184]	This paper
Cross validation	Precision	70	70	3 - 5	9	2-52	65	90*	2-57	76
	Recall	45	64	87 - 90	91	66-79	22	77*	74-81	58
Time	Precision	X	64 / 73	3	X	X	42 / 39	86*	X	74 / 93
	Recall	X	48 / 46	79 - 85	X	X	16 / 24	77*	X	37 / 27

*Estimated from the graphs and reported data of the paper.

7.8 Discussion

7.8.1 Implications

The study concerns two scenarios: one that corresponds to the random split of the datasets and one corresponding to the time split.

A direct implication of the results is that in this context, historical data are good in supporting relatively accurate predictions on future vulnerabilities. This is quite encouraging since it suggests that VPMs can be useful and practical. As shown in Figures 7.4 and 7.5 the top-performing prediction models achieve precision values of approximately 75% with recall of approximately 50%, which are judged by the studies of Morrison *et al.,* [126] and Shin *et al.,* [171] as satisfactory. Interestingly we found a small influence of the imbalanced data (as shown by the differences between Figures 7.4 and 7.5) on our results. This is probably due to the difference in the dataset, still as it is contrary to the results reported by Morrison *et al.,* [126], there is a need for further research on the impact of data imbalance on the prediction models.

Another important findings highlighted by the results regard the MCC values found. The MCC coefficient quantifies the quality of the predictions when compared to a random one. Thus, an MCC value equal to 1 represents a perfect prediction, while 0 a random prediction one. Therefore, all of our predictions (under all studied settings) are far better than the random ones (since we get MCC values in the range of 0.25 - 0.6) which indicates that the built models do manage to learn relatively well.

Interestingly, the results of the practical case (time split) are closer to those of cross validation (Table 7.1). This can be explained by the fact that Linux kernel vulnerabilities can be well predicted by the historical data (as discussed in the beginning of this section). Thus, most of these data are probably selected by the 9 training folds that are used in every of the 10-fold cross validation iterations. Therefore, 10-fold cross validation gives similar results with the random split. Of course if historical data were not enough, cross validation would probably provide an overestimation of the models' performance.

Overall, the data suggest that in the practical case (time split) the most effective approaches are the one based on the Includes and the Function Calls. This is somehow surprising since these methods were the first to be suggested. In contrast, the Text Mining technique is by far the best one when considering the general scenario or favouring precision over recall. This could be seen as an ability of Text Mining to easily learn what the training data have to offer, whereas Include and Function Calls require a more representative training dataset.

7.8.2 Differences With Previous Studies

In this study, we replicate three of the main VPM methods. A natural question to ask is how the presented results compare with ones reported in previous studies. Table 7.3 summaries this data. Interestingly, there are some differences especially regarding the recall values. Regarding the Includes and Function Calls methods [133], our data are in line with those reported in literature (case of cross validation), *i.e.,* same precision values with slightly better recall ones.

With respect to the software metrics method, we found completely different results from previous studies [167, 171, 184], *i.e.,* better precision and significantly lower recall. This difference could be explained by the way we construct our datasets and/or by the fact that undersampling is used in these studies to balance the datasets which may have impacted the drawn result.

Finally, with respect to the text mining approach we observed comparable result in terms of precision with the study of Scandariato *et al.,* [161]. The recall found was, however, lower than the one claimed, up to 50% in the case of time splits. In the study of Walden et al. [184] there was a large variation on the reported results, *i.e.,* 2% of precision in a case while, 57% on another one. This makes a comparison with our study difficult and probably irrelevant. Nevertheless, the same study only used cross validation where we found a better precision and worse recall than them.

7.9 Threats to Validity

Regarding construct validity, we rely on the Data7 framework to collect bugs and vulnerability fixes. This framework ensures the retrieval of known and fixed vulnerabilities but, undiscovered or non-fixed vulnerabilities are ignored. This might result in false negatives with a potential impact on our measurements. However, given the size of the Linux kernel and the long history of vulnerability reports, we believe that it is unlikely to have many such cases.

Regarding internal validity, this work only considers source code files written in C, but these are not the only files that can be linked to vulnerabilities. For instance, there are parts of the Linux kernel which are written using assembly code. However, since the great majority of the Linux kernel is written in C, it limits the impact of this threat. Additionally, potential bugs in the implementation may also influence our results. Also we might unintentionally not re-implement exactly the original approaches. To reduce these threats we carefully inspected all of our code, parameters and experiment decisions with respect to the exact replication of the previous approaches. We also manually tested and verified the implementation. Since the results are in line with previously published ones, these threats are not of particular importance. Furthermore, following the suggestion of Shin *et al.,* [167], we used the three best metrics according to Information Gain to build Software Metrics model. Still there is a possibility that additional metrics could provide different results.

Regarding external validity, the study is limited to the Linux Kernel and thus, our results might not be generalizable to other projects or contexts. However, we studied a real, large and widely used project; the Linux kernel. Also, we studied a large number, much larger than any previous study, of real vulnerabilities (actually all reported vulnerabilities in NVD). On the other hand, studies presented in the remaining part of this thesis will use more projects.

7.10 Conclusions

In this chapter, we performed an exact independent replication of three of the main VPM approaches, Software Metrics, Includes and Function Calls and Text Mining in the context of the Linux Kernel. We showed that several parameters like the evaluation method, application scenario and the composition of the dataset, often ignored in literature, can have a major impact on the reported results. We also demonstrated that when aiming at predicting future vulnerabilities, the Includes and Function Calls methods offers an interesting trade off, while when aiming at random instances of vulnerabilities, Text Mining clearly outperforms the other methods.

While this study provides us with insight on VPM, some issues remain to be addressed. First of all, are the settings suggested by the authors of the initial studies really the best? Second, what is the real impact of unbalance datasets on the performance of VPMs? Finally, do the results generalize to other projects? These issues are addressed in the next chapters, starting by the presentation of a framework to evaluate VPM.

8

FrameVPM: a Framework to Build and Evaluate VPMs

Replication and comparison of approaches are important and necessary process. Yet, aside from the study from Walden et al., [184] and the one presented in the previous chapter, there is no study comparing different VPM approaches. To facilitate, the development of new approaches and ease the comparison with previously proposed ones, hence tackling our second challenge, we introduce a framework named FrameVPM (standing for Framework for Vulnerability Prediction Modelling) allowing the evaluation and comparison of VPMs. This framework is an extension of the Data7 framework presented in chapter 4 and is composed of three different parts. The first one "organize" transforms the dataset gathered by Data7 into one that is suitable for evaluating VPM approaches. The second one "analyze" deals with the feature extraction of the different approaches on this dataset, while the last one "learning" handles the ML aspects and the evaluation of the approaches.
This chapter is based on a project available on Github : `https://github.com/electricalwind/framevpm`.

Contents

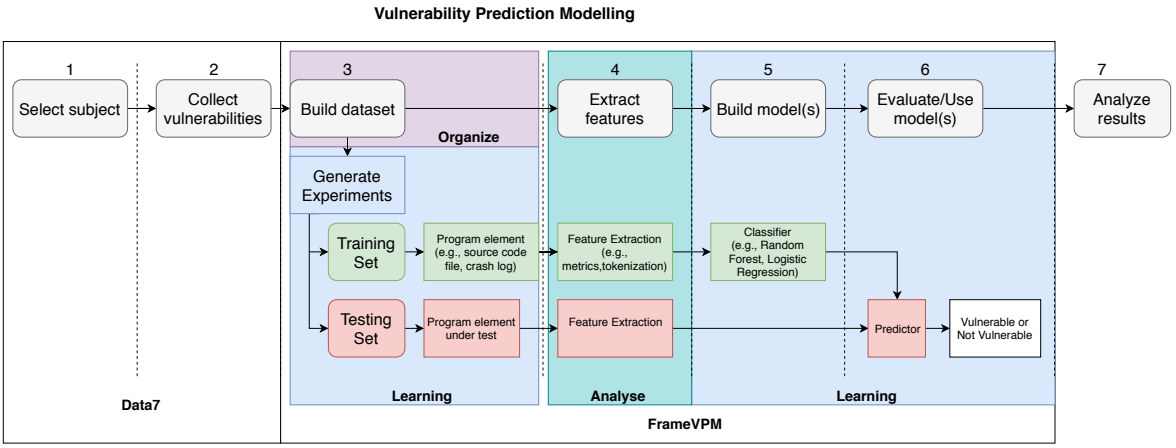


Figure 8.1: FrameVPM parts in VPM evaluation

8.1 Introduction

In this chapter, we introduce the FrameVPM framework. FrameVPM extends the Data7 in order to enable the comparison between VPMs. FrameVPM is composed of three different parts, the first one, called "organize", transforms the data gathered by Data7 into one that is suitable for evaluating VPMs. The second one, called "analyze", deals with the feature extraction of the different approaches on the dataset, while the third one, called "learning", performs the ML and the evaluation of the methods. Figure 8.1 how the different part of the FrameVPM fits into the context of VPM.

The collection of vulnerabilities is handled by the Data7 framework presented in Chapter 4. However, collected data is not suitable to perform vulnerability prediction as it contains only vulnerable examples. Thus a dataset transformation is required (step 3). Previously, we perform this through the augmentation of this set with buggy files ("experimental" dataset) and with clean files ("realistic" dataset). This augmentation process is handled by the "organize" component of the FrameVPM framework.

Once the dataset ready, the next step is features extraction (4). In the case of Text Mining techniques, this means tokenizing the files and extracting bags of words, while for Code Metrics, this implies computing the metrics for each file. This part of the process is handled by the "analyze" component of the FrameVPM framework.

The last step is the actual evaluation, which is handled by the "learning" component of the framework. The evaluation can be split in three parts, the first one regards the evaluation methodology (3). Typically, in the previous chapter, three possibilities were explored, stratified 10-fold cross validation, random splitting and time-based splitting. The second part concerns the choice of the ML settings (5), *i.e.,* classifier, filters, while the third one is the execution of approach(es) (6).

The framework currently supports all projects handled by the Data7 framework and integrates the 3 approaches presented in the previous chapter along with a new one that uses the naturalness of Software as a feature. We introduce the new approach in the Chapter 12. The following section provides additional information on the different components as well as examples on how to use the framework.

8.2 Details on the Framework

8.2.1 Organize component

The "organize" component gathers (using the Data7 framework) and transforms the data, suitable for the training and evaluation of VPM. Depending on the purpose of the experiment, several strategies are possible. The first one, consists of gathering non-vulnerable data with instances from another category (similar to what we did in the previous chapter). This strategy is useful to evaluate the distinction capabilities of the models. However, it is not helpful when evaluating the real usage of a VPM, *i.e.,* as it would be used by developers. A second strategy, which is the one implemented by the "organize" component, consists in mapping the vulnerabilities to the software releases they impacted to create a release-based dataset. Most of previous studies on VPMs are in fact relying on release-based datasets. Those datasets can then be used for evaluating one release at a time with strategies such as cross validation or random split, or using a set of releases for training and another for testing.

Mapping vulnerabilities to the impacted releases can be done using the information provided by the NVD (NVD provides a list of impacted versions per recorded vulnerability). Though, it is impractical to study every single release of each project. Thus, a subset of release has to be selected. In the case of the projects from the Data7 framework, we select 64 versions of Linux from 2.6.12 to 4.15, 22 versions of Wireshark (all major, i.e., ending with 0) and 10 versions of OpenSSL (all major). In our study, the collected vulnerabilities are mapped to the selected releases. In case a vulnerability has no match to any of the selected releases, we matched it to the closest one (previous release) according to the commit timestamp. Arguably this can introduce some errors in our data. This is not considered as important given the large number of releases that we considered and the fact that the vulnerability introduction is quite likely to be much earlier than the vulnerability discovery time.

Once we make the mapping, the vulnerable files should be extracted from the vulnerability fixes in order to declare files in the releases as vulnerable or not. This is straightforward as a simple look at the file modified by the commits fixing the vulnerability is sufficient. Still, as explained in Chapter 4, data generated by the Data7 framework is raw and contains a lot of noise when applied to VPM. These cases are the following: First, some commits found by Data7 are not fixing commits but ones that add regression test to prevent the vulnerability to return. To prevent this, we don't consider file present in testing directory as vulnerable. Second, duplicate commits exist in the dataset. Indeed, when fixing different branches several commits with the same content can be made. Similarly, one commit can fix several vulnerabilities and thus be declared more than once. These two cases are not especially problematic when simply stating if a given file in a release is vulnerable, but reveal troublesome as we keep for each vulnerable file in a release the information on its related vulnerability and fix to allow further evaluation. To handle this, we maintain a list of all seen commits along with a hash of their commit message. Thus, if several commits have the same content, we only keep the first one, while if the same commit appears in more than one vulnerability, we add the information of this vulnerability to the concerned files.

Finally, it is possible that a vulnerability is fixed in more than one commits, as seen in Chapter 3, thus if the same file is modified more than once to fix the vulnerability it is not wise to keep several "fixed" versions of the files. Thus, we only keep the file in its original and final version as additional information on the vulnerability.

In addition, the framework offers the ability to integrate bug fixes to this mapping using the output of the Bug Collector package. In case such a choice is made, an additional filtering is performed to remove commits that have the same commit message as vulnerabilities.

8.2.2 Analyze Component

The "analyze" component is in charge of computing all the necessary metrics, *i.e.,* features of the different approaches, for all the files present in the dataset. This component uses the output of the organize component as input and will download automatically all the releases which possess vulnerabilities from the internet to obtain all the files from the releases to compute the metrics on.

Metrics can be computed separately on a per approach basis or in one run. Once computed, the metrics are stored and will not need to be recomputed again. So far, 8 sets of metrics are implemented, but it is possible to add metrics for new approaches by simply extending the Analyze class and implementing the processFeatures method.

Here are the already implemented set of metrics: *Simple Metrics*, *Complexity Metrics*, *Code Churn and Developer Metrics*, *Include*, *Function Calls*, *Bag of Words*, *Previous Release Naturalness*, *Other File Naturalness*.

The first 6 metrics correspond to the one used for the approaches previously investigated and can be measured at one file at a time, while the last two are from a novel approach that will be introduced in Part IV, which required additional knowledge. It should be noted that the computation of the first five set of metrics is conditioned to the analysis of C project which is not a problem in the current state of the Data7 framework.

8.2.3 Learning Component

The "learning" component is handling everything related to the evaluation of the models. Starting by determining training and testing set.

As mentioned above, several possibilities exist when performing evaluation on a release dataset. Currently, only two strategies are implemented, last release validation and three last release validations. While the first one trains on one release and evaluates on the next one, the second trains on the last three releases and evaluates on the next one. These strategies have been previously used by Hovsepyan *et al.,* [78], *resp.* Shin *et al.,* [167]. However the framework is not limited to those strategies and can be extended through the ExperimentSplitter class.

In addition to the determination of the training and testing set, the framework offers the possibility to choose the class to be evaluated. Indeed, depending on to the option to select in the organize component, the files of a given release will be categorized in 3 or 5 categories, *i.e.,* Vulnerable, Vulnerable History, (Buggy, Buggy History) and Clear. Thus, if the traditional way of performing the evaluation of VPM involves a binary classification with vulnerable and non-vulnerable components, it is possible in FrameVPM to completely customize the evaluation, like performing a ternary classification by integrating Bug to the class, or even to ignore given types of files in the set to perform an evaluation of bugs against vulnerabilities like in the previous Chapter. Currently, three strategies are implemented, the traditional one considering everything that is not vulnerable as non-vulnerable, a ternary classification considering files that have a history of being buggy or vulnerable as clear, and a last one only keeping likely to be buggy files and vulnerable files in the dataset.

Regarding the ML part, the framework rely on Weka [23] 3.9.2. Thus, any classifier and filters present in Weka are available.

Once everything id set, the approaches can be evaluated. The evaluation can be performed in a two-phase process. In the first step, the training and testing sets are created by retrieving the desired features for each one of the models using the results of the "analyze" component. Then, the chosen classifier is trained and the evaluation is performed on the testing set. Note that an approach in the learning component is not directly linked to an approach in the analysis component. Hence it is possible to create in the learning component new methods by combining the metrics from different techniques or use only parts of the metrics. Currently, 6 approaches are offered, *i.e.,* Code Metrics (combining all code metrics), Includes, Function Calls, Bag of Words, Naturalness (see 12) and CodeMetricsNaturalness (combines Code Metrics and Naturalness approach).

8.3 Using the Framework

The output of each component is automatically saved and can be accessed through a JAVA API. For the case of the "learning" component the framework also offers a Comma Separated Value (CSV) exporter.

Figures 8.2, 8.3 and 8.4 present the information related to the field of the objects saved by *resp.* the "organize", the "analyze" and the "learning" component under the form of an UML diagram.

As already explained, components need to be run in the following order (1) organize (2) analyze and (3) learning. The "organize" component returns a ProjectData object containing a map of ReleaseData objects. The ReleaseData object holds information related to vulnerable (and likely to be Buggy, if the option is selected) files of the release.

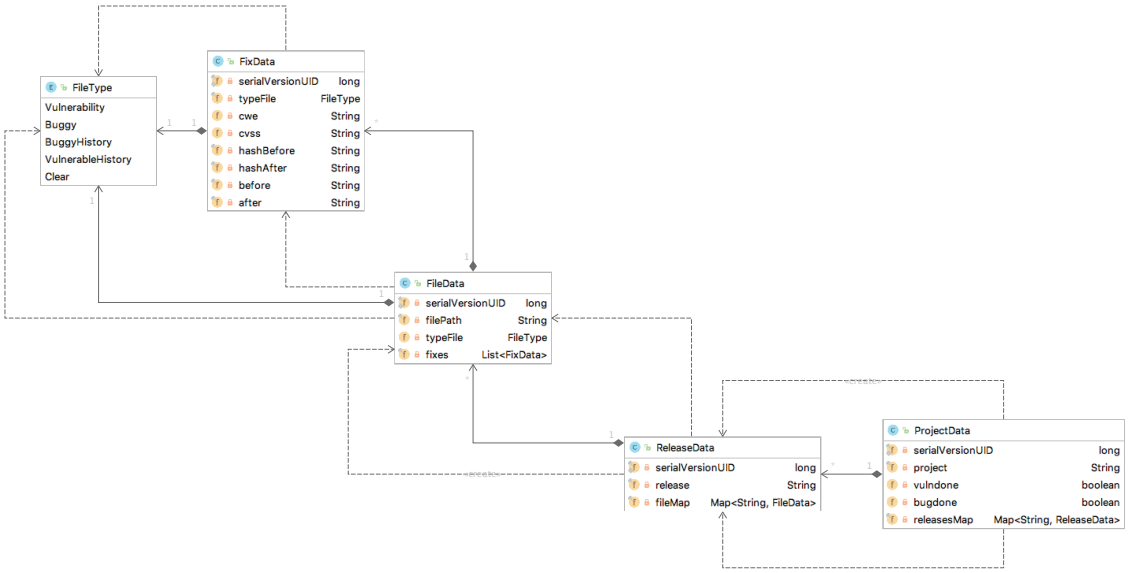


Figure 8.2: Organize Component API

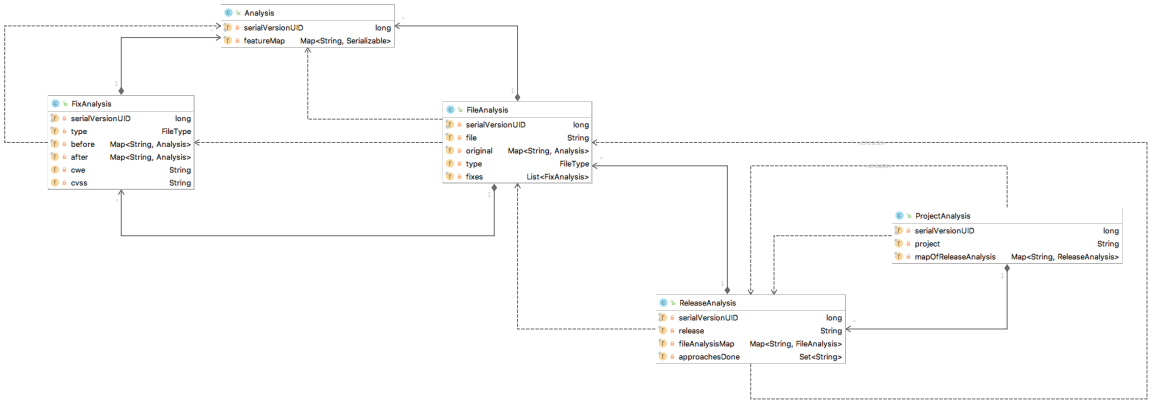


Figure 8.3: Analyze Component API

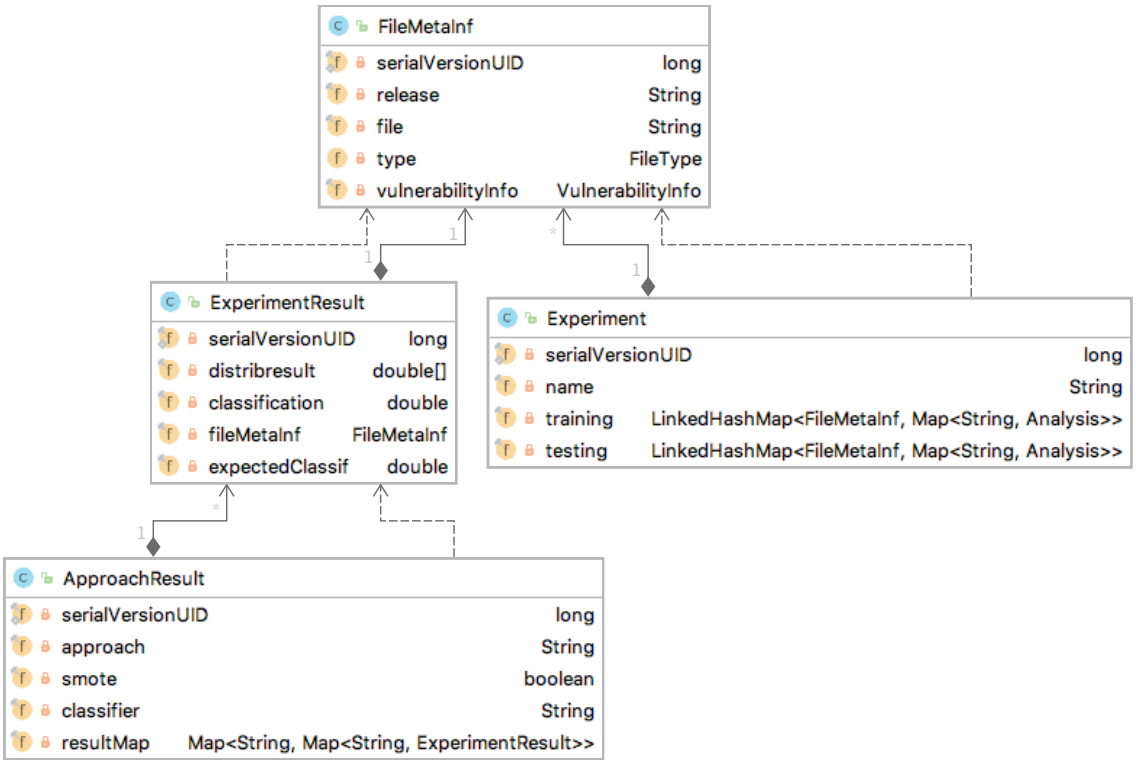


Figure 8.4: Learning Component API

The "analyze" component should be executed individually for each considered approach. The component first loads the corresponding ProjectData object and in case it has been run on a different approach previously the ProjectAnalysis Object that stores the additional results of the current analysis. At the end of the execution, the ProjectAnalysis object is returned with the included metrics of each one of the approaches.

The "learning" component has two steps. The first one is the output of the evaluation strategy, *i.e.,* training and testing sets under the form of a List of Experiment objects. Those objects stored the analysis data of the training and testing set. This process duplicates the analysis data but ensures that all approaches can be evaluated on the same sets. The second one is the final results, for a given classifier and for a given list of experiments under the form of an ApproachResult object. This object contains a map of the different experiment results. This map contains in turn the result of all samples of the testing set including the distribution probabilities, as well as information on the sample.

The framework can be downloaded from GitHub and the latest version can be found at `https://github.com/electricalwind/framevpm`. FrameVPM requires maven and Java (8+). To proceed with the installation, the user should type mvn install in a terminal at the location of the project. The project can then be used from any maven project by adding the dependency presented in Listing 9.

Succeeding to the compilation, the user needs to define a path (to the place where the binary will be saved). Then Listing 10 indicates how to run the organize and analysis components. Note that the boolean parameter in the method balance() of the Organize class indicate whether information from the bug dataset should be integrated, while the runAll() methods from the Analyse class allow to perform the analysis for all registered approaches.

Listing 11 presents an example of the use of the learning component. First a class model needs to be chosen. In the example, the traditional binary classification vulnerable, non-vulnerable component is chosen. Then, the way to split the dataset into training and testing is determined, in this case, using the last three releases as training and the next one as testing. Next, the approach to evaluate is picked, here Bag of Words. Finally, the approach can be run with a selected classifier that can be found in the Classifiers class. The smote parameter indicates whether the SMOTE algorithm [44] should be used. This algorithm is often used in presence of largely unbalanced datasets to oversample the minority class. Additionally the last two lines show how to export the result to a CSV file.

Listing 9 Dependency to add in pom.xml to use FrameVPM

```
<dependency>
    <groupId>lu.jimenez.research</groupId>
    <artifactId> framevpm </artifactId>
    <version>1.0</version>
</dependency>
```

Listing 10 Calling the Organize and Analyze Components

```
ResourcesPathExtended pathExtended = new
   ResourcesPathExtended("Path To Save Tour Data into");
Project project = CProjects.LINUX_KERNEL;
new Importer(pathExtended).updateOrCreateDatasetFor(project);
new
   BugCollector(pathExtended).updateOrCreateBugDataset(project.getName());
new Organize(pathExtended, project.getName()).balance(true);
new Analyse(pathExtended, project.getName()).runAll();
```

Listing 11 Evaluating an Approach in FrameVPM

```
ResourcesPathExtended pathExtended = new
   ResourcesPathExtended("Path To Save Tour Data into");
Project project = CProjects.LINUX_KERNEL;
//...
ClassModel classModel = new VulNotVul();
ReleaseSplitter experimentSplitter = new
   ThreeLastSplit(pathExtended, project.getName());
List<Experiment> experimentList =
   experimentSplitter.generateExperiment();
Approach[] approaches = new BagOfWordsApproach(experimentList,
   model);
ApproachResult result = approach.runWith(classifier, smote);
CSVExporter csvExporter = new CSVExporter(pathExtended);
csvExporter.exportResultToCSV(project.getName(),
   experimentSplitter.getName(), classModel, result);
```

8.4 Limitations

FrameVPM is still in its early days and is thus less mature than its Data7 counterparts. As such, the current version has some potential limitations, which are discussed here.

First of all, some functionalities are still missing especially in the learning component. Good examples are the Experiment Splitter and Class Models parts with only two, *resp.* three strategies implemented so far. Still the implementation of additional strategies can be done easily by extending the abstract classes. Regarding the evaluation of approaches, some changes to the architecture may be done in the future to ease the use of filters. Another limitation directly linked to the maturity of the project is its lack of documentation aside from the present chapter. Still as the next chapter will show, the framework is already functional and will improve over time.

8.5 Conclusion

In this chapter, we introduced the FrameVPM, a framework exploiting the data gathered by the Data7. FrameVPM purpose is the training and evaluation of prediction models in experimental and realistic settings. The framework is composed of three different components that should be executed in a given order.

Currently, the framework allows the computation of 8 sets of metrics used by 6 different state of the art VPMs, but can easily be extended with additional metrics and/or approaches. The toolset despite being in its early days, it can already be used to perform large scale comparisons between different approaches. In the context of this document, the framework is used in chapter 9 and 12.

9

An Evaluation of VPMs wrt. Severity and Mislabelling Noise

In previous studies, VPMs have always been evaluated on different datasets, settings and features. Chapter 7 presented an exact independent replication of 3 approaches. Yet, this study only address two of these concerns, i.e., different datasets and features. Indeed replicating an approach does not require the exploration of alternative settings, e.g., ML algorithms, filters. This can bias conclusion as there is no guarantee that the suggested settings are indeed the best one. Additionally, previous studies mainly focused on the "raw" performances of VPMs without investigating some of their specificity like how they handle severe vulnerabilities or the issue of "mislabeling noise". Thus, this chapter presents the largest experimental study of VPMs investigating all the aforementioned points. The evaluation is performed using FrameVPM framework, presented in chapter 8, on three projects handled by Data7 (Linux Kernel, Wireshark and OpenSSL).
This chapter is based on yet unpublished material.

Contents

9.1 Introduction

In Chapter 7 an exact independent replication of three of the main VPM approach was presented. The results although interesting can be discussed as to truly replicate an approach settings like classifiers and filters are bound to be the same as those from the original study. Yet those settings are not guaranteed to be the "best" ones. In all three original studies, results of only one classifier is presented along with a word from the authors stating that it is the best one.

Furthermore, none of the previous approaches takes into account vulnerability severity. Nevertheless the ability to predict, not only vulnerable components but also severe ones, is important for any practical deployment, because many remediation strategies naturally involve prioritization according to severity. With the existing state of knowledge, there is no scientific evidence that predictive modelling does not, for example, merely identify low severity vulnerabilities, while failing to identify any high severity vulnerabilities. Yet, results from Chapter 5 suggest that it should be possible to tune models to find severe vulnerabilities.

In order to address these issues, a comprehensive empirical study of the three main previously proposed approaches is conducted using FrameVPM. In particular the use of five widely used classifiers is investigated.

The question of vulnerability severity is addressed by investigating the ability of each technique to predict, not only vulnerabilities, but their ability to prioritize the most severe ones. An additional phase of tuning is performed by systematically attempting to improve the prioritization performance of each classifier, both by weighting according to severity, and also by enlarging the set of training data to include additional examples of defects believed not to contain vulnerabilities.

An investigation of the sensitivity of the predictive models to the presence or absence of the vulnerabilities is as well performed, by applying the models both before and after the vulnerabilities have been fixed. With this extra aspect of the experimental methodology, we seek to provide an approach to a form of "round-trip" validation; the classifier should predict that the vulnerability is present before fix, and absent after it. Although this is a very demanding criterion from the ML point of view, it provides important information to the practitioner. Should it turn out that VPM are good in general, but perform poorly in this additional more specific sensitivity evaluation, then a practitioner could use vulnerability prediction to prioritize human follow-up investigations, thereby partly mitigating the primary scaleability concern of prioritizing such follow-up according to likelihood and severity. However, the engineer could not (and *should not*) rely on specific predictions, relating to the presence or otherwise of vulnerabilities in particular pieces of code.

Finally, it is unlikely that a practitioner could ever hope to have a complete and reliable ground truth on which to train any model. New vulnerabilities can be (and are) identified in long-standing and widely used code. New techniques converting apparently non-vulnerable faults into vulnerabilities are also designed. Vulnerability discovery is an adversarial process, in which those seeking to exploit faults keep innovating.

Table 9.1: Vulnerability Dataset

Systems	No. of Vulnerabilities	Average CVSS	No. of Vulnerable Components
Linux Kernel	1,202	5.41	1,508
Wireshark	265	4.99	221
OpenSSL	126	5.34	164
Total	**1,593**	**5.38**	**1,898**

A practitioner could only hope to train a predictive model on a *partial* ground truth that would include miss-classified faults and code. Inevitably, some code would be miss-classified as non-vulnerable, due to the presence of as-yet-undiscovered faults. More importantly for vulnerability detection, some previously identified and apparently innocent faults may subsequently turn out to induce vulnerability.

To address this issue of mislabeled data (also known as "mislabelling noise" [125, 174]) we also investigate the impact of noisy historical vulnerability data on the predictive performance. That is, we construct a predictive model at time t, based on all vulnerabilities known at the time t, and use it for predictions in subsequent releases. Because we also have information on additional vulnerabilities discovered in those subsequent releases (and also present-yet-undetected at the time t), we thus have partial information about Miss classifications that can yield insights into the resilience of predictive models in the presence of realistic noise.

In summary this chapter makes the following contributions:

- It provides evidence that prediction modelling can indeed be effective (MCC values of 0.80, 0.65 and 0.40 on Linux, OpenSSL and Wireshark).
- It shows that the RF is the algorithm working best for all considered approaches.
- It shows that relatively low precision is achieved when predicting severe vulnerabilities, but overall with good top ranked suggestions (having 7, 5 and 1 severe components in the top-10 ranks).
- It demonstrates that vulnerability prediction can identify all severe vulnerable components by considering at most 60% of the systems' components.
- Finally, it shows that performance is significantly inhibited by the needle effect [167] (few vulnerable training instances compared to non-vulnerable ones) and the availability of accurate data.

9.2 Experimental Environment

9.2.1 Studied Systems

Similarly to Chapter 7, we perform the study on a large dataset of vulnerabilities gathered by the Data7 framework. The 3 projects with the largest number of vulnerabilities were chosen, *i.e.,* the Linux kernel, the OpenSSL library and the Wireshark tool. Table 9.1 reports a summary of the collected data

Yet compared to chapter 7 the decision was made to evaluate on a release basis instead of a commit basis to be closer to realistic use cases. As such we use data from previous release(s) to predict the vulnerable components of the next release. To characterize such data, we map the vulnerabilities with the releases they affected (using the NVD data). This is somehow safe, as we can approximate when each vulnerability was removed given the exact vulnerability fix time. This mapping is realized by the "organize" component of the FrameVPM framework introduced in the last chapter.

All in all, the investigation was done on 64 versions of Linux from 2.6.12 to 4.15, 22 versions of Wireshark (all major, i.e., ending with 0), 10 versions of OpenSSL (all major).

9.2.2 Selected Approaches and Classifiers

Similarly to chapter 7, we chose to investigate the three main VPM approaches, *i.e.,* Imports and Function Call, Bag of Words, Code Metrics. Yet, to avoid bias from a specific classifier we considered five popular ones, *i.e.,* AdaBoost, J48, KNN (k=5), LR and RF. These classifiers are typically used in prediction studies. To train and evaluated the models we used the learning component of FrameVPM (8).

As mentioned before in Chapters 2 and 7 the dimensionality, *i.e.,* the number of features, can be a problem. In *Code Metrics*, the feature matrix has a fixed number of columns, i.e., one per metric. This is not problematic, but *Imports*, *Function Call* and the *Bag of Words* approaches have a non-constant number of columns (these are determined by the features extracted from the training set), which is likely to introduce dimensionality issue. To handle this we used minimum support as suggested by Neuhaus *et al.,* [133] and set it to 5%.

9.3 Research Questions

In this section, the research questions we aim to answer in this empirical study are discussed, while Section 9.4 describes in greater details the experimental design and analysis we undertook to answer them.

In prediction modelling, a good practice is to analyse the characteristics of the underlying data to better understand the problem. This kind of analysis similar to the one presented in Chapter 5 can explain poor results like for example, if we have a significant prevalence of non-severe issues, it is expected that the prediction models are unable to perform well. This motivates the first question:

RQ1: *How many and how severe are the reported vulnerabilities in security intensive subjects?*

To answer RQ1 we count for each considered release of the systems, the ratio of vulnerable components and distribution of severities.

After checking data quality, we proceed to design and build prediction models. In particular, our second research question investigates how effective are prediction models at identifying vulnerable components between consecutive software releases:

RQ2: *How well the models identify vulnerable components between software releases?*

This investigation is needed to confirm the results obtained in chapter 7. To answer RQ2 we carry out a comprehensive empirical analysis of 74 releases of our three subject systems by using five classifiers, *i.e.,* AdaBoost, J48, KNN, LR and RF, with three different sets of independent variables gathered following the approaches proposed in previous work, *i.e., Imports, Function Calls, Code Metrics and Bag of Words.*

If we confirm that prediction models can identify vulnerable components, we can turn our attention to predict the severity of such vulnerabilities, thus asking:

RQ3: *Do prediction models identify severe vulnerabilities?*

To answer RQ3, we use the same techniques and independent variables as for RQ2, but restrict our attention to severe vulnerable components, *i.e.,* we predict whether a component is likely to hold a severe vulnerability or not (binary dependent variable). Following the recommendation of NIST (CVSS $v2.0$ and $v3.0$ Ratings) [22] we consider severe the vulnerabilities with a CVSS value higher than 7.

Once we assess the predictive ability of the models, we further investigate whether augmenting the training sets with more information, either human- or defect-based, could improve the performance of the above prediction models:

RQ4: *Can we improve the accuracy of severe vulnerabilities prediction by providing prediction models with either severity- or defect-based information?*

To answer RQ4, we repeat the analysis carried out for RQ3 but we use modified training sets to build the models, and compare their performance against the models built using the original training sets. In particular, we include in the training sets custom weights to the vulnerabilities based on their CVSS values and also on defect-related information. We examined three ways to assign the weights: Assigning the exact CVSS number as weight, assigning a larger weight to the severe vulnerabilities, assigning a lower weight to those components which were defective but still considering them as vulnerable.

To better understand the robustness of vulnerability prediction, we evaluate the ability of the methods at specific corner cases such as vulnerable components that were fixed at the evaluation time. In particular, we evaluate the prediction probability produced by the models on components directly before and after vulnerability fixes, which has been an overlooked aspect in literature [110]:

RQ5: *How sensitive are the prediction models at contrasting components before and after vulnerability fixes?*

Another frequent issue regards the creation of models based on incomplete or mislabelled data [174]. In the case of vulnerability prediction, this problem can occur when vulnerabilities that have not been yet reported at the training time are part of the training data as non-vulnerable components. This fact can mislead the classifiers as it mistakenly learns vulnerable components as clean. To investigate this, we train the prediction models using the information available at release time and evaluate against previous results. Thus, we ask:

RQ6: *How sensitive are the prediction models to vulnerabilities unreported at the time of training?*

9.4 Experimental Design and Analysis

9.4.1 Methodology

To evaluate VPMs and answer our RQs we performed the following analysis using the FrameVPM framework.

For every considered release, we iteratively train on the previous release(s) and evaluate on the current one. We consider two typical cases, training on the last release, similar to Hovsepyan *et al.,* [78] and Harman *et al.,* [73], and training on the last three releases, similar to Shin *et al.,* [167, 171]. We start the evaluation from the fourth release and on (since we need at least three releases to train on). We consider releases with at least 10 vulnerable components. This restriction is mandatory in order to have sufficient data for analysis.

This means that we ended up evaluating on 61, 6 and 7 releases of Linux, Openssl and Wireshark.

A usual problem in prediction modelling and especially in VPM is the class imbalance one [44]. This is because we have few vulnerable components (less than 10% per release) compared to the non-vulnerable ones, which makes it hard for the classifiers to identify the properties of the minority class, which is the one of interest. To deal with this issue we investigate the use of the SMOTE Algorithm [44]. In particular we repeated our experiments with and without it. All previous studies just let their data as such or chose to apply an over or under sampling algorithms such as SMOTE but never investigated the impact of using one.

Another usual issue, regards the use of classification algorithm with the different approaches. As it is not certain which classifier fits best with a given approach for the problem at hand, we examined their combined use on the data.

9.4.2 Pre-Analysis

The experiment involves a large number of settings that require the repetition of the analysis 35,520 times, as we need to investigate 4 approaches with 5 classifiers, 2 validation ways (using the last three or one releases), 2 ways of handling imbalance (using/not using SMOTE) for the 3 systems of interests (74 releases studied in total) and for 6 different independent variables (answering RQs 2, 4, 6). To deal with this issue, we first determine the best combination of the pair (classifier, prediction method) for all the releases on our corpus and then we applied the best-performing ones to answer the RQs. In particular, we gathered the results of RQ2 for all combination and ranked the performance of the classifiers for a given approach. Then, we computed the average rank of all pairs and normalized them to obtain a value between 0 and 1. This score indicates the best classifier for each approach. The results are presented in Table 9.2 and show that the best general ranking is RF (it achieves the best results in all columns). Therefore, we used this pair in our analysis. The best general ranking is RF (it achieves the best results in all cases). Therefore, we used this pair in the rest of the analysis.

Table 9.2: Relative ranking of classifier-method pairs

The values indicate the ratio of best performing method among the results of the method (best combination per column). It is noted that these are ranking-based results (normalized according to each method-column) and are not comparable between methods (rows are not comparable).

Classifier	Bag Of Words	Code Metrics	Function Calls	Imports
AdaBoost	0.41	0.22	0.39	0.30
J48	0.75	0.64	0.67	0.69
KNear	0.52	0.64	0.41	0.56
LR	0.56	0.62	0.66	0.55
RF	0.75	0.87	0.86	0.90

We then evaluate whether we should use SMOTE or not and whether we should train on the last three releases or just on the last one. We thus, computed all the MCC values and compared them using the Wilcoxon signed rank test. We found that SMOTE can sometimes have a positive effect, with statistically better performance metrics for all the approaches and classifier combination we examined. We also found no statistically significant differences between the last releases and the last three releases. Though, we decided to perform our analysis using the last three releases as it provided slightly more stable results than when using just the last one.

In conclusion, in the rest of the analysis we use the following settings: training using SMOTE on the last three releases with the RF classifier.

9.4.3 Performance Measurements

To evaluate the effectiveness of the studied classifiers, we rely on precision and recall and MCC metrics presented in Chapter 1.3.3.

Yet, classifiers are in fact not directly returning a class in which an element is supposedly belonging, but are instead returning a probability of the said element to belong to a class. This probability can then be used to classify the element into one of the classes, depending on a threshold. This probability of a component to be vulnerable according to a model that we call *Prediction probability* $PredP$, can be used as a ranking basis where component would be ranked in ascending order. The position of a specific component in this ranking gives us its relevant ranking, which we denote as RR.

Since we are interested in a subset (severe vulnerabilities) of our classification results, we also need a metric related to the effort put in by the users. We thus, consider the number of components that need inspection in order for someone to inspect all vulnerable components. We define this metric as:

$$Score = \frac{\#components\ need\ inspection}{\#components}$$

The *Score* metric represents the effort needed to inspect all vulnerable components. As users may focus on the most likely cases, we also need a metric focussing on the top ranked components. We define the top-n metric. This is defined as the number of vulnerable components in the top n places of the RR. We use n value of 10. When there are multiple vulnerabilities in one component, we consider the component as vulnerable and retain the most severe vulnerability score.

9.4.4 Analysis

To answer RQ1 we record the number and distribution of severity values of the systems components we analysed. We report the results as boxplots per release and analysed system.

To answer RQ2 we evaluate the prediction ability of the studied methods by using the performance metrics (recall, precision and MCC).

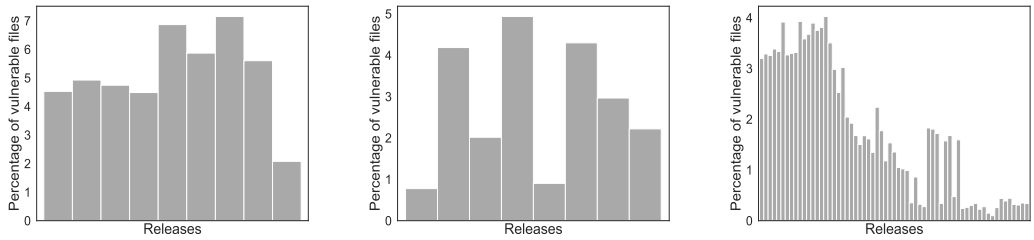
To answer RQ3 we evaluate the prediction ability of the studied methods to pinpoint severe vulnerabilities (vulnerabilities with CVSS above 7) using the performance metrics (recall, precision and MCC). We also evaluate their RR according to the *Score* and top-10 metrics that we defined in Section 9.4.3. This is done by considering vulnerabilities with CVSS above 7 as vulnerable and the others as non-vulnerable (during the evaluation).

To answer RQ4 we assign customize weights related to vulnerability severity on the vulnerable components. This can potentially give a higher value on the components with higher CVSS and result in better severe vulnerability predictions. More precisely, we examined three strategies; *CVSS* using the CVSS values as training weights; *Severe* using a weight of two for the components with severe vulnerabilities, while weight of one for the rest; *CVSS & Buggy* similar to CVSS strategy, by considering buggy files as vulnerable with weight of one.

It is noted that the CVSS & Buggy approach attempts to further tackle the class imbalance problem by augmenting the vulnerable component set with defective components, as suggested by Shin *et al.,* [171]. We thus, mined defects for the programs under analysis by searching the commit messages for bug identifiers. Then, by considering defects as vulnerabilities (only on the training phase) and assigning them a much lower weight than that of vulnerabilities we attempt to better tune our classifiers. We evaluate the methods by repeating the analysis of RQ3 on the specific weighted training sets.

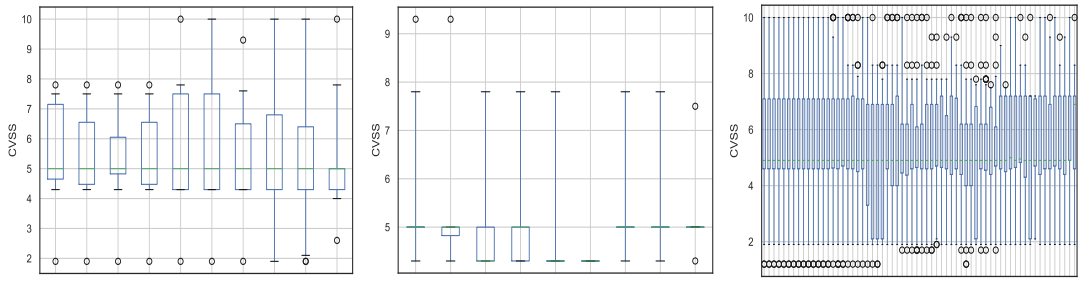
To answer RQ5 we evaluate the prediction ability of the studied methods by comparing the $PredP$ of the vulnerable components before and after their patching obtained from RQ2, as described in Section 9.4.3.

To answer RQ6 we evaluate the ability of the methods by training on the set of known vulnerabilities (at release time) and evaluating on the whole set of vulnerabilities. The evaluation is performed by repeating the analysis of RQ2 with altered training sets.



(a) Openssl (b) Wireshark (c) Linux Kernel

Figure 9.1: Ratio of vulnerable components per release (RQ1).



(a) Openssl (b) Wireshark (c) Linux Kernel

Figure 9.2: Distribution of severities, per release (RQ1).

9.5 Results

9.5.1 RQ1: Prevalence and Severity of Vulnerabilities

Figure 9.1 shows the ratio of vulnerable components per release for the three considered projects. This data shows that the ratio of vulnerable components is between 1-7% for all the systems we studied. Such a ratio indicates a largely unbalanced dataset, which can be challenging for the prediction modelling methods. We also observe that the number of vulnerable components rises up each major release.

Figure 9.2 shows the distribution of the severities (CVSS values) of vulnerable files. We see that for all three projects, the distribution of severities is relatively constant over the releases. The results of Wireshark seem more disparate, which is due to the few data points involved.

9.5.2 RQ2: Performance of Vulnerability Prediction

Figure 9.3 records the distribution of the predictions' evaluation metrics (MCC, Precision and Recall) for all methods and subjects we consider. Interestingly, the results are consistent for all the studied subjects, with the *Bag of Words* and *Function Calls* methods achieving the best results, i.e., yield the highest MCC. Code Metrics is the most precise method, but overall this advantage is small. With respect to recall, *Bag Of words* clearly outperforms the other methods.

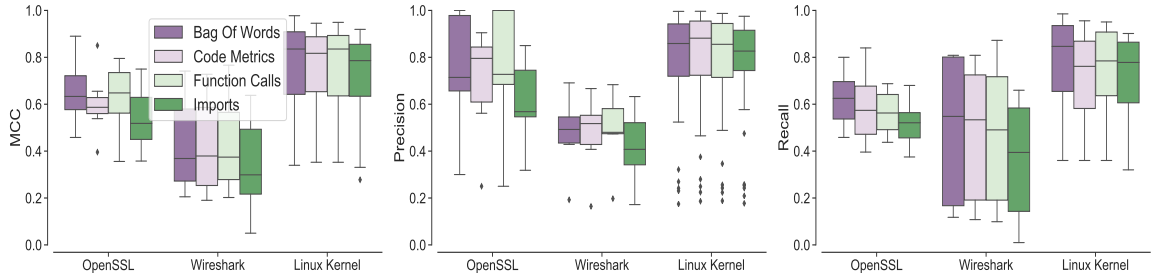


Figure 9.3: Performance metrics for all vulnerable components (RQ2).

We also observe very good results for the Linux subject with median values of MCC, Precision and Recall being above 0.8 for all approaches (except the recall of CodeMetrics). The results for OpenSSL are lower than those of Linux, but overall good. The results of Wireshark are much lower with Recall and Precision values less than 0.5. This is mainly due to the low prevalence of vulnerabilities in the project releases of Wireshark as we showed in RQ1.

To further validate our findings, we perform inferential statistical analysis on our results. The lower triangular of Table 9.3 records the p-values and the effect size measures $\hat{A}_{12}$. Focusing on the white cells, we can observe that all the p-values are significant and all $\hat{A}_{12}$ are lower than 0.5 indicating that *Bag of Words* achieves the best prediction results (though without sizeable difference).

Overall, statistical analysis confirms the box plots observations: there is a small degree of variation in the performance of the different approaches over different programs. Nevertheless, based on our data, we can conclude that *Bag of Words* offers the best performance.

9.5.3 RQ3: Performance of Severe Vulnerability Prediction

Figure 9.4 presents the results related to severe vulnerabilities. As expected, performances are decreased as we focus on fewer instances, *i.e.,* only on severe vulnerabilities. Yet, this analysis shows that the relative performance differences between the approaches is small and the trends are slightly different from those shown in RQ2. We see that the *Bag of words* outperforms the other approaches on OpenSSL and Linux, but not in Wireshark. In Wireshark the *code metrics* is the best performing approach.

The upper cells of Table 9.3 indicate that *Bag of Words* is the best performing approach with all $\hat{A}_{12}$ score above 0.5. Though, the difference between the *Bag of Words* and *Function Calls* is so small that it could be characterized as unimportant.

Having compared the performance of the prediction methods, we evaluate the best performing one, *i.e.,* the *Bag of Words*, with the *score* and top-10 metrics. Figure 9.5, presents the *scores* and top-10 for all vulnerable components and only the severe ones.

Table 9.3: RQs 2-3

This table comprises two separate triangular sets of results. The lower triangle presents results comparing approaches based on all vulnerable components, while the upper triangle shows results comparing approaches based on severe vulnerable components. Each cell contains a p-value and a A_{12} (line, column) effect size measurement.

	Bag Of Words	Code Metrics	Function Calls	Imports
Bag Of Words	-	1.10E-03/0.53	1.38E-04/0.51	1.27E-04/0.53
Code Metrics	1.34E-13/0.44	-	0.9405/0.49	2.23E-02/0.50
Function Calls	5.72E-10/0.47	5.87E-05/0.54	-	0.1796/0.51
Imports	1.78E-19/0.41	1.80E-04/0.46	5.74E-15/0.43	-

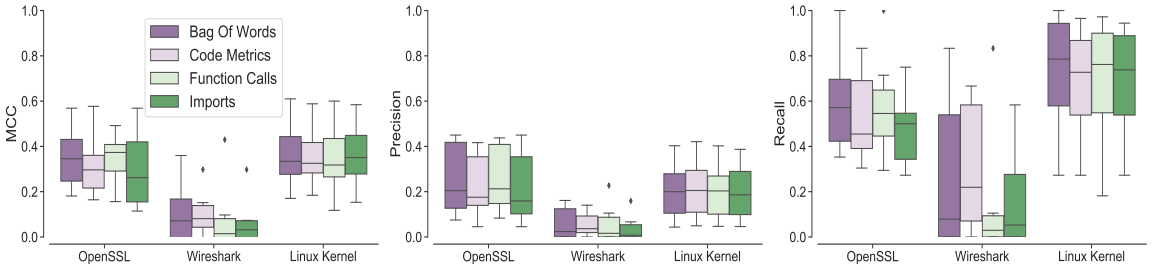


Figure 9.4: Performance metrics for severe vulnerable component (RQ3).

We observe that the score related to all vulnerable components is poor, especially for the Linux Kernel with a median above 0.9. This means that 90% of the components need to be inspected in order to inspect all vulnerable components. Yet when investigating the score of severe predictions, we observed an important decrease indicating that severe vulnerabilities are ranked much higher.

Regarding the top-10 metrics, we observed that for the Linux Kernel and OpenSSL almost all the first 10 files are vulnerable which indicates good performance. Predictions of the severe vulnerabilities are much lower, which means that most of the top-ranked components are not severe vulnerable.

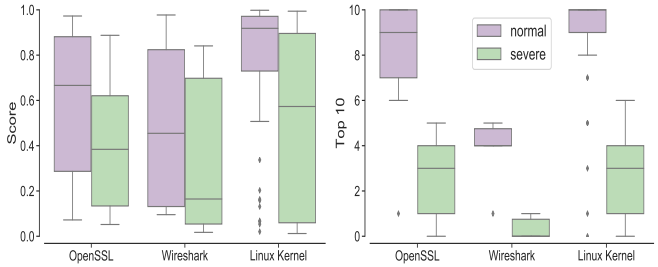


Figure 9.5: Score & top-10 metrics for *Bag of Words* (RQ2, RQ3).

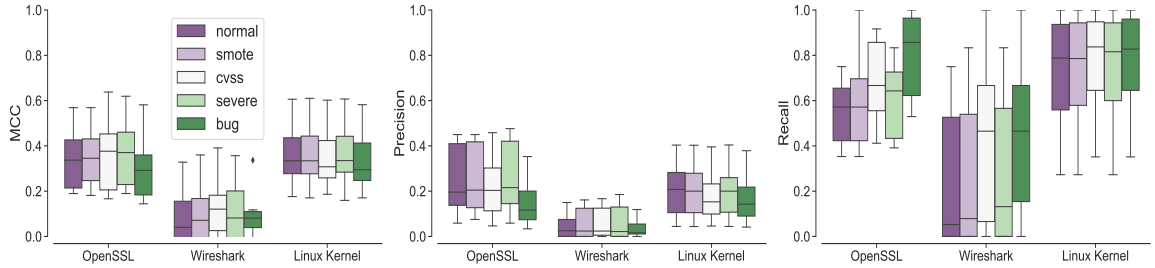


Figure 9.6: Performances for the five weighting strategies (RQ4).

9.5.4 RQ4: Augmenting Training Data to Improve Predictions

Figure 9.6 presents the performance of the three adopted strategies along with the standard one (all weights set to 1) with and without SMOTE. The figure presents the distributions of MCC, Precision and Recall for *Bag of Words*. We observe that the strategies have a small impact on precision (with the exception of the 'bug' strategy). This can be explained by the fact that only in the 'bug' strategy we are altering the learning phase by adding examples. Interestingly, we observe that it is possible to increase the recall with the CVSS and buggy strategies indicating that we can trade precision with recall. By inspecting the MCC values we see that the CVSS strategy cannot really improve the results as it improves them in OpenSSL and Wireshark but worsen them in Linux.

Interestingly, we observe that the adopted strategies help prioritize severe components by ranking them higher. This is shown in Figure 9.7 which depicts the score and top-10 metrics. In particular, we observe that score values have not changed, while the top-10 values have significantly improved. In the normal case 4, 9 and 9 out of the top-10ranked components in the three systems we study are vulnerable, with 1, 3 and 3 of them being severely vulnerable, while 'severe' strategy improves to 1, 5 and 7.

Overall these results suggest that it is possible to prioritize by ranking at the top places the most severe vulnerabilities but not to prominently improve prediction performance. A potential explanation is that the CVSS values are human computed scores, which are non-deterministic. Another explanation can be that we still train on all vulnerable instances. Nevertheless, it is still interesting to see that a model based on defects performs well in OpenSSL. A potential explanation of this is that due to the nature of OpenSSL, its bugs have stronger links with vulnerabilities than in the other projects.

9.5.5 RQ5: Sensitivity to Vulnerability Fixes

Figure 9.8 shows the difference in probability of a component to be vulnerable before and after a vulnerability fix. We expect this value to be negative since fixed components should be less likely to be vulnerable. However, our results show that this is not the actual case.

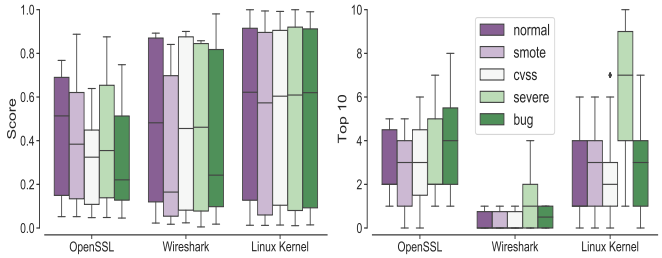


Figure 9.7: Score and top-10 metrics for *Bag of Words* (RQ4).

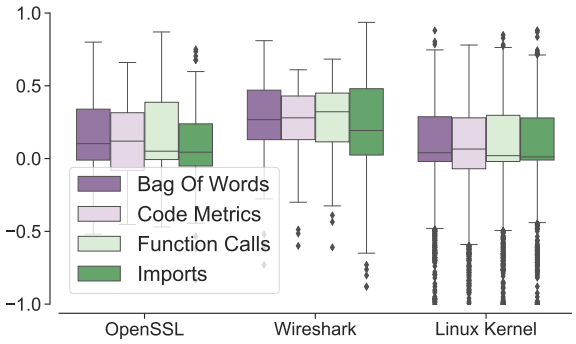


Figure 9.8: Vulnerability probability before and after vulnerability fix (RQ5)

This can be seen as a sign that prediction models indicate components that provide good opportunities for vulnerabilities to emerge. In view of this, the fact that the model points out to components where vulnerabilities observed indicates that the model somehow does what it is supposed to do. Nevertheless, what the prediction model actually does is to learn from features that are related to the vulnerabilities such as dangerous function calls. These features are unlikely to change much when vulnerabilities are fixed.

9.5.6 RQ6: Sensitivity to Mislabelling Noise

This RQ investigates the practical applicability of the prediction models. Thus, it investigates the predictions that one can achieve by using the reported vulnerabilities at the software release time. This means that the training happens on the reported, at the release time, vulnerabilities. This fact introduces noise and bias on the prediction methods as vulnerable components that have not been reported at that time are considered (labelled) as non-vulnerable. Additionally, training on the available information results on an overall small number of vulnerable components, which makes the problem severely unbalanced.

Figure 9.9 shows our results. 'experimental' results are those obtain with the knowledge of all vulnerabilities (all vulnerable components are labelled as vulnerable and all non-vulnerable are labelled as non-vulnerable). 'realistic' results are those obtained with the restrictive knowledge of vulnerabilities (labelled vulnerable components are those that have been reported at the release time, while the rest are labelled as non-vulnerable).

Finally, 'noiseless' results are those obtained when removing the vulnerable components from the training set (labelled vulnerable components are those that have been reported

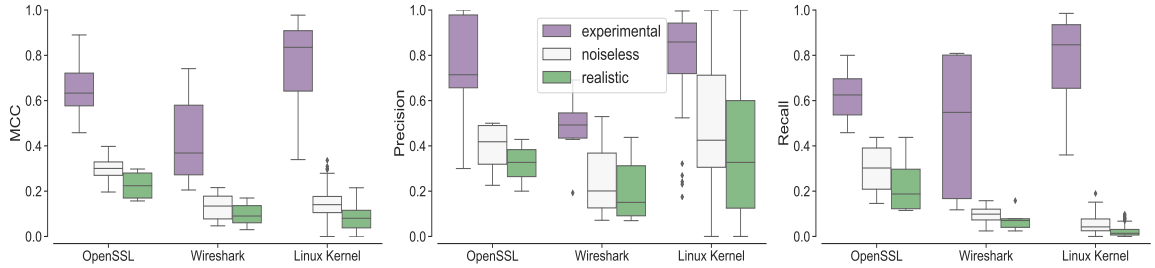


Figure 9.9: Prediction performance with subsets and noisy data (RQ6).

at the release time, vulnerable components related to unreported vulnerabilities are removed and the rest are labelled as non-vulnerable). This 'noiseless' results allow controlling for misclassification and unbalanced.

The figure shows a drastic drop-of-all-performance indicators between the 'experimental' set and the 'noiseless' set. This drop is attributed to the reduction of data in the minority class which creates an even more unbalance training set, whereas the smaller difference between 'noiseless' and 'realistic' corresponds to the effect of the noise introduced by misclassifying actual vulnerable files as non-vulnerable in the training set. We observe that this effect is smaller in comparison to the other when it comes to comparing the 'realistic' set with the 'experimental' set.

9.6 Threats to Validity

Potential defects in our FrameVPM framework may unintentionally influence our results. To reduce this threat, we carefully inspected our code and tested it. Threats may also arise due to the classifiers configuration. All classifiers were configured with the hyper-parameters setting of WEKA: although using a same default setting allowed us to compare all the techniques on a level playing field, performing hyper-parameter tuning could further improve performance [160, 117, 61, 175].

Also, we did not use the SVM classifier as in the study of Neuhaus *et al.,* [133] due to technical problems with WEKA when using it on the bag of words and metrics methods. Nevertheless, we achieved better results with RF than with the SVM for the particular case of Neuhaus *et al.,* Additionally, previous research on bag of words [161] and metrics [167] showed that RF and LR performed better than SVM.

Finally, it is not possible to claim that the results generalize beyond the subjects studied. However, to reduce this threat, we studied three large open-source systems with a large number (3-4 times larger than in previous studies) of real (reported in NVD) vulnerabilities.

9.7 Related Work on Bug Severity Prediction

Previous work has focused on the construction of vulnerability prediction models but no previous study has investigated the severity of vulnerabilities, while bug severity has been considered in previous study. This section offers an overview of such studies.

Menzies and Marcus [124] proposed a rule learning technique to assign severity levels to bug reports. They tokenize bug reports, perform stop word removal, stemming and use term frequency-inverse document frequency, information gain and a classification approach named Ripper rule learner [51].

Lamkanfi *et al.,* [108] used text mining to assign severity levels to bug reports. This method extracts word tokens, processes them, and feeds them into a Naive Bayes classifier. Different from Menzies and Marcus, they classify coarse grained bug severity labels, i.e., severe/non-severe, based on Bugzilla severity classesand omitting normal severity bugs from their analysis. Subsequently, Lamkanfi *et al.,* [109] extended their work by exploring additional classification algorithms and showing that Naive Bayes classifier performs better than other approaches on a corpus of 29,204 bug reports.

Chaturvedi and Singh [42] observed that specific machine learning techniques are applicable in determining the bug severity levels with reasonable accuracy and f-measure. The same authors [43] predicted the severity of close source (NASA's PITS Projects and Issue Tracking System) and open-source projects (Eclipse, Mozilla and GNOME) using the same techniques and found that SVM works reasonably well for open source and NB for closed source projects.

Tian *et al.,* [179] proposed an information retrieval based the nearest neighbour method. Similar to the work by Lamkanfi *et al.,* [108, 109] they consider bug reports on Bugzilla repositories of various open-source projects and benchmarked their approach with that of Menzies and Marcus [124] on a corpus of more than 65,000 bug reports. The results of this study revealed that the approach of Tian *et al.,* [179] achieved significant F-measure improvements.

Yang *et al.,* [195] proposed a method for the bug triage and bug severity prediction that uses historical bug reports for extracting topic(s) and then maps the bug reports related to each topic. The authors found that reports having similar multi-features (e.g., component, product, priority and severity) with new bug reports can be used to suggest the most appropriate developer to fix each bug and predict bug severity.

Zhang *et al.,* [198] proposed a concept profile-based severity prediction technique that analyses historical bug reports and builds concept profiles. The authors evaluated the performance of their proposal to use bug reports from the bug repositories of Eclipse and Mozilla Firefox finding that it works effectively for prediction of the severity of a given bug.

Tian *et al.,* [180] pointed out that the proposed approach to assess the performance of automated classification approaches, which factors in the noisy nature of bug reports. Armaghan *et al.,* [30] attempted to predict the severity level of bug reports opened in future releases/development cycles. They used a text mining to extract the summary terms and trained a classifier using these terms.

9.8 Conclusion

In this chapter, we presented the largest empirical study on vulnerability prediction models. This study done with FrameVPM shows that VPM can indeed reach good performance when trained on sufficient and accurately labelled data. However, performance becomes poor when considering realistic partial and miss-labelled data. The miss-labelled data together with the needle effects [171], vulnerabilities are the major factors behind this drop in performance. The study also shows that severe vulnerability prediction is possible and can be improved. A final finding is the poor performance of the said models to detect a fix of vulnerabilities. This confirms that models should be used as a guide for the review process but not as accurate detectors.

10

Wrapping Up Part III

In the second part of the thesis the emphasis was put on how to collect and better understand vulnerabilities. In this second part, the focus was shifted to the comparison of existing VPMs in order to address the second challenge. In the first Chapter 7, an exact independent replication of three of the main approaches was presented. This replication was performed on a dataset of Linux kernel vulnerabilities under different settings. Evaluating the ability of the models to distinguish either between bugs and vulnerabilities, either vulnerabilities in a largely unbalanced dataset. Overall, the results suggested that approaches based on Imports and Function Calls offered the best trade-off memory/time/performance while Bag of Words approach was the best performing one. Performing this study highlighted the fact that replicating and comparing existing approach is a hard and time-consuming task. To enable further research in the area to compare themselves with current state of the arts techniques, Chapter 8 introduced FrameVPM a framework to evaluate VPMs. This framework is then used in the Chapter 9 to perform the largest empirical study on VPMs. This study confirmed that existing approaches can reach great performances in experimental setting but face significant drop in performance when facing more realistic with partial and miss-labelled data. Additionally the study validated that the output of the models should be used as a guiding process and not as indicators as the model often fail to detect fixes. On the other hand, the study also showed that severe vulnerability prediction is possible and can be improved.
The next part will focus on the "Naturalness of Software".

Part IV

Naturalness of Software

11

An Empirical Study on the Use of Naturalness on Source Code

Source code like natural language is composed of small snippets that are repetitive and follow predictable patterns. This phenomenon called by researchers as naturalness *of the language can be used to build language models. Recent research shows that language models, such as n-gram models, are useful at a wide variety of software engineering tasks, e.g., code completion, bug identification, code summarization, etc. In this part, we aim at building a VPM approach based on them, yet, the use of such models requires the appropriate set of numerous parameters. Moreover, the different ways one can read code essentially yield different models (based on the different sequences of tokens). In this chapter, we focus on n-gram models and evaluate how the use of tokenizers, smoothing, unknown threshold and n values impact the predicting ability of these models.*
This chapter is based on work that has been published in the following paper:

- *On the Impact of Tokenizer and Parameters on N-Gram Based Code Analysis (ICSME'18)*
  M Jimenez, M Cordy, Y Le Traon, M Papadakis

Contents

11.1 Introduction

NLP [95] techniques realize the assumption that humans exploit partially the complexity of the language by following particular norms. Thus, natural language is composed of small snippets that are repetitive and follow predictable patterns. This phenomenon is called by researchers as the *naturalness* of the language. Recently, the study of Hindle *et al.,* [77] showed that source code follows the same trend. This means that code (small code snippets) is also repetitive and predictable.

This observation paves the way for using statistical language models for code analysis. Among the possible usage of those models, we are especially interested in applying them to build a VPM approach. This application seems promising as such models have been proved useful for providing code completion [77], complementing static bug finders [156, 186], automatically generating code explanations [82] and synthesizing code from natural-language specifications [114]. Many other interesting applications exist and have been surveyed by Allamanis *et al.,* [25].

N-gram models operate by *tokenizing* documents, *i.e.,* breaking these into words, and calculating the number of times every sequence of n words appear in a given document corpus. Based on that they estimate the likelihood that a given sequence appears. Their application requires setting parameters such as the length n of the sequences, the unknown threshold (ignoring tokens that appear fewer times than the threshold) and the smoothing technique (scoring unknown sequences).

In the case of code, the appropriate way of tokenizing documents *e.g.,* source files is not evident as code can be processed in many ways. Naturally, one can read code as any text document, that is, typically from left to right and top to bottom. However, developers tend to follow the flow of the program (that is not necessarily sequential) by taking advantage of the code characteristics, *e.g.,* the grammar (programming language) used, while automated tools like compilers rely on program abstractions such as flow graphs and AST.

Overall, the prominent use of n-grams for source code analysis requires setting a number of parameters. Previous work in the area set them as in the case of natural language. However, given the differences between code and natural language, it is imperative to revalidate and tune the application of the language models in the context of code. This is because there is no empirical evidence related to the generalization of the existing results, from the natural language field to the source code analysis field. For instance, choosing the most appropriate smoothing technique (way of treating unseen sequences), is not evident due to the vocabulary and structural differences between code and text. Moreover, as there is a plethora of parameter possibilities, there is a need for checking the sensitivity of the models w.r.t. these choices and the overall impact of the untuned parameter selection.

We therefore, investigate the effect of code representation on language models. We use the tokenizers presented in Table 11.1. The first two correspond to "an outsider's point of view", *i.e.,* the reader does not have knowledge about the code structure. The next two correspond to "a developer's point of view", *i.e.,* the reader knows the grammar of the language. Whereas the last four correspond to "the automated tools point of view", *i.e.,* the reader is a parser transforming code to a representation like AST.

The differences of 'UTF' with 'UTF woc' and 'Java Parser' with 'Java Parser woc' are due to the way comments are handled. This differentiation is useful as comments can generate noise in the models. The last four tokenizers differ from the way an AST is processed (typically in depth-first or in breadth-first order), and whether or not the AST is pruned of redundant nodes. This is important as language models work with sequence of tokens, which in this case are the different orders that one can visit the tree representations.

In this chapter, the problem of tuning n-gram models to a given purpose is addressed by evaluating 120 different configurations of n-gram models (6 n values, 4 smoothing techniques, and 5 unknown thresholds) combined with the above-mentioned tokenizers. These configurations are implemented in a tool publicly available presented in Section 11.8. In the first part of the evaluation, we assess the capability of the configurations to capture regularities within 20 open-source Java projects.

For each project and configuration, we compute the cross entropy of the project. The cross entropy is a measure commonly used in NLP to assess the efficiency of a language model. Intuitively, it represents how "surprised" a model trained on a given set of documents is when it encounters an unseen one. Therefore, the best configurations should give the lowest entropy, given that one can find local regularities within a given project [77, 181]. This allows us to check the sensitivity of the approach with respect to the studied configurations.

Comparing tokenizers is tricky as each involves its own specific building blocks. Thus, entropy values cannot be compared directly. We bypass this problem by comparing the tokenizers according to the relative entropy differences and the entropy-based rankings of source code files, i.e., we measure whether models judge and select the unlikeliest or likeliest files similarly.

To further strengthen the study, in the third part of the evaluation we consider a particular experimental scenario and demonstrate that the use of different tokenizers leads to contradictory conclusions. We thus investigate whether buggy files are more likely to have higher cross-entropy values than non-buggy ones, and whether fixing bugs results in a reduced file cross-entropy. These objectives were inspired by the study of Ray et al. [156] and represent a concrete example of research that can be influenced by n-gram model tuning.

Overall, the study involves 20 large open source Java projects and a dataset of 3,800 bugs. The results show that the Modified Kneser-Ney smoothing technique performs best. Choosing an n value equal to 4 or 5 seems to be the most appropriate choice for all tokenizers. Perhaps more importantly, we find a large disagreement between the tokenizers and show that not all of them are appropriate for particular problems. We further demonstrate this by investigating the link between entropy and bugginess, showing that only 2 out of our 8 tokenizers are capable of exploiting this link. Interestingly, our results show that the closer, to human perspective (unprocessed code), the used tokenizer is, the better the model is at detecting the effects of bug fixes. In this regard, tokenizers treating code as pure text are thus the winning ones.

Table 11.1: Studied Tokenizers

Tokenizer	Representation	Delimiter	Technology	Specificities
UTF	Raw	Non-Alphanumeric	Terrier	-
UTF woc	Raw	Non-Alphanumeric	Terrier	without comments
Java Parser	Raw	Java Grammar	Java Parser	-
Java Parser woc	Raw	Java Grammar	Java Parser	without comments
AST Depth First	AST	Node	Java Parser	depth first
AST Breadth First	AST	Node	Java Parser	breadth first
Pruned AST Depth First	AST	Node	Java Parser	Pruned, depth first
Pruned AST Breadth First	AST	Node	Java Parser	Pruned, breadth first

In summary this chapter makes the following contributions:

- It identifies and explores the impact of different parameters on the predictability of the n-gram models for code.
- It demonstrates large disagreements between the predictions of models that use different tokenizers.
- It provides evidence that untuned n-gram models have the potential of biassing research conclusions.

11.2 Background on N-Gram Models

Language models operate on sequences of words and compute their probability distribution. In code analysis, such sequences are the code fragments such as source files, Java classes or specific code lines. Words are the constituent tokens of the code fragments. Let $s = t_1, \ldots, t_m$ be a finite sequence of tokens. We denote by $P(t_1, \ldots, t_m)$ the non-zero probability that can be estimated for s by a given language model. The model is first *trained* on a set of sequences, named the *training corpus*. The training process determines the probability distribution of the known sequences, which in essence compose our model. The distribution typically results from the computation of the maximum likelihood estimates, that is, the probability of a (sub-)sequence is given by the number of times it appears in the training set divided by the number of (sub-)sequences in the set.

N-gram models are a particular type of language models that are fast to train and easy to use. Their origin can be traced back to Shanon's work [163] that presented the task of guessing the next letter in a text. Such models statistically estimate the probability that a token follows a given preceding sequence. Accordingly, the probability of a sequence is defined as the product of the probability of each token to follow its prefix. Thus, $P(s) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2)\ldots P(t_m \mid t_1\ldots t_{m-1})$. N-grams also assume a Markov property of order $n-1$. Thus, the probability of occurrence of a token in a sequence depends on the $n-1$ previous tokens, i.e., $P(t_i \mid t_1\ldots t_{i-1}) = P(t_i \mid t_{i-n+1}\ldots t_{i-1})$. Then, the probability of a sequence becomes a product of n-sized conditional probabilities. For example, for $n = 3$ the probability of s is given by $P(s) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2)\ldots P(t_m \mid t_{m-2}t_{m-1})$. Following the above equation, an estimate of the probability of s is the product of estimates for its constituent conditional probabilities (based on the training corpus).

A maximum likelihood estimate for $P(t_i \mid t_{i-n+1} \dots t_{i-1})$ is obtained by dividing the number of occurrences of $t_{i-n+1} \dots t_i$ by the number of occurrences of the prefix $t_{i-n+1} \dots t_{i-1}$.

Interestingly, the training corpus is not the only one that can impact the utility of an n-gram model. There are multiple parameters that can influence these results, with the most obvious one being the size n. To evaluate alternative models, one can carry out intrinsic evaluations to measure the performance of the models on some unseen data. In our case, this *test corpus* consists of code fragments that were not part of the training corpus. Then a model m_1 has a higher utility than a model m_2 if it can better predict the sequences of the test corpus. In other words, m_1 assigns a higher probability to the test corpus. In practice, one does not use the raw probability but rather rely on a derived measure named *cross entropy*. It is given by :

$$H(s) = -\frac{1}{m} \log P(s)$$

which, for an n-gram model of size n, is equivalent to

$$H(s) = -\frac{1}{m} \sum_{i=1}^{m} \log P(t_i \mid t_{i-n+1} \dots t_{i-1})$$

A lower cross-entropy thus means a better model. Intuitively, the cross-entropy indicates how "surprised" the model is when confronted to s. More formally it describes the average number of bits required to encode the data from the test set that have a distribution P using the code that is optimal for a distribution Q (the model built using the training set).

The choice of the n-gram size n can have a major impact on the model utility. Indeed, a higher n allows the model to better discriminate the sequences of tokens. However, it takes a longer time and more memory to train since more sequences have to be considered when computing the conditional probabilities.

Another point that can influence the model is the way it deals with *unknown words*. It may indeed happen that the model encounters some tokens (in the test corpus) that never appeared in the training corpus. The probability of this token is thus zero according to the model, which leads to an infinite cross entropy. In source code, this problem typically arises when new variable names are introduced. Of course, it is unrealistic to consider all potential variable names. The *vocabulary* of our model is thus *not closed*.

A common way to deal with this issue is to replace all words with less than k occurrences in the training corpus with a special token `<UNK>` (where $k > 0$). Since `<UNK>` occurs in the training corpus, the model estimates and assigns some probability values for this token. Then, each time an unknown word appears in the test corpus, the model interprets it as `<UNK>` and assigns it a non-zero probability. The aforementioned parameter k, named the *unknown threshold*, obviously affects the quality of the model since it modifies the estimated probability value of every token.

A similar problem occurs when dealing with *data sparsity*. As it is rather unlikely to observe every possible sequence of tokens in a training corpus, it might happen that sequences absent from the training corpus appear for the first time in the test corpus. This is even more common than unknown words, especially for higher-sized n-grams that work with long sequences. To prevent the model from assigning zero probabilities to these sequences, several *smoothing* techniques have been proposed. Intuitively, smoothing reserves a part of the probability mass for the unseen sequences, and estimates a probability for known sequences based on the rest of the probability mass. Smoothing has the effect of improving the accuracy of the models, especially in the case of probability estimated from few counts.

There are many smoothing techniques but we only focus on the four most popular ones. For additional details on the subject please refer to the comprehensive survey of Chen and Goodman [47]. We study the following four techniques: Witten Bell [33, 188], Absolute Discounting [134], Kneser Ney [103] and Modified Kneser Ney [47].

Witten Bell was first introduced for text compression, but it can be used for smoothing language models as well. It is an instance of another smoothing technique called Jelinek Mercer [86] where the n-th order smoothed model is defined recursively as a linear interpolation of the maximum likelihood for n-th order and the (n-1)th order smoothed models. This technique uses as λ the probability of observing an n-gram for the first time, i.e., the number of n-grams appearing more than once over this number plus the total count of n-gram.

Absolute Discounting involves an interpolation between higher order and lower order. Instead of multiplying the higher order by a computed λ, a fixed discount is subtracted from it.

Kneser Ney is an extension of Absolute Discounting with a cleverer way of computing the discount, based on the idea that lower-order models are significant only when the number of occurrences is small or zero in the higher-order model.

Modified Kneser Ney is a further improvement that uses three different discounts depending on the number of occurrences of the considered n-gram.

Like the unknown threshold, the choice of a specific technique is important as it impacts the cross entropy returned by the model. Earlier work on software naturalness [77] argue that *Kneser-Ney* is the most appropriate, but have not presented detailed experiments confirming this claim. As we will see, our experiments fill this gap and empirically evaluate the different techniques w.r.t. other parameter values.

11.3 Research Questions

Our aim is to investigate how the different parameters involved when building n-gram models impact the source code analysis tasks. In particular we seek to investigate the parameters related to the size n, the smoothing technique and the way code is tokenized. This is important as the factual differences between natural language and source code have not been exploited. For instance, natural language almost always flows sequentially, while the source code includes many conditional jumps, which may necessitate a different analysis. Therefore, the use of different tokenizers should play a major role on the model's utility. Thus, our first research question is:

RQ1: What is the impact of the different parameters on n-gram models when used for source code analysis? Is there an optimal configuration for all tokenizers?

We answer this question by computing, for 8 different tokenizers, the average cross entropy of 20 Java projects using 24 sets of parameter configurations, *i.e.,* 6 n values * 4 smoothing techniques. Then we check whether optimal parameters stand out across all the tokenisers.

Measuring the cross entropy provides an insight into the relative performance of studied configurations. However, this information, i.e., low or high entropy values, says nothing about the use of different tokenizers for distinguishing source code files with respect to naturalness of code. Indeed, cross entropy measures the distance of the test corpus from the generated model with respect to the involved building blocks of the model trained on the training corpus. Different tokenizers rely on different views of the code and hence result in models trained on different building blocks. Since the building blocks differ, we cannot directly compare the tokenizers with cross entropy.

To bypass this problem, we compare the tokenizers with respect to the relevant information they provide, *i.e.,* their ability to distinguish and rank source code files (natural and unnatural ones). Hence our second research question regards the information that different tokenizers learn:

RQ2: Do models built based on different tokenizers learn the same information for the same code?

To answer RQ2 we consider specific configurations that return good results (in terms of lower cross entropy). Then, we measure the relative agreement between the tokenizers, by computing the correlation between the cross-entropy values they provide on the source code files we study. We deem this comparison as valid as it measures the relative volume of agreement between the entropy value differences and their relative rankings with respect to a set of source code files. We also consider the correlation between these entropy values and the number of lines of code in order to check the effects of size on our results. Strong correlations indicate a large agreement between the tokenizers, while weak correlations indicate a disagreement.

As we will see later, our results show a large disagreement between tokenizers. Yet, it is unclear whether these differences are capable of impacting the findings of research studies. Thus we ask:

RQ3: Does the use of different tokenizers has the potential of impacting research results?

To answer this question, we set a simple experiment investigating whether buggy files are more likely to have higher entropy values than non-buggy ones and whether fixing bugs results in a reduced file entropy. These objectives were inspired by the study of Ray *et al.,* [156], which investigates the link between buggy lines of code and naturalness (and the impact of bug fixes on it).

Finally, we investigate the use of a 'special' parameter of the n-gram models. This is the unknown threshold k, which determines the confidence on the estimations made by the models. This is a special parameter as it involves a trade-off between the accuracy of the model and the information it captures. Thus, by setting the threshold at higher values we get more accurate but also more coarse-grained models. Therefore we investigate:

RQ4: What is the impact of setting the unknown threshold at different levels?

We examine this issue by using 5 unknown thresholds k and measuring their impact on both entropy and results of RQ3.

11.4 Methodology

In this section, we first introduce the dataset we used in all of the experiments. Then we present the toolchain that we developed to build n-gram models out of the source code and carry out analyses on these models. Finally, we describe the evaluation process.

11.4.1 Test Subjects

To answer the research questions, we rely on data gathered from 20 open source software from the *Apache Commons* project [59]. Apache Commons comprises reusable open source Java software projects which are intensively developed and maintained. At the time of writing this paper, a query about "org.apache.commons" on Github returned close to 7,000,000 different Java files. This indicates that the selected projects are popular.

Table 11.2: Dataset Statistics

Project	Latest	Files	kLoC	Versions	Bugs
BCEL	6.1	488	75	5	93
BeansUtils	1.9.3	257	72	18	155
CLI	1.4	50	12	6	91
Collections	4.1	525	118	12	186
Compress	1.15	329	70	18	309
Configuration	2.2	457	125	15	325
CSV	1.4	28	8.4	5	67
DBUtils	1.7	92	15	8	23
EMail	1.4	47	12	8	51
FileUpload	1.3.3	54	10	10	67
IO	2.5	227	55	14	213
JCS	2.2.1	562	102	6	102
Jexl	3.1	108	23	8	126
Lang	3.6	318	141	20	567
Math	3.6.1	970	218	16	830
Net	3.6	270	59	20	246
Pool	2.4.2	79	24	22	154
Rng	1.0	124	14	1	3
Text	1.1	104	25	2	38
VFS	2.2	382	52	4	214
Total	-	5,471	1,230	218	3860

Building our experiments around Apache Commons projects has many benefits. First, the projects follow strict development guidelines. This fact has a potential effect of improving the performance of language models (as repetitiveness is encouraged). We deemed this as an advantage as it reflects industrial settings where code conventions and implicit coding rules are followed throughout whole companies. Second, each one of the selected projects has its own usage context and implements different functionalities. This fact challenges our models, whose performance should generalize over all projects. This counterbalances the facility offered by coding conventions while further increasing the transferability of the results to real-world or industrial projects (indeed, a company typically develops software for slightly different application domains). Third, every Apache Commons project report their bugs on the same platform with similar reporting guidelines. This facilitates the creation of a bug dataset, which is required to address RQ3.

Apache Commons involve 41 projects. We selected 20 based on the following criteria: **Date of the last update**. A recent update indicates that the project is still active and of interest. It also means that developers continue to fix bugs.

Size of the project. A larger project increases the size of the training corpus, and thereby reduces the risk of overfitting for our models.

Length of project history. A long history usually implies a higher number of bugs to study and the possibility to observe whether results generalize over releases.

From the data of the 20 projects we select (recorded in Table 11.2), we analyse their Java files. Everything else was not considered as it contains irrelevant information for the study. In the first two and the last research questions we analyse the source code of the latest project release. For the third research question, which involves the analysis of bugs, we had to go back in the project history in order to identify and collect a large set of bugs. As a consequence, we had to gather multiple releases of the projects and identify the versions containing the studied bugs. We also had to identify the versions where these bugs were removed (fixed). As this process is really similar to the one of the Data7, we implement the following procedure:

1. We crawl the full commit history of the projects and identify all the commits that mention an issue ID.

2. For each issue ID we check whether the issue is mentioned on the issue tracker. We then check whether it refers to a bug and if so, we retrieve the affected version.

3. For each issue ID referring to a bug, we go back to the corresponding commits and get the list of files modified by these commits. Then we store those files and flag them as fixing the previous buggy version.

Table 11.2 presents the characteristics of the dataset as of end 2017. `Latest` is the latest version of the project at the time of writing, which is also the version we consider for RQs 1, 2 and 4. `Files` is the number of Java files in this version. `kLoC` is equal to the number of lines of code of the version (in thousands). `Versions` refer to the total number of versions of the project we studied in RQ3. Finally, `Bugs` refers to the number of unique bug-related issues retrieved by the above procedure.

11.4.2 N-Gram Model Configurations

Tokenizers We build n-gram models from source code using 8 tokenizers. Details about the tokenizers we use are presented in Table 11.1. These can be categorized in three main groups. The distinction between the groups regards the representation of the source code. Thus, the first group (first two rows in Table 11.1) comprises tokenizers that treat code as text and directly use it as input. In these cases a sequence of words is separated by delimiters. The second group (rows three and four in Table 11.1) comprises tokenizers that delimit code based on the language grammar. Tokenizers of the third group, (the last four rows in Table 11.1), are defined based on the AST representation of the code. Thus, these tokenizers perform the tokenization based on a serialized representation of the AST.

The tokenizers of the first group are standard UTF tokenizers. These are similar to the one used by the open-source search engine Terrier [176]. They split the text into groups of alphanumeric tokens while still keeping the delimiters in the sequence. The only difference between the two is that in the first considers the complete file, whereas the second ignores comments. This should give us insights about the sensitivity of the models wrt. code comments. Note that previous works tend to completely ignore code comments, *e.g.,* [77].

UTF tokenizers take all non-alphanumeric characters as delimiters. This fact may yield inappropriate tokenizations for specific cases. For example, variable names using an underscore are separated in three. A similar case is the float numbers (also separated in two words). This motivated the need for the second group of tokenizers. Thus, the `Java Parser` and `Java Parser woc` tokenize code according to the language (Java) grammar. This implies a correct identification of the Java tokens. Another effect of using the grammar is that the `Java Parser` considers a block of comments as a single token. Thus, ignoring comments should have a minor impact on our models. In our implementation we perform the parsing based on the Java Parser tool [149]. The reason behind this choice is that this tool facilitates the treatment of AST by providing specific data structures,*i.e.,* following the *visitor* design pattern, which was useful for implementing the remaining tokenizers.

The four tokenizers of the third group differ in the way they process the program's AST. The first two of them, serialize the complete AST (they print the type of every node as well as package, method, and variable names) in a specific order that depends on the visit strategy, *i.e.,* breadth first or depth first. The last two tokenizers of this group implement a pruned version of the serialization process (only the text of non-redundant nodes is considered). We consider a node as redundant when it does not directly correspond to a string in the source code. In other words, this node serves a structural purpose, e.g., every variable name is preceded by a node of type `NameExpr`. Studying all these alternative tokenizers helps us understand whether the visit strategy and the redundancies in the ASTs have a significant impact on our results.

Language Modeling To compute the cross entropy, we need to use some code parts as a training corpus and some others as a test corpus. We thus, define an n-gram model as a stateful service interface with two methods: (1) `train`, which takes as input a corpus and trains the model accordingly; (2) `entropy`, which returns the cross entropy of an input sequence of tokens based on the trained model. Our implementation uses the Kyoto Language Modeling Toolkit (Kylm) [131]. Kylm is an established tool developed in Java that provides all the functionalities needed for our experiments. Indeed, it allows one to specify the size n of an n-gram model, its unknown threshold, and the associated smoothing technique.

11.4.3 Research Protocol

RQ1 To address RQ1, we consider 24 configurations, which are the combinations of n values 2 to 7 with 4 smoothing techniques (absolute discounting, Kneser-Ney, modified Kneser-Ney, and Witten-Bell).

For each tokenizer t, project p and configuration c, we build an n-gram model parameterized by c, and compute the average cross-entropy over 10-fold cross validation of p's source code tokenized by t. This leads to a total of 3,840 cross validations.

Each iteration of a given 10-fold cross validation involves 90% of the source files for training the models, whereas the remaining ones compose the test corpus. We operate on a file-level granularity as it is common in defect prediction, and for simplicity when using the AST tokenizers. Indeed, in Java, ASTs are built by class, and a (public) class is commonly contained within one file. It is noted that for now, we set the unknown threshold k to 1, as the influence of the unknown threshold k is studied in RQ4.

RQ2 To address RQ2, we build an n-gram model m based on the configuration that is the most representative (identified in RQ1) and, for every tokenizer t, we make m compute the cross entropy of all source files tokenized by t. Then, we check whether there is a correlation between the cross-entropy values across each pair of tokenizers. This allows us to check whether tokenizers agree between them when comparing the source code files. We also verify the existence of a correlation between the number of Lines of Code (LoC) and the entropy values associated to each tokenizer to check whether our observations are influenced by the code size.

To perform the comparisons, we carry out two correlation tests. First, we compute the Pearson correlation coefficients to formally assess whether there is a strong linear relationship between the tokenizers, *i.e.,* the entropy values change similarly among the files when using different tokenizers). We also check this relation with the LoC. Second, we measure the ordinal association between the same variables using the Kendall's tau coefficients. Ordinal relations differ from the linear relations as they do not consider the size of the differences between the values. This allows determining whether the code files are ranked differently according to their entropy and LoC.

RQ3 To address RQ3 we investigate the influence of tokenizers on the findings of a research experiment we design. Thus, we investigate whether (1) buggy files tend to be more unnatural than non-buggy ones and (2) fixing a bug makes a file more natural.

For (1), we compute the entropy of each file successively in the release using all other files for training. This process ensures a common training and evaluation ground that is deterministic and reproducible. This way we avoid using large training corpus and focus on the relative differences between the files under analysis and the rest of the project. The idea is that the more improbable a file, w.r.t. the others of the same project, the more likely it is to be problematic. Future work includes the use of cross-project training or past-release project training.

Based on the entropy values, we can observe whether files flagged as buggy have indeed a higher cross entropy. Then for (2), we compute, for each buggy file, its cross entropy in the release just before the patch and after the patch. We use a model built on the last affected release – excluding the assessed file – and analyse the percentage of difference between the two cross-entropy values.

To see the actual impact of the tokenizers, we compare the conclusions that one can draw for the above experiments when using one tokenizer instead of another. In case of contradictions, we can conclude that the tokenizer choice is important (as a different choice may imply a different conclusion for a given task).

RQ4 To study, the impact of the unknown threshold, we repeat the analysis followed in RQ1, but for different thresholds. Thus, the following values of k are studied, 1, 2, 4, 8 and 16. This adds up 15,360 new 10-fold cross validation to the one done for RQ1. Then we study the impact of this parameter on the findings of RQ3. To do so, we measure the differences between buggy and non-buggy code and the impact on entropy when fixing a file (using a k equal to 8). Finally, we compare these results with those obtained in RQ3.

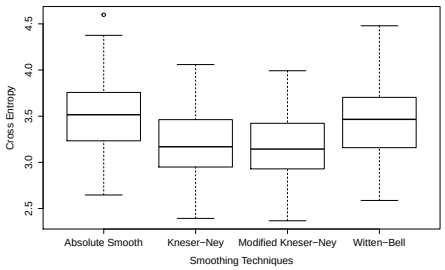


Figure 11.1: Cross entropy of source code files

When using different smoothing techniques for AST breadth-first tokenizer and $n = 4$.

Statistical Comparison To judge the significance of the observed differences we use standard statistical tests. We used the Wilcoxon signed-rank test to measure the arbitrariness of our results. We choose the Wilcoxon hypothesis test as it is non-parametric and thus, it does not make any normality assumptions. As it is typically performed, we adopt a significance level of 0.05, below of which we consider the differences statistically significant. To measure the size of differences we used the Vargha Delaney effect size $\hat{A}_{12}$, which quantifies the size of the differences (statistical effect size).

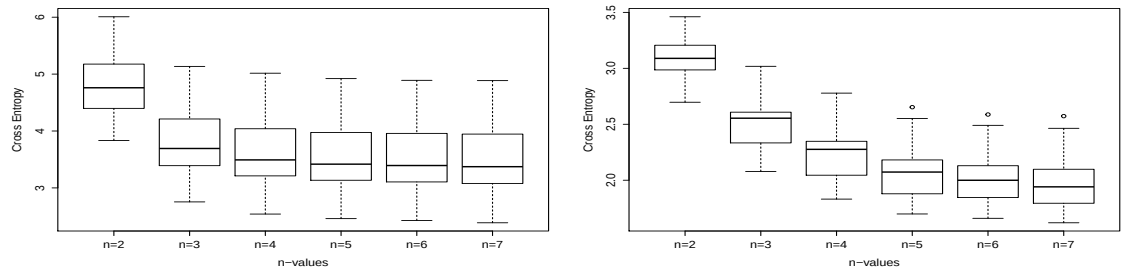
11.5 Results

11.5.1 RQ1: Optimal Configuration

We start the analysis by identifying the impact of the smoothing techniques on the entropy of the source code files. Our analysis is based on the principle that a smoothing technique giving lower entropy values than another one for the same files, tokenizers and n-values is preferable. We therefore computed every combinations of n-values, smoothing techniques and tokenizers to identify the most appropriate configurations.

The results are consistent across all tokenizers and n-values: they show that the Modified Kneser Ney smoothing is the most appropriate which is in line with what was found by the NLP community[47]. Although the difference with Kneser Ney is thin, it is statistically significant (using Wilcoxon signed-rank test) and has $\hat{A}_{12}$ values in the range from 0.50 to 0.53. Figure 11.1 presents an example of our data for the case of the AST breadth-first tokenizer and n equal to 4.

Having shown that the Modified Kneser Ney is the best smoothing technique, we turn to see the impact of choosing an appropriate n-value. Again, we have similar trends for the n-values across the different tokenizers. However, we observe that some tokenizers do converge faster than others. For instance, the UTF tokenizer stabilizes when n equals 5 while the depth-first tokenizer does not. Figures 11.2a and 11.2b demonstrate these results. We also observe that benefits of using n values higher than 4 is small, all the n-values result in statistically significant differences. In particular, all the tokenizes have $\hat{A}_{12}$ values in the range from 0.54 to 0.65 when comparing n=4 with n=5. These drop to 0.53 to 0.57 when comparing the n=5 with n=6. Therefore, the general most appropriate choices are the n=4 or n=5. The AST depth first and the "Java Parser" tokenizers are the only ones that continue to improve (slightly) beyond n=6.



(a) UTF tokenizer (b) AST depth-first tokenizer

Figure 11.2: Cross entropy of source code files

when using different tokenizers with Modified Kneser Ney smoothing for n-values in the range n=2..7.

Table 11.3: Correlations between tokenizers and number of lines of code

Upper (resp. lower) diagonal gives, for each pair, the median of the Pearson correlation coefficients (resp. Kendall's tau coefficients) over all projects.

	LoC	UTF	UTFw	JP	JPw	DF	BF	PDF	PBF
LoC		0.52	0.32	0.31	0.29	0.25	0.15	0.24	0.17
UTF	0.52		0.83	0.78	0.77	0.71	0.60	0.74	0.64
UTFw	0.32	0.66		0.90	0.90	0.88	0.79	0.91	0.81
JP	0.29	0.60	0.77		0.99	0.89	0.81	0.90	0.79
JPw	0.29	0.60	0.77	0.93		0.91	0.82	0.92	0.79
DF	0.30	0.57	0.73	0.76	0.79		0.85	0.96	0.78
BF	0.14	0.47	0.62	0.65	0.66	0.70		0.85	0.94
PDF	0.28	0.58	0.76	0.77	0.78	0.84	0.69		0.85
PBF	0.15	0.48	0.62	0.64	0.64	0.64	0.80	0.70	

11.5.2 RQ2: Tokenizer Correlations

The second research question examines the correlation between the cross-entropy values returned by the 8 tokenizers. We also consider the correlation between these values and the number of LoC. We computed the correlation coefficients for all files by considering every pair of tokenisers (and LoC). Table 11.3 summarizes the results. It gives the median of the coefficients over all projects for each pair. The upper triangle of the table records the Pearson coefficients, whereas the lower one is about the Kendall's tau coefficients. A higher coefficient means stronger correlation. All coefficients are statistically significant with a p-value lower than 0.05 in every case.

A first observation is that Pearson coefficients are higher than Kendall's coefficients for each pair of tokenizers. This indicates that large differences in cross-entropy are more likely to lead to an agreement between the tokenizers than smaller differences. The strongest correlations are in the case of JP with the JPw. If we exclude this case, the Pearson correlations are in the range of 0.60 to 0.96, while the Kendall ones are in the range of 0.47 and 0.84.

Another general observation is that the correlation between LoC and the cross-entropy values is generally weak (< 0.33). The only exception is the UTF tokenizer, which has a correlation with LoC of 0.52. This is due to comments. Indeed, UTF is the only tokenizer that tokenizes comments as any English text. The JP tokenizer includes comments as well, but considers a block of comments *e.g.,* the Javadoc of a method, as a single token. The impact of comments for this tokenizer is thus limited, as witnessed by the strong correlation between JP and JPw (Pearson 0.99; Kendall 0.93). Given the difference in the way code and natural language are written, it is expected that comments significantly increase cross entropy. Moreover, the number of comments is likely to increase with the number of lines of code. We also see that AST tokenization further reduces the correlation with LoC (≤ 0.25). Indeed, the number of tokens depends on the number of AST nodes, which is not necessarily proportional to LoC.

Interestingly, we observe that the differences between the ways the AST is visited, in a breadth-first or depth-first manner, plays an important role, as it gives the lowest correlation values. Generally, the breath-first tokenizers give in all cases, lower correlations than their depth-first counterparts. This indicates that breadth-first AST tokenizers capture the most different information than any other tokenizer. In other words, the disagreement with the other tokenizers is higher. This can be explained by the fact that the other tokenizers have inherently different views of the code, *i.e.,* structure-oriented versus sequence-oriented.

We also observe that there are no significant differences between AST tokenizers (regardless of the visit strategy) and their respective pruned variants. Redundant nodes thus have a limited impact on the captured information. More generally, it might imply that how the AST is constructed is unimportant, although this must be confirmed by additional experiments with alternative parsing tools.

Taken together, the results suggest that tokenizers indeed judge code files differently. They tend to agree on the majority of the cases but still they tend to disagree on a significant number of cases.

11.5.3 RQ3: Impact on results: Bug Analysis

Having confirmed that tokenizers from different groups learn (largely) different things, our third research question regards their possible impact on the findings of an experimental study. To answer this question, we investigate the hypothesis that bugs are linked with naturalness. We do so by checking whether unnatural files are more likely to be buggy than the natural ones. In case we find significant differences, we can conclude that a link between bugs and naturalness exists. However, this link might simply be the result of other (unknown) factors (such as the size of files, the defects' location or others). In other words, we need to show that entropy is linked with both presence and absence of bugs. To control for arbitrary factors, we check whether fixing a file reduces its entropy. In case we observe a reduction in most of the case, then we have strong evidence supporting our hypothesis, while in the opposite case we do not.

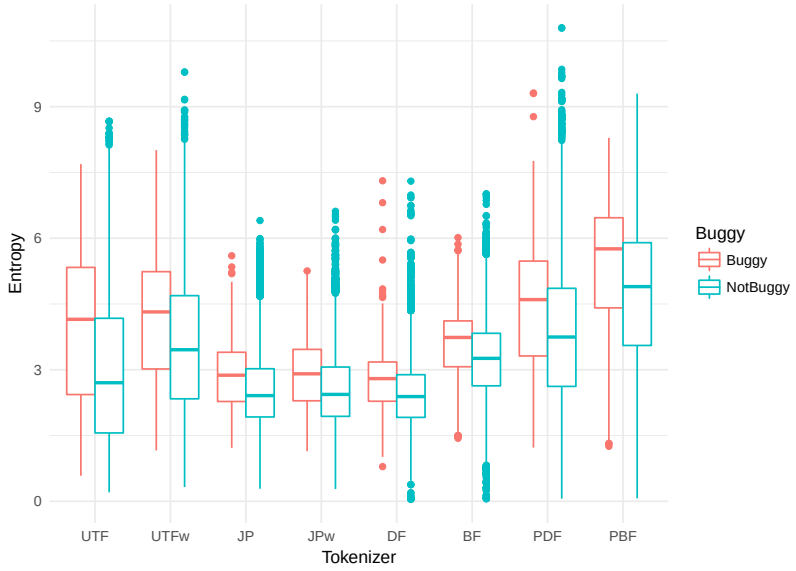


Figure 11.3: Comparison over all projects

Between buggy (red) and not buggy file (green) cross entropy.

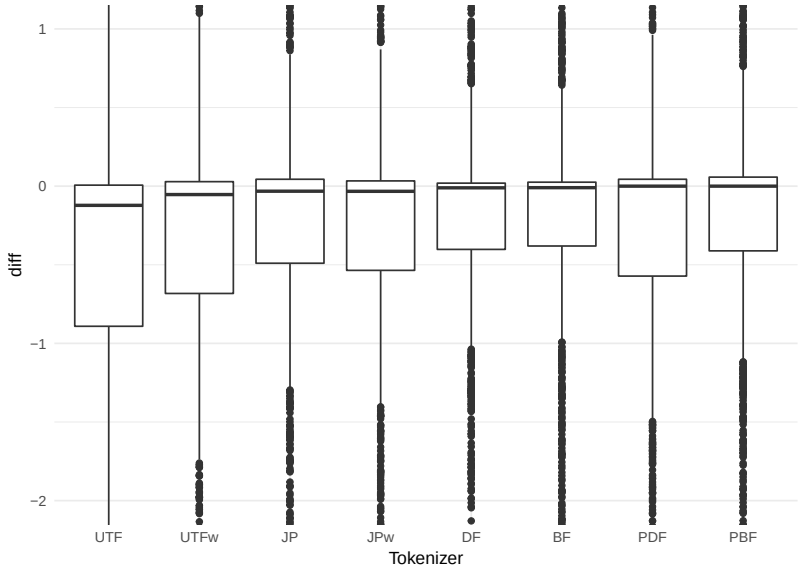


Figure 11.4: Difference in cross entropy

between the buggy version of a file and its fixed one per tokenizer.

Figure 11.3 reports the results for all considered project releases and tokenizers. From these we can make two main observations. First, buggy files have a higher entropy than their non-buggy counterparts regardless of the tokenizers used. This provides a first indication that our hypothesis might hold (this result is inline with the results of Ray *et al.,* [156]).

Second, UTF and pruned AST based models present the largest variance in entropy which could make them targets of interest when using them for prediction modelling.

Figure 11.4 presents the results obtained when studying the entropy differences between the buggy and the fixed versions of our files, following the procedure described in Section 11.4.3. We observe that in many cases, the cross entropy is indeed slightly reduced (values are below 0) after the fix process. However this does not for a clear majority (approximately 50% of the files have values higher or equal to zero, median values are 0). This means that fixing a file might reduce the entropy or might not, which in turn indicates that bugs appear in files that are unnatural but naturalness is not necessarily linked with the presence of bugs. Considering the starting point of naturalness *i.e.,* developers tend to write code that is repetitive, hence more natural, this means that bugs are located in files further from developers usual comfort zone.

Perhaps the most interesting observation is that the differences are more accentuated in the case of UTF tokenizers. These are the only models having median values below 0. As all other tokenizers have their median almost at 0, we can conclude that one can get evidence supporting our hypothesis, only by using the UTF tokenizers. We also statistically examine the differences and find that they are statistically significant with an effect size, close to 42% (when comparing the UTF tokenizers and the others). Interestingly, these cases are the only ones with both statistical significance and effect size differences. The difference between JP, BF and DF are not statistically significant whereas the difference between BF, DF and their pruned counterparts are significant (though with negligible effect size). To make these results clear, Figure 11.5 shows the level of agreement (on the impact of fix) between the tokenizers, *i.e.,* how frequently every pair of tokenizers agree that fixing a file results in reduced or increased entropy. From these results we see that tokenizers largely disagree on their judgements.

The above results imply that the closer, to human perspective (unprocessed code), the used tokenizer is, the more robust the n-gram model is in detecting the effects of a fix. Thus, only UTF tokenizers are robust in this regard. This is also interesting as the UTF tokenizers are not the ones with the lowest entropy.

To summarize, we found that tokenizers have the potential of changing the conclusions of a research study. We demonstrated that only 2 out of the 8 tokenizers are robust at detecting (as they should) the differences between buggy and fixed files. Therefore, researchers need to be cautious that their conclusions may change if they use different tokenizers. Moreover, our data suggest that the most prominent choice of tokenizer for bug identification is the UTF.

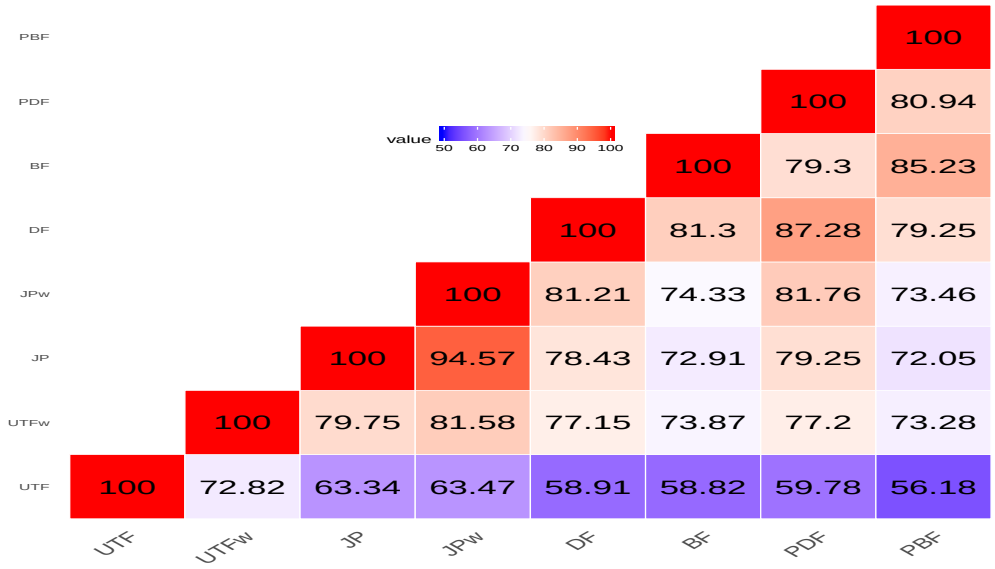


Figure 11.5: Percentage of agreement between tokenizers.

Values represent the ratios of files judged similarly by the tokenizers (increase or decrease).

11.5.4 RQ4: Setting the "unknown threshold"

In n-gram model related literature, a specific parameter called unknown threshold is often evoked, but has never been examined. Increasing this parameter may make the entropy lower but at the price of a less general model.

Figure 11.6 presents the results we obtain in the experiment of RQ1 while observing five different values of k. We observe a huge decrease in entropy as k increases. This means that the model copes better with low-count tokens. For source code, this could be explained by the fact that the model is removing variable or function names that are barely used. While this could be interesting in some situations, *e.g.,* when one is interested in general patterns or trends, it can have a negative impact on naturalness-based studies. In Figure 11.7, we present a comparison of the values obtained with two different k, *i.e.,* $k = 1$ and $k = 8$ for two of our tokenizers.

In Figure 11.7a we observe the reduction in entropy between the two values of k, yet the difference between buggy files and non-buggy ones is still clear. However, Figure 11.7b reveals that the reduction of entropy after a fix is compromised when increasing k. This is more interesting in the case of AST depth-first tokenizer, where for $k = 8$ the entropy increases in more cases than for $k = 1$. This can be explained by the fact that a high entropy is caused by unlikely tokens which are removed when using a higher k, gathering them under a common, more likely one. This, in turn, reduces the opportunity to observe the effect of a fix: If the new token introduced by the fix replace an unknown one but has a lower probability than the unknown the entropy could increase in some case.

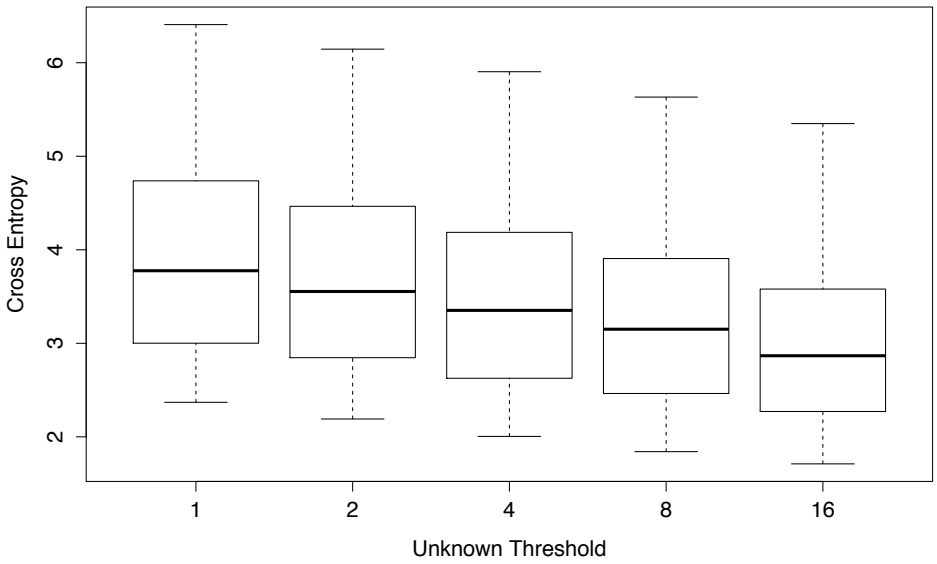
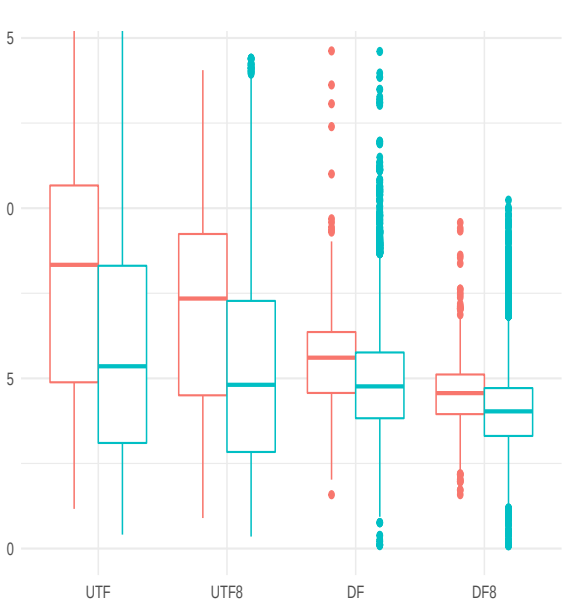
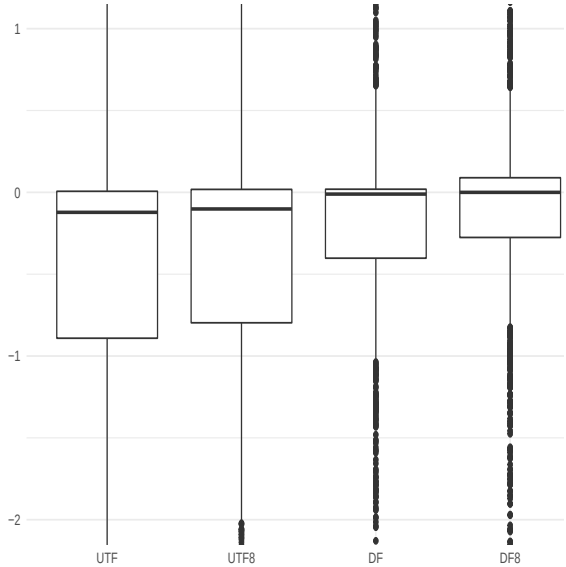


Figure 11.6: Effect of Threshold k on the Cross-Entropy



(a) Difference in entropy between buggy file(red) and non-buggy(green)



(b) Percentage of difference in cross entropy between the buggy version of a file and its fixed one per tokenizer

Figure 11.7: Effect of the unknown threshold

For the UTF and DF tokenizers when k are equal to 1 and 8

11.6 Threats to validity

The generalization of the result is again a threat to validity. In this chapter, we used Java projects from Apache Commons, which may not be representative. Similarly, the results might not hold on other programming languages. We choose the Apache Commons to gather a large variety of projects with different functionalities. Moreover, Apache is a large organization and follows a similar development process with many other organizations.

Similarly, we showed that tokenizer impact the n-gram models in Java. We expect a similar result on other languages as the basic differences between the sequences of tokens and ASTs appear in all languages. However, we still do not know whether n-gram models are similarly sensitive when using other languages.

Another threat to validity regards the toolchain that will be presented in the next section. To build the toolchain, different external tools are integrated, thus an error in one of those tools or in our integration could influence the result. To mitigate this, we only rely on tools that are known to be reliable. Terrier, from which we use their UTF tokenizer, is a well-known information retrieval framework. We also carefully tested our tokenizers to ensure of their behaviour. Java Parser is also used by more than 50 libraries and 100 projects on Github, and many companies use it and update it regularly. Nevertheless, as all tokenizers were carefully integrated (using their documentation) and tested, we do not consider this threat as important.

KYLM is widely used for comparing many recent n-gram approaches, *e.g.,* the work of Pickhardt *et al.,* [152]. Since this tool considered as relevant by the NLP community, we believe it is trustworthy. Of course we carefully analysed and tested it before using it. To further reduce these threats we will make our toolchain and data available once the paper is accepted.

A threat related to construct validity regards the way we built the defect dataset (used in answering RQ3). The dataset used for this research question is automatically generated using git commit messages and the JIRA Apache bug tracker. Thus, imprecise information or wrongly categorized issues in the tracker or misleading commit messages could generate noise in our data. However, given the strict guidelines used for the development and reporting of bugs in Apache Commons project, we believe that this could only be the case for a small percentage of the files. Therefore, the influence on the results would be relatively small.

11.7 Related Work on the Naturalness of Software

The application of machine learning to software engineering has received a growing interest in the recent years [25]. In particular, the study of software naturalness [77] has given birth to many approaches for generating source code ,*e.g.,* code completion [77], synthesis [155], review [76], obfuscation [115] and repair [154]) and performing static analyses [81, 104, 140].

According to Allamanis *et al.,* 's survey [25], n-grams are among the most popular language models. They have been used mainly for code completion [77, 136, 135, 157, 181], program analysis [81], bug detection [156], code review [76], and information extraction [165, 191]. Despite being popular we are unaware of any systematic and empirical evaluations that analyse the sensitivity of these models w.r.t. their parameters, although many papers give a few insights.

In their seminal work on code naturalness, Hindle *et al.,* [77] already inform us that Kneser-Ney smoothing gives good results for software corpora. However, they state that *"these are very early efforts in this area"*, which motivated our systematic evaluation of other smoothing techniques. According to their experiments, the reduction of cross entropy with higher-order n-grams saturates around 3- or 4-grams, whereas our evaluation shows that the reduction from 4-grams to 5-grams remains statistically significant in many cases. Finally, they tokenize code just like any English text and do not consider alternatives like AST-based tokenization.

In [136], Nguyen *et al.,* add semantic information, e.g., the data type of a variable, to lexical tokenization in order to improve code suggestion. Their approach inherently considers n-grams of multiple sizes; thus we do not know how a fixed n-gram size would affect their results. Also they used only additive smoothing, which is the simplest but arguably the less efficient technique [62, 63].

A subsequent work [135] tackle the problem of suggesting API calls. With this objective in mind, the authors argue that graph-based representations (e.g., AST and control flow graph) are more appropriate than n-grams computed from lexical tokenization. Based on such representations, they implemented API suggestion algorithms that outperform 8-gram models equipped with additive smoothing. These results motivate our interest towards AST tokenization, although we found that lexical (UTF) tokenizers are better than AST tokenizers at detecting bugs. Similarly, Hsiao *et al.,* [81] tokenize a program using a dependency graph (representing data-flows between the program statements). Their tokenizer hardly scale from 5-gram onwards, but even with smaller n-gram sizes it outperforms 7-gram models obtained from lexical tokenization. The authors do not mention the use of any smoothing technique.

Tu *et al.,* [181] propose a cache model that captures local regularities. Their evaluation shows that the best results (in terms of cross-entropy reduction) are obtained by combining the cache with standard n-gram models, as these capture different regularities. They also show that the size of the cache has a significant impact on cross entropy, and suggest that trigrams are sufficient. They do not explicitly mention what smoothing technique they use. As before, we argue that 4-grams and even 5-grams

can yield significant improvements in some cases, and that the choice of a smoothing technique is not without impact. On the other side we did not consider integrating a cache model, which is an interesting direction for future work.

Raychev *et al.,* [157] focus on suggesting API calls. The originality of their approach lies in that it combines n-grams with recurrent neural networks. Their experiments show a substantial improvement in effectiveness over standard n-grams, yielding 90% of relevant suggestions in top 3 candidates. They rely on trigrams and Witten-Bell smoothing, but did not study how these choices affect their results.

In a peripheral work, Hellendoorn *et al.,* [76] correlate the naturalness of pull requests in GitHub (computed by n-gram models) to their acceptance rate and the degree to which the requests are debated. They acknowledge the importance of choosing an appropriate size and smoothing technique, although they do not report on the sensitivity of their approach wrt. these parameters.

Sharma *et al.,* [165] propose an approach to identify tweets related to the software industry. More precisely, they use n-gram models to compute the cross entropy of tweets, and rank these accordingly. They evaluate the effectiveness of their approach with different n-gram sizes, and discover that 4-grams still offer an interesting marginal gain. As for smoothing, they assess only the Katz back off model [98] and do not consider the other alternatives.

Saraiva *et al.,* [159] perform a study on n-gram model specificities for source code, but focused on different research questions than the present paper. They first attempt to determine whether building language models specific to an application or specific to a developer can lead to better results. Then they investigate the importance of the temporality of language models. They found out that developer- and application-specific models were indeed performing better than general models, while temporality has little to no effect.

Finally, Yadid and Yahav [191] make use of n-gram models to correct and complete code fragments that were extracted from video tutorials. They use unigrams and bigrams conjointly, but have not investigated other parameter settings. Also, they do not mention the smoothing technique they use.

The above discussion highlights that n-grams is a frequently used statistical model. Many of the previous studies recognize the importance of the chosen parameters, but paradoxically evaluate only a few configurations. Moreover, none of the previous approaches considers alternative representations and their impact on the experimental-conclusion one can draw. Thus, our paper raises the awareness of what can go wrong and what should be tuned in order to draw reliable experimental conclusions.

11.8 Additional Tooling

To perform this study, a framework named TUNA (TUning Naturalness-based Analysis) that enable the performance of naturalness-based analyses of source code was developed. This framework is composed of 4 different modules, one for building the defect dataset, one for tokenizing source code, one for building N-Gram models and a last one for replicating the presented experiments. TUNA is the first open-source end-to-end toolchain to carry out source code analyses based on naturalness and is available at: `https://github.com/electricalwind/tuna`.

The defect dataset module is close to the Data7 framework as most of the code is shared. The module currently only works with Apache projects but can be extended at will. The major difference with the Data7 consists in the fact that information is first retrieved from the version history and then checked on the bug tracker, whereas Data7 first collect information from vulnerability reports.

The tokenizer module contains various tokenizers for source code. Currently, tokenizers for JAVA and C programming languages are implemented. For each language at least one tokenizer exists per types of representation presented in the chapter.

The N-Gram module allows to parameterize n-gram models, train them based on tokenized source code, and compute the cross-entropy of one or more source files. The module provides an implementation based on Kylm [131]. Following the interface segregation principle, alternative implementations can easily be added in the future.

11.9 Conclusion

Research on naturalness of code is focussing on assisting software engineering tasks using n-gram models. However, the use of such models requires setting a number of parameters. In this chapter, we perform a study and show that the choice of smoothing, tokenizer, unknown threshold and n values can impact the predicting ability of the models. We demonstrate that the Modified Kneser-Ney smoothing technique performs best, while n-values equal to 4 or 5 are generally appropriate. We also show that the closer, to human perspective (unprocessed code), the underlying representation is, the more robust the n-gram model is. This suggests that the most prominent choice of tokenizer, wrt. bug identification is the UTF one. Finally, we demonstrated with an experiment that researcher can come to wrong conclusions if they do not properly tune their models.

These results lead to the conclusion that in order to use N-Gram model to build VPM, only choosing one setting would result in a loss of information. Models built on different source of information learn different things and depending on the settings results may vary as well. Thus in the next chapter, several settings will be considered to build an approach.

12

A New VPM Approach Based on Naturalness

In chapter 9, a large experimental study of VPMs using FrameVPM was presented. In this chapter, we suggest to extend this study with the results obtained by two additional approaches, one based on naturalness only and one combining naturalness and Code Metrics. Indeed, previous research showed that defects and naturalness were linked, thus it is likely that vulnerability and naturalness are as well linked.
This chapter is a direct extension of chapter 9.

Contents

12.1 Introduction

Chapter 9 presented the largest experimental study of VPMs. This study highlighted some issues with existing approaches starting by the fact that models tend to overlook the fix of a vulnerability. This means that models output cannot be directly trusted but should be instead used as a guiding process. A possible explanation lies in the fact that the features the models are learning on are unlikely to really change after the fix. Another issue raised regarded "mislabelling noise", performances of existing approaches drastically drop when confronted to the issue.

In this context, the development of new approaches is necessary. Naturalness presented in the previous chapter seems like an interesting candidate to build an approach upon. Indeed as naturalness measure the degree of surprise of a language model when given a piece of text, an approach based on it should thus be more sensitive to changes in the text such as fixes. Moreover, previous studies [156] have shown a direct link between buggy code and naturalness which open the doors to studies on naturalness and vulnerabilities.

To build an approach, the choice of features is of upmost importance. In the case of a model based on naturalness, a feature corresponds to a specific setting of a Language Model. Chapter 11 taught us that models build on different code representation learn different things, thus an approach should consider more than one representation. Additionally different parameters of the language models such as n and unknown threshold should be considered. A last point that has not been raised in the previous chapter is on what the language model should learn. Indeed , like in the previous chapter one could consider training the language model on all other files of the release. The resulting naturalness would thus represent how the file fit in the current version of the project. On the other hand, one could also consider training on the previous release. The naturalness would thus give indication on the evolution of the file or if it was created, in the meantime, how the file fit in the project.

Thus, in this chapter, we extend the study of chapter 9 by considering 2 additional approaches, one build upon naturalness values from different settings and another one combining the naturalness approach and the Code Metrics one. Code Metrics and Naturalness approaches are both approaches with a predetermine number of features which make their combination quite natural. In particular this extension focus on how well both approaches performs in a standard environment and whether they handle the two aforementioned issues better than the original approaches.

In summary this chapter makes the following contributions:

- It presents an extension of Chapter 9 with two novel VPM approaches based on Naturalness.
- It reveals that models based solely on naturalness are not sustainable.
- It shows that models combining Code Metrics and Naturalness are in fact performing better than models based on Bag of Words in realistic setting (with mislabelling).

12.2 Research Questions

In this extension, we evaluate two additional approaches, the emphasis is thus put on the comparison in performances. Thus some research questions from the initial study don't need to be reinvestigated, starting with the first one which regards the distribution of class and severity in the dataset. However the second and third research questions are of interest in this extension. Thus we reformulate them as:

RQ1: *How well the newly introduced approaches identify vulnerable components between software releases compared to existing ones?*

RQ2: *Do the new approaches identify severe vulnerabilities?*

The forth research question from the initial study was investigating the possibility to tune those models to find more severe vulnerabilities. The results being positive, this question is not further develop in this extension.

The fifth original research question is of particular interest in our case. Indeed this question was investigating the robustness of the approach to the presence vulnerable components that were fixed at the evaluation time. Naturalness being by design more sensitive to change in text should provide better result in this situation than existing approaches:

RQ3: *Are Naturalness-based-approaches less sensitive to the issue of vulnerability fixes?*

Finally, the last initial research question regarded the creation of models based on incomplete or mislabeled data [174], which in the case of vulnerability prediction occurs when vulnerabilities that have not been yet reported at the training time are part of the training data as non-vulnerable components. Approach previously investigated were particularly sensitive to this with important drop in performances. As this is an important problem often overlooked, we ask:

RQ4: *Are Naturalness-based-approaches less sensitive to vulnerabilities unreported at the time of training?*

12.3 Naturalness Based Approaches

In this section, the design and setting chose to build the naturalness approach are detailed. The first part focus on the different elements requires to obtain a representative set of features, while the second present how these features are transformed into approach.

12.3.1 Feature Extraction

To build a VPM based on naturalness values, it is necessary to obtain different values of naturalness that will be used as features. Ideally each value should be independent and bring new information that would help build a better model. Fortunately, results of the previous chapter indicates that N-Gram Models built on different tokenizers learn different information. Thus by using different tokenizers, we can obtain several naturalness-value accounting for different information. Yet the different tokenizers used for the previous study were designed solely for Java programs and all projects from the Data7 framework are coded in C. Thus, additional tokenizers were added in Tuna to handle C projects. Hence, for this study we rely on 3 different tokenizer each one accounting for one the code representation presented in the previous chapter, *i.e.,* UTF tokenizer, Lemme Tokenizer and AST Tokenizer. While the UTF tokenizer is the same as the Java one, the Lemme Tokenizer is based on the C grammar of ANTLR and the AST one used the output of the Joern tool.

Tokenizers are not the only parameter that can be modified to obtain naturalness values with different meanings. Indeed, as naturalness is a measure of the degree of surprise a language model trained on a given corpus when faced with a piece of text, its essential part lies in its training corpus. The same value of naturalness given by models trained on different corpus cannot be interpreted similarly as if the same training corpus were used. In the previous chapter, the choice was done to compute the relative naturalness, *i.e.,* each file was evaluated against a model trained on all the other files of the project. This setting provide interesting result as it gives a notion to how close a file is from the rest of the project. Still this requires to create one model per file to evaluate, which can become problematic in the case of large projects such as the Linux Kernel with more than 20,000 files to evaluate per release. Another setting commonly used by previous studies [156, 77] consists in training on the previous version. This offers the advantage that only one model needs to be created per release to analyse instead of one per file. Yet it has the inconvenient that a file that has not been modified in between will be present in both training and evaluation which could bias the analysis. Moreover, in this setting it is not possible to compute the naturalness of the first release as there is no training corpus available. Both solutions having their up and downside, we integrate both of them for each tokenizer. Those two solutions are integrated in FrameVPM as OtherFilesNaturalness and PreviousReleaseNaturalness.

Finally, previous chapter shown that some Language Models parameters could influence the output such as the n and the unknown threshold. To be as complete as possible we investigate the use of two different n and thresholds, 4 and 6, *resp.* 1 and 4. Regarding the smoothing technique as the choice of Modified Kneser Ney was undoubtedly the best, we decide to only use this one.

This results in a set of 24 naturalness based features on which we can build an approach.

12.3.2 Naturalness VPMs

Once all features computed, it is possible to build VPMs. In this extension, we investigate two types of models using those features. The first one named "Pure Naturalness" solely rely on those features. The results of this model should give indicators as of the potential of naturalness as indicators of vulnerability. Yet, naturalness might not be enough on its own and a model might require additional data to better perform. Among the previously introduced approaches, only the Code Metrics one can ideally be combined with a Naturalness one. Indeed, both approaches rely on a predetermine number of features which guarantee that they will all be considered by the model, while in the case of Imports and Function calls and Bag of Words the naturalness features might be "lost" among all the other features, thus making the analysis of the benefit of naturalness impossible. Thus the second type of models investigate combine features from the Code Metric and Naturalness approach.

12.4 Results

12.4.1 RQ1: Performance of Naturalness Based Approach

Figure 12.1, similarly to fig. 9.3, records the distribution of the predictions' evaluation metrics (MCC, Precision and Recall) for all methods and subjects we consider. Interestingly, the Pure Naturalness approach doesn't perform well at all in terms of recall and MCC with average values below 0.2 for the three projects. The results in terms of precision are slightly better but stay behind the other approaches. Regarding the combination of Naturalness and Code Metrics, we observe a slight drop in recall and MCC compared to the Code Metrics approach in most of the cases, but interestingly this is offset by an increase in precision. This suggests that Naturalness on its own is not enough to build a VPM but could be used to tune the precision of a model at the cost of the recall.

Interestingly, the results are consistent for all the studied subjects, with the *Bag of Words* and *Function Calls* methods achieving the best results, i.e., yield the highest MCC. Code Metrics is the most precise method, but overall this advantage is small. With respect to recall, *Bag Of words* clearly outperforms the other methods.

Regarding the statistical analysis on our results presented in the lower triangular of Table 12.1, we can observe that all the p-values are significant except regarding imports and the combination of Naturalness and Code Metrics and all $\hat{A}_{12}$ in the last two lines are lower than 0.5 indicating that they are not performing better than the other approach. Additionally if the Combination of Code Metrics and Naturalness is performing reasonably worse than the other approach, the Pure Naturalness approach if not performing at all with $\hat{A}_{12}$ of 0 in every case.

Overall, statistical analysis confirms the box plot observations, the two suggested approaches perform worse than the existing ones.

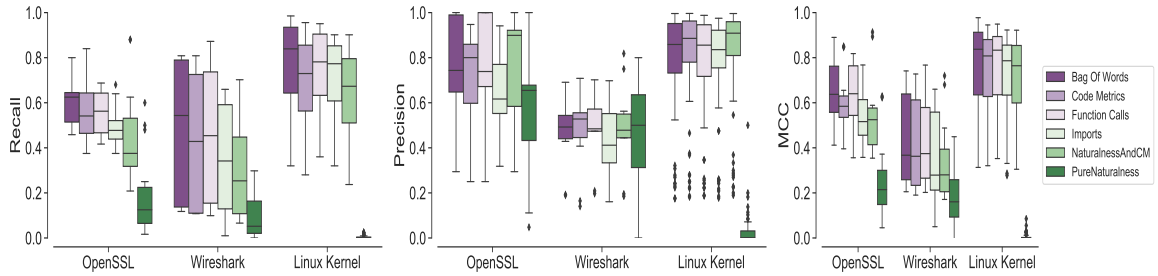


Figure 12.1: Performance metrics for all vulnerable components (RQ1).

Table 12.1: RQs 1-2

This table comprises two separate triangular sets of results. The lower triangle presents results comparing approaches based on all vulnerable components, while the upper triangle shows results comparing approaches based on severe vulnerable components. Each cell contains a p-value and a A_{12} (line, column) effect size measurement.

	Bag Of Words	Code Metrics	Function Calls	Imports	Naturalness and CM	Pure Naturalness
Bag Of Words	-	1.10E-03/0.53	1.38E-04/0.51	1.27E-04/0.53	5.98E-04/0.53	9.2E-26/0.89
Code Metrics	1.34E-13/0.44	-	0.9405/0.49	2.23E-02/0.50	1.49E-01/0.51	6.25E-26/0.91
Function Calls	5.72E-10/0.47	5.87E-05/0.54	-	0.1796/0.51	6.73E-01/0.52	1.45E-25/0.91
Imports	1.78E-19/0.41	1.80E-04/0.46	5.74E-15/0.43	-	4.50E-01/0.50	1.25E-25/0.90
Naturalness and CM	6.90E-20/0.39	6.34E-15/0.44	6.56E-14/0.41	3.9E-1/0.48	-	1.55E-25/0.88
Pure Naturalness	4.9E-26/0	4.9E-26/0	4.9E-26/0.0	7.76E-26/0.0	4.9E-26/0.0	-

12.4.2 RQ2: Performance of Severe Vulnerability Prediction

Figure 12.2 presents the results related to severe vulnerabilities. Similarly to what was observed in previous chapter, the performances of the two new instance decrease. Yet, this time the precision of the combined approach is not better than of the original approach. In fact it performs worse for all measures and project, with the exception of the Linux Kernel where the results are somehow similar.

The upper cells of Table 12.1 indicate that none of the result of the combined approach are statistically significant except with Bag of Word which makes impossible the comparison with other approaches. Overall it seems that for severe vulnerability the combined approach provides similar result than the existing one. Regarding the pure naturalness approach, result is again disappointing with really low A_{12} value, which confirms that naturalness cannot be used alone.

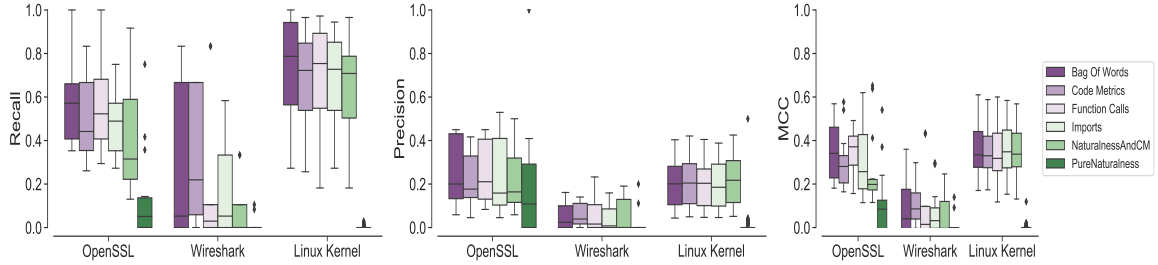


Figure 12.2: Performance metrics for severe vulnerable component (RQ2).

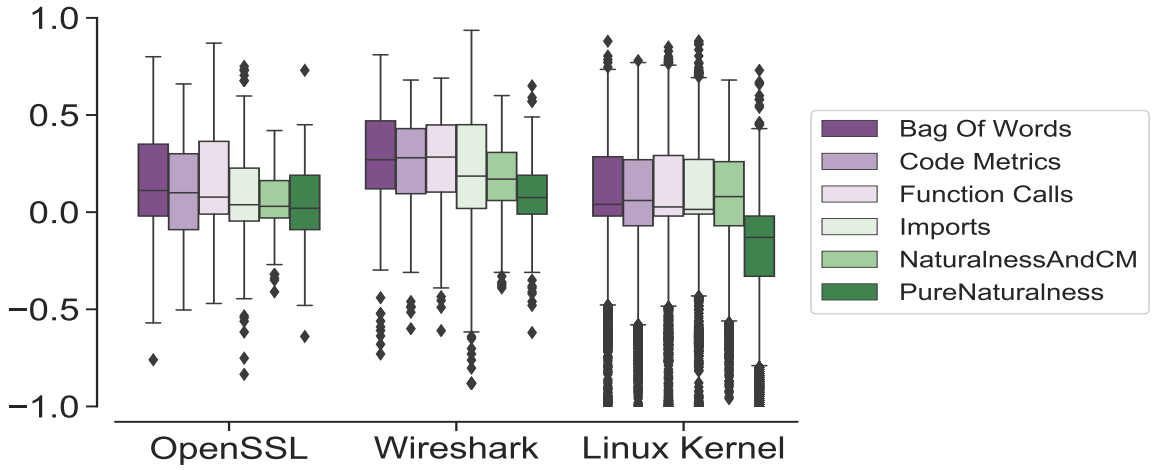


Figure 12.3: Vulnerability probability before and after vulnerability fix (RQ3)

12.4.3 RQ3: Sensitivity to Vulnerability Fixes

Figure 12.3 shows the difference in probability of a component to be vulnerable before and after a vulnerability fix. Ideally, values should be negative since fixed components should be less likely to be vulnerable. If existing approaches fail to show such trend, the two suggested approaches provide better result with a lower difference but remain most of the time positive. While the Pure Naturalness shows the lowest difference but results from previous research questions demonstrated that it cannot be used alone, the results of the combined approach is encouraging as a drop is still observed which means that the addition of naturalness features can reduce the sensitivity to the issue of fixes.

12.4.4 RQ4: Sensitivity to Mislabeling Noise

This RQ investigates the predictions that one can achieve by using the reported vulnerabilities at the software release time. This means that the training happens on the reported, at the release time, vulnerabilities. In the original study, only the results of the Bag of Word were presented as it was shown as the best performing approach by previous research questions.

Figure 12.4, 12.5, 12.6 shows the results obtained when comparing the result of Bag of Words, Code Metrics and Combined approaches. As a remainder 'experimental' results are those obtained with the knowledge of all vulnerabilities (all vulnerable components are labelled as vulnerable and all non-vulnerable are labelled as non-vulnerable). 'realistic' results are those obtained with the restrictive knowledge of vulnerabilities (labelled vulnerable components are those that have been reported at the release time, while the rest are labelled as non-vulnerable). Finally, 'noiseless' results are those obtained when removing the vulnerable components from the training set (labelled vulnerable components are those that have been reported at the release time, vulnerable components related to unreported vulnerabilities are removed and the rest are labelled as non-vulnerable).

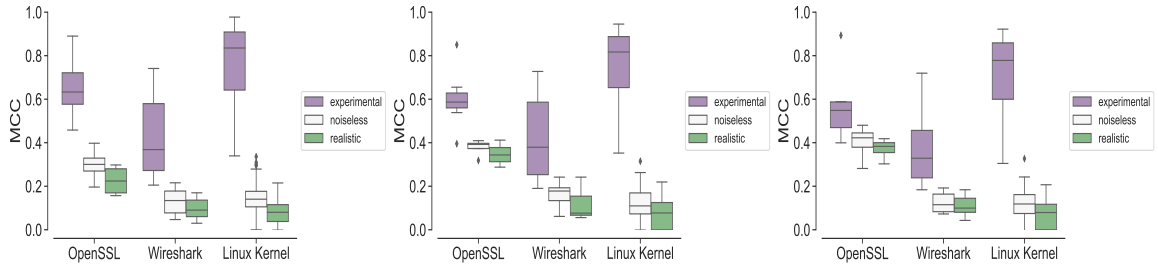


Figure 12.4: MCC of the approaches with subsets and noisy Data (RQ4).

Bag Of Words (left), Code Metrics (center) and Combined (right)

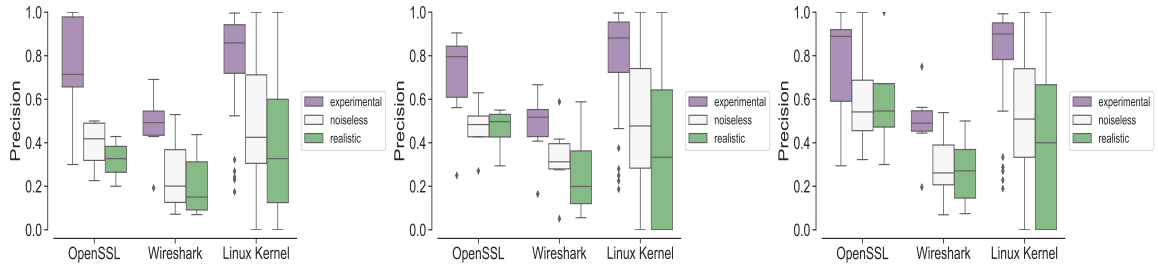


Figure 12.5: Precision of the approaches with subsets and noisy Data (RQ4).

Bag Of Words (left), Code Metrics (center) and Combined (right)

Interestingly the figures show that in terms of Precision and MCC models using Code Metrics are performing better in the noiseless and realistic setting than the ones using Bag of Words, especially for OpenSSL. This could mean that models based on Code Metrics generalize better and are less impacted by the mislabeling and unbalance, whereas Bag of words model will be more specific and will perform better on a full dataset. Regarding the combined approach an even more interesting phenomenon is observed, the drop between noiseless and mislabeled data is reduced in all cases compared to the Code Metrics approach. This drop is corresponding to the cost of mislabeling data, thus its reduction means that the addition of naturalness can counter some of the effect of mislabeling. Overall, even though the results are quite poor the combined approach is the one performing best in the realistic setting, which was unexpected given the result of the first research questions.

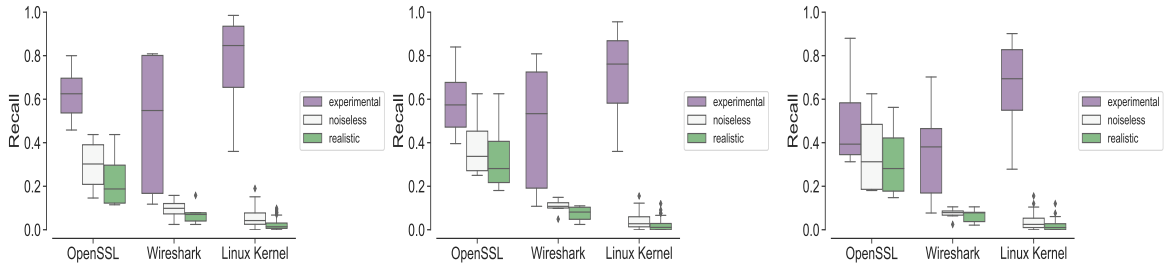


Figure 12.6: Recall of the approaches with subsets and noisy Data (RQ4).

Bag Of Words (left), Code Metrics (center) and Combined (right)

12.5 Conclusion

In this chapter, we presented an extension of the study presented in chapter 9. This extension introduced two new VPM approaches one based solely on naturalness values as features and one combining code metrics and naturalness. While the first one gave overall disappointing results, the latter offered some interesting findings. Hence, it turns out that the use of naturalness can reduce the sensitivity of existing models to issues such as already fixes vulnerabilities and mislabeling noise. This indicates that naturalness features have the potential to improve other approaches but cannot be used in a standalone setting. Results also showed that while Bag of words undoubtedly remains the best approach in an ideal setting, in a realistic a combination of Code Metrics alone or even better combine with Naturalness can perform better.

13

A Negative Result, the Naturalness of Mutants

Previous chapter presented studies exploiting the naturalness of software for software engineering tasks such as bug detection and vulnerability prediction. The state of the art is filled with various applications of the naturalness, but interestingly, there is little to few studies applying it to the context of software testing. In this chapter, we address our fourth challenge by considering the naturalness of software as a way to improve the selection of "fault revealing" mutants. We seek to identify how well artificial faults simulate real ones and ultimately understand how natural the artificial faults can be. The intuition behind this suggestion is that natural mutants, i.e., mutants that are predictable and are semantically useful and generally valuable. It is as well expected that mutants located on unnatural code locations will be of higher value than those located on natural code locations. Based on this idea, this chapter proposes mutant selection strategies that rank mutants according to a) their naturalness b) the naturalness of their locations and c) their impact on the naturalness of the code that they apply to. An empirical evaluation of these issues is performed on a benchmark set of 5 open-source projects. Overall the findings are negative but they are of interest as they confute a strong intuition, i.e., fault revelation is independent of the mutants naturalness.
This chapter is based on work that has been published in the following paper:

- *Are mutants really natural? A study on how "naturalness" helps mutant selection (ESEM'18)*
  M Jimenez, T Checkam, M Cordy, M Papadakis, M Kintis, Y Le Traon, M Harman

Contents

13.1 Introduction

Empirical and experimental evaluations of software testing are typically performed by using artificial faults. These faults are seeded in selected programs and are used as the objectives for comparing techniques. Thus, the techniques and test cases are assessed by measuring their ability to detect these types of faults.

This type of assessment is known as fault seeding or mutation testing. Fault seeding is performed by altering the syntax of the programs. Thus, researchers transform (mutate) the syntax of the programs with the aim of generating program versions (mutants) that are semantically different. By demonstrating (revealing) the semantic differences between the mutants and the original program, one can effectively measure test effectiveness [46, 29, 148].

Evidently, as the mutant faults are generated by altering the programs' syntax, they alter the program semantics. However, how semantically useful are such artificial faults? In practice, most of the mutants tend to have a major effect on the program semantics, which makes them non-useful to testers (since they are trivial and can be revealed by too many tests). Whereas testers need mutants with a small effect on the program semantics as these are hard to reveal and result in strong test cases [139, 145]. Nonetheless, the key question is how well mutants (which are in a sense artificial faults) mimic real code and real faults?

To this end, recent research has indeed shown that some (very few) mutants are realistic [29, 148]. However, since the number of realistic mutants is very small, compared to the total number of mutants [28, 148], these have almost no practical effect [46, 145]. In other words, mutation introduces a very large number of non-interesting (bad) mutants and very few interesting (good) ones. This raise the question of how to select mutants that are semantically useful and natural, e.g., simulate well real code and faults.

To identify semantically useful mutants, we need a model capable of capturing the goodness of mutants. Previous research has focused on identifying the types of mutants that are the most important ones [107, 147]. However, these techniques have little or no success as they fail to outperform the random mutant selection [107, 45]. One potential explanation could be that it is the location of the mutants that make mutants good and not their type. Another potential explanation could be that good mutants are the result of the combination of the location with the mutant type.

Nevertheless, we need a model capable of identifying the interesting program locations, interesting mutant types and interesting pairs of location and types. Thus, in this chapter, we investigate the use of N-Grams models, as an approximation mechanism for capturing the program semantics and select mutants. Ideally N-Grams models should be able to exploit the implicit rules, coding conventions and general repetitiveness of source code and categorize mutants (and their locations) as "natural", *i.e.,* mutated code that is likely to appear in a code base (follow the implicit coding norms of developers), and "unnatural", *i.e.,* mutated code that is unlikely to appear in the code base.

Naturally, the notion of "naturalness" raises the question of how natural or unnatural mutants are, since they are simulated faults. This intriguing question motivated the study and desire to understand the properties and connections between program syntax and semantics from a testing (fault revelation) perspective.

The intuition is that natural mutants (considered as probable by such models) are more valuable than the unnatural ones because they follow the implicit norms and the way programmers code. Mutants located on unnatural code locations, which previous research linked with error proneness [156], are thus expected to be of higher value than those located on natural code locations.

In essence, the question regards the likelihood for developers to do things wrong. Natural code fragments are easier for developers to compose and more probable of being semantically right (since they are highly repetitive) than unnatural code fragments [156]. Thus, we expect that mutants making a code fragment more natural, while at the same time being semantically different from the original version, to have more utility than mutants making a code fragment less natural. This is because such mutants are likely to introduce expected semantic deviations, which have small effect on the program semantics. Furthermore, such mutants are worth investigating since they form likely alternatives to the original code.

To investigate this hypothesis, we consider a set of real bugs from 5 Java open source projects. We measure the naturalness at both the file level of granularity (used to compute the naturalness of mutated files, *i.e.,* Java classes) and at the statement level (used to compute the naturalness of the original and mutated code statements).

We use the naturalness measurements to rank the mutants according to: a) the naturalness of mutated code files, b) the naturalness of the original code statements and c) the impact on the naturalness of the mutated statement(s) (difference on the naturalness of the original and mutated code). We evaluate these ranks wrt. their probability to be killed by test cases that reveal real faults. Thus, we assess whether mutants ranked higher are indeed preferable than those ranked lower (i.e. their killing implies the revelation of real faults).

In summary this chapter makes the following contributions:

- It presents the negative results of an empirical study on the use of naturalness for mutant selection.
- It confutes the afore presented intuition and thus increase the understanding of the interconnections of program syntax, program semantics and software faults.
- It shows that the fault revealing utility of mutants is independent of their naturalness

13.2 Background on Mutation Testing

Mutation is a well-studied technique with increasing popularity among researchers and practitioners alike, as it is evident from the most recent survey in the area [147]. Mutation works by inserting artificial faults into the program under test, termed the *original program*, thus, creating many different versions of it, each one containing a single syntactic change. These versions are called *mutants*. Mutants are used to evaluate test cases based on their ability to distinguish the mutants behaviour from that of the original program. If such a test case exists (or can be created) for a particular mutant, then we term the mutant *killed* (or *killable*). A mutant is termed"fault-revealing" with respect to a particular fault if the test cases that kill it are a subset of the test cases that can also reveal that fault, *i.e.,* lead the program under test to an observable failure.

Not all mutants can be killed by test cases. In such a case, we say that the mutants remain *live* and we need to investigate why this happened. A mutant can remain live after its execution with test cases for two reasons, either the test cases are not "strong" enough to exhibit the behavioural differences between the mutant and the original program, thus, indicating a weakness of our test suite, or the mutant is an *equivalent* one. Equivalent mutants are syntactically different versions of the original program but semantically equivalent, meaning that their behaviour is the same to the original program for every the possible inputs.

Mutation systematically introduces syntactic changes to the original program. These changes are based on specific, predefined rules called *mutations* or *mutant operators*. Such operators can replace relational operators with each other, replacing $>$ with $<$, for example, or increase the values of variables by inserting appropriate arithmetic operators to variable usages. Research has shown that the choice of mutation operators and their implementation affects the effectiveness of mutation and its tools [29, 101], thus, it is important to carefully select the mutants and the tools that one uses when applying mutation.

In mutation testing the identification of "valuable" mutants is a known open issue [148]. Previous research has shown that the majority of the mutants is redundant and this can induce severe problems in the mutation test assessment process [146, 107]. This means that not all the mutants are of equal value. Indeed, some few mutants are useful, while the rest (majority) are easy-to-kill, are duplicates of other mutants [100], or are redundant wrt. to the useful ones. This begs the question: *How can we distinguish the valuable mutants before analysing them?*, or equally, *Do valuable mutants have specific properties that can distinguish them from the less valuable ones?*. In this chapter, we attempt to use the naturalness of software to answer these questions.

13.3 Research Questions

We start our investigation by checking whether mutants alter the naturalness of a project and to which direction (make the code more natural or unnatural). This poses:

RQ1: What is the impact of mutants on the naturalness of code?

We answer this question by checking the differences in the naturalness of the original and mutant program files. We also check the number of mutants having the same naturalness values. The answer to this question ensures that we can leverage natural language models in mutation testing. Given that we found evidence that mutants have different naturalness values, we turn to design naturalness-based mutant selection strategies. We thus, investigate the fault revelation ability of the mutants that can be categorized as natural and unnatural. Hence:

RQ2: Is "natural" mutant selection stronger than the "unnatural" mutant selection?

To answer RQ2, we need to know the probability of revealing a fault when killing a mutant, for every mutant in our set. We therefore repeatedly applied mutation testing on our benchmark sets and compute the fault revelation of both natural and unnatural mutant sets (of different sizes). We report on the differences in fault revelation of three strategies based on: (1) the naturalness of mutated code fragment (2) the naturalness of the original code fragments and (3) the impact of mutants on the naturalness of the code (entropy difference between the original and mutated code fragments). This information is useful for designing effective naturalness-based mutant selection strategies.

After experimenting with the different naturalness-based mutation testing strategies, we evaluate them with respect to other methods. Previous research has shown that the most effective mutant selection strategy is the random mutant selection one [107, 145]. Thus, our next RQ is:

RQ3: How does naturalness-based mutant selection compares with random selection?

To demonstrate whether there are benefits related to the naturalness-based mutation testing, we repeatedly compute the fault revealing probabilities of the compared approaches and determined their fault revelation probabilities.

Table 13.1: Java Subjects Details

Test Subject	Description	LoC	#Faults Used
JFreeChart	A chart library	79,949	19
Closure	Closure compiler	91,168	92
Commons Lang	Java utilities library	45,639	30
Commons Math	Mathematics library	22,746	74
Joda-Time	A date and time library	79,227	15
Total	-	**318,729**	**114**

13.4 Methodology

13.4.1 Test Subjects: Real Faults, Mutants and Test Suites

For the experiment, 5 real-world projects and 230 bugs from the Defects4J database [96] are used. Defects4J includes a reproducible set of real faults mined from source code repositories, along with scripts that facilitate the conduction of experiments on these faults. In total, we consider 357 real-world faults accompanying the test subjects. This last point justifies the use of Defect4J instead of the two defect datasets introduced in this dissertation, *i.e.,* in chapter 4 and 11, as they are not storing the tests corresponding to the fault which is required to validate mutants.

The first four columns of Table 13.1 present details about the test subjects: their names, a small description, the source code lines (reported by the `cloc` tool [4]) and the number of faults available.

To compose test pools with a large number of tests, data from the study of Papadakis *et al.,* [148] are used. It involves two state-of-the-art test generation tools (EvoSuite [60] and Randoop [142]). The test pools are composed of the available developer test suites and 20 test suites, 15 from EvoSuite and 5 from Randoop. In total the test pools are composed of 1,375,341 automatically generated tests and 58,131 tests from the project developers.

For the creation of mutants, Major [97] is used. Major implements the main mutation operators *i.e.,* the Arithmetic (AOR), Logical Connector Replacement (LCR), Relational (ROR), Bitwise (BTW), Shift (SFT), Unary Operator Insertion (UOI) and Statement Deletion (SDL). This tool is robust, easy to use and has been used to study the relationship between faults and mutants. It also operates at the source code level, which is necessary for calculating the naturalness of the source code. In the experiments, we apply the tools on the fixed program versions of the datasets using all the supported operators.

13.4.2 N-Gram Model Building

To compute the naturalness of the mutant, we rely on the Tuna framework presented in chapter 11.

Two tokenization schemes are considered, one at the file level of granularity and one at the statement level. The *file-level tokenized content* is the result of tokenizing the code files, *i.e.,* Java classes, as a whole. The *line-level tokenized content* is the result of separating the tokens according to the statements they belong in the code files. It is noted that the comments are discarded.

Regarding the settings for the N-Gram Model, we pick according to the result of chapter 11 the following parameters: n equals to 4 with *Modified Kneser Ney* as smoothing technique and set unknown threshold to one.

Overall, to compute the naturalness of all mutants, we proceed as follows (for every mutated file):

1. Collect and tokenize all source code files of the projects containing the mutated file under evaluation
2. Exclude the file that has been mutated
3. Use the resulting set of source files as the training corpus to build *two N-Gram models*, one at the file-level and another one at the line-level
4. For the mutant files (test corpus):
    - Tokenize the mutant.
    - Compute its cross entropy as well as the original one using the *file-level model*.
    - Do a `diff` between the line level tokenized versions of the original and the mutant.
    - Measure the cross entropy (naturalness) of the deleted lines and added lines using the line level model and attribute it to the original and mutant, respectively.

13.4.3 Evaluation Process

To answer the stated RQs, we applied mutation testing on the project files where the faults appeared. We then measured the naturalness of all the original and mutant files using the process described above. Since the models are measuring the naturalness of code fragments based on the training corpus, we need to separate the training from the evaluation corpus in order to avoid biasing the ability of the model.

Thus, for each fault, we train our models on all the project files excluding the faulty ones, which is the default strategy adopted in Chapter 11 and the strategy known as Other file Naturalness in Chapter 12. This establishes a clear separation of training and evaluation targets as it ensures that the same files do not belong to both training and evaluation.

To answer RQ1 and show that the syntactic differences of mutants can be scored by language models (ranked according to their naturalness), we collect all mutants and categorize them according to the entropy of the original files (we record the entropy differences of the original and mutant files). We thus seek to identify trends regarding the syntactic transformations introduced by the mutants *e.g.,* whether mutants make natural files more or less natural.

To answer RQ2 and demonstrate the ability of naturalness to assist mutation testing, we rank the mutants according to their naturalness. We investigate three scenarios: the naturalness of the a) mutant location, b) mutated file and c) absolute difference of the original and mutant files. To check whether natural or unnatural cases are interesting, we rank the mutants in an increasing and decreasing order (of entropy) and contrast their fault revelation abilities. To determine fault revelation we repeatedly apply mutation testing by selecting the x% of the top rank mutants (we consider sets of 0, 5%, 10%, 15% to 100%) and compute the fault revelation probabilities of these sets. To account for coincidental and other random factors, we applied our process 1,000 times for every considered set of mutants.

The fault revelation probabilities of the selected mutant sets were computed by measuring the ratio of the times that the faults were revealed by the test suites that kill all the considered mutants. The test suites that kill the candidate mutant sets were selected based on the following procedure: We start from empty test sets and stop when we kill all the mutants. At each step we select the next mutant in the list and randomly pick a test that kills it (selected from the pool of the available test suites). To avoid composing test suites with large redundancies we remove all the mutants that are killed by every test we select. In case no test kills a targeted mutant, we discard the mutant. This process mimics what a tester does when she uses mutation testing [29] and ensures that the selected tests are relevant to the mutants we study. Overall, this is a typical evaluation process that has been followed by many studies [111, 29, 146].

To answer RQ3 and compare with the random mutant selection we repeat the process followed in RQ2 for random mutant orderings. To cater for the stochastic nature of the random orderings we repeat this process 100 times and compare our results with the naturalness-based orderings.

13.5 Results

13.5.1 RQ1: Impact of Mutants on Naturalness

The first research question checks whether mutants change the cross entropy of the code under analysis. Thus, we check the ability of the N-Gram models at identifying the syntactic changes introduced by mutants. To do so, we compare the cross entropy of the mutated (and original) code files.

Table 13.2 records the distribution of entropy values of the mutants. We can observe that entropy can distinguish the great majority of the mutants. Figure 13.1 presents the entropy differences between the mutant and the original files. The boxplot presents the values resulting by subtracting the entropy of the mutated file to the one of the original. Thus, positive values indicate that mutants are less natural than the original, while negative values the opposite.

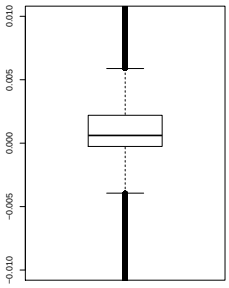


Figure 13.1: Cross Entropy difference between mutant and original files

We observe that mutants alter the cross entropy of the code with the majority of them making the code less natural.

Table 13.2: Distribution of entropy values

Number of mutants with equal entropy values and their frequency. For instance, 83,707 mutants have unique entropy values, while 10,347 mutants have entropy value equal to another mutant.

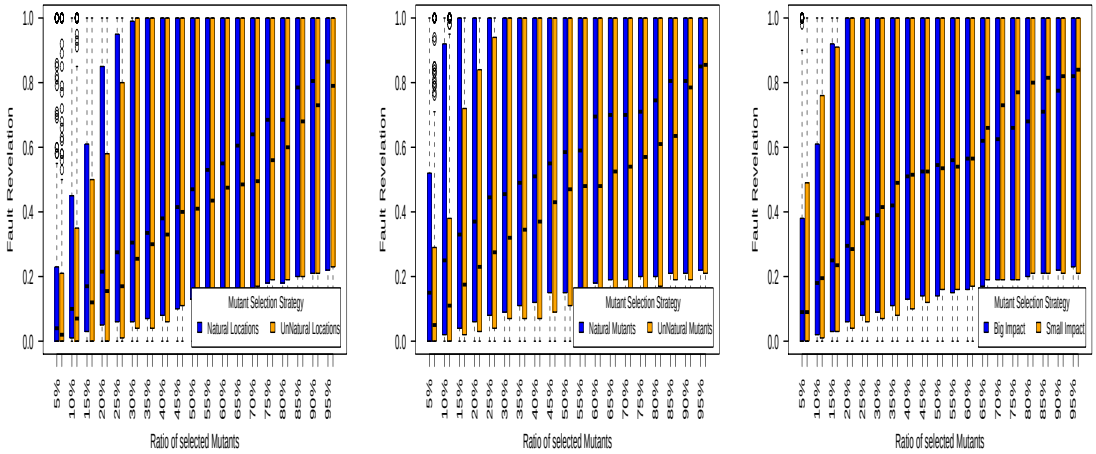
No of mutants	1	2	3	4	5	6	7	8	¿=9
Frequency	83,707	10,347	2,506	1,203	550	361	253	176	511

We observe that models can indeed capture the syntactic differences between the original and the mutant files. Interestingly, we observe that mutants sometimes make the code more natural and sometimes less natural. The tendency is to make the code less natural as the majority of the mutant files have higher entropy than the original files. In our results, only 30% of the mutant are more natural than the original.

Overall, the results suggest that mutants change the cross entropy of the code under analysis and they make it sometimes more and sometimes less natural. This leads us to the question of whether there is a link between the value of mutants and their naturalness.

13.5.2 RQ2: "Natural" Vs "Unnatural" Mutant Selection

This RQ regards the design of naturalness-based mutant selection strategies. To this end, we need to evaluate whether natural or unnatural mutants are preferable. We thus, collect all the mutants from our subjects and measure their entropy, using both the file-level and statement-level tokenizers. We then collect a) the entropy of the code fragments where mutants are applied (using, the statement-level tokenizer) b) the entropy of the mutant files (using, the file-level tokenizer), and c) the absolute difference in entropy between the original and mutated files (using, the file-level tokenizer) and form naturalness-based mutation testing strategies (by ranking mutants in an increasing and decreasing order).



(a) Natural VS Unnatural code locations (b) Natural VS Unnatural mutants (c) Big VS Small impact (entropy differences introduced by mutants)

Figure 13.2: Identifying fault revealing mutants

Natural VS Unnatural program locations, Natural VS Unnatural mutant files and Big VS Small mutants' impact. The x-axis records ratios of the top ranked selected mutants, while the y-axis records the fault revelation ability of the two selected strategies, i.e., fault revelation of natural and unnatural mutants. Higher values indicate higher fault revelation.

These three entropy measurements help us investigate which ones of the strategies we can compose leads to interesting mutant sets. We investigate these particular cases because they involve interesting properties: a) regards the locations that should be mutated, b) regards the natural/unnatural order of mutants, while c) regards the 'extreme' mutants, *i.e.,* choosing mutants that impact the entropy measure too much, either by making the code much more natural or much more unnatural.

We apply naturalness-based mutation testing by ranking mutants in an increasing and decreasing entropy order, *i.e.,* we follow the procedure explained in the previous section, and obtain the fault revelation ability of our mutant sets. Here we compare the increasing and decreasing entropy orders in order to identify the strategy that leads to the most promising results. We use the same number of mutants in every comparison to establish a fair comparison. In the following subsections we discuss the results related to the cases a), b) and c). All our results are presented by computing the differences in the fault revelation values of the increasing and decreasing order strategies. Thus, by observing positive values we can conclude that increasing strategies are preferable over the decreasing ones.

Mutant Locations Figure 13.2a depicts the results related to the fault revelation ability of the mutants located on natural and unnatural code locations for several ratios of selected mutants. Higher values indicate that natural locations are preferable.

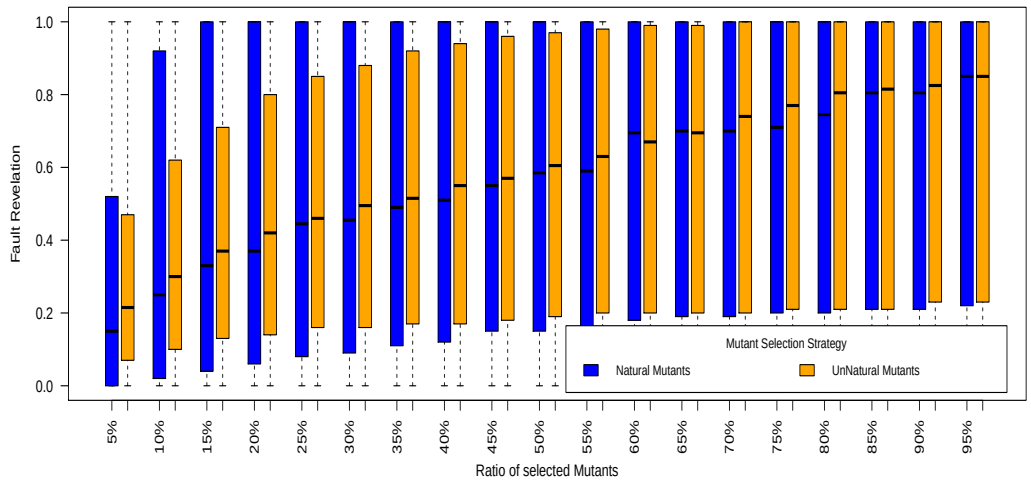


Figure 13.3: Fault Revelation of naturalness and random mutant selection

The x-axis records ratios of selected mutants and the y-axis records the fault revelation for every fault considered.

We observe a small difference in favour of the natural locations over the unnatural ones. To validate this, we performed a Wilcoxon signed-rank test and found no statistical differences (at the $a < 0.05$ significance level). These results suggest that the naturalness of the code locations is not discriminative of the fault revelation ability of the mutants. In other words, ranking mutants according to the naturalness of their location is not really helpful (is not a good feature of the semantic usefulness of the mutants).

Mutant Files Figure 13.2b records the results regarding the fault revelation ability of the natural and unnatural mutants. Natural mutants are those having mutant files (whole files) with low entropy. In this case the results show a tendency towards the natural mutants but the difference is small. By performing a Wilcoxon signed-rank test we find statistical differences (at the $a < 0.05$ significance level) when selecting in the range 5% to 25%. When selecting more than 25% of the mutants the differences are not significant. When measuring the effect size of the differences, using the Vargha and Delaney $\hat{A}_{12}$ [182], we get values of approximately 0.57 to 0.58 (meaning that natural mutants are preferable in 57-58% of the cases). Interestingly the fault detections of both natural and unnatural mutants are much higher than those of the natural or unnatural locations indicating a potential of such strategies.

Mutants Impact Figure 13.2c records the results regarding the strategies with extreme impact, *i.e.,* abs(original entropy - mutant entropy) (impact on the entropy of the file). The underlying idea is that the 'extreme' mutants (mutants with the largest impact) are more interesting than non-extreme ones. Unfortunately, the results indicate that this choice does not matter much (since almost all such values are close to each other. The differences are not statistically significant indicating that the impact on the naturalness is not a discriminative factor that we could use.

13.5.3 RQ3: Naturalness Mutant Selection VS Random

This RQ regards the comparison of naturalness-based mutation testing with a baseline in order to see if it is of any practical value. To investigate this, we select the two best performing strategy, *i.e.,* selecting the most natural mutants, and compare them with the random selection. As we discussed earlier, random mutant selection forms a tough baseline and thus, by demonstrating that the naturalness outperforms the random selection, we effectively establish an approach capable of discriminating between good and bad mutants.

Figure 13.3 summarizes the results of the comparison. The boxes record the fault revelation ability of the mutant sets that are composed of (0-100%, in steps of 5%) of the considered mutants. As can be seen the naturalness-based strategy performs similarly to the random mutant selection. By performing a Wilcoxon test, we find that the results are not statistically significant. This means that the differences are marginal and we cannot expect any important benefit.

13.6 Discussion

13.6.1 Visualizing Naturalness and Fault Revelation

To further investigate the relation between naturalness and fault revelation, we visualize the data (with the hope to see some general trends that might not be captured by the analysis). Figure 13.4 plots the naturalness (naturalness of the original file minus the naturalness of the mutant file) and fault revelation probabilities for every mutant we consider. In the figure we observe that there is no exploitable pattern.

Mutants with high fault revelation (points on the x-axis with values above 0.5) are spread across all the spectrum of naturalness values. Mutants with low or no fault revelation (close to 0 value on the x-axis) have the most extreme negative values. Nevertheless, the visualization helps us demonstrate the absence of relation between the examined variables.

13.6.2 Additional Attempts with Negative Results

Our results are in a sense negative (the expected benefit was not reached). However, this could be attributed to a number of parameters that were not considered. To account for some of them, we repeated the experiment (without any success) with different parameters. Thus, we also used the different tokenizers included in TUNA, we also composed models by considering n-values up to 10, and also considered the use of a much larger training corpus, *i.e.,* train on the 20 (related) Java programs from Chapter 11 and we measured the number of tests and equivalent mutants required (by the naturalness-based and random mutant selection) to reach the same level of fault revelation. All these attempts yielded quite similar results and overall no significant differences between naturalness-based and random mutant selection was found.

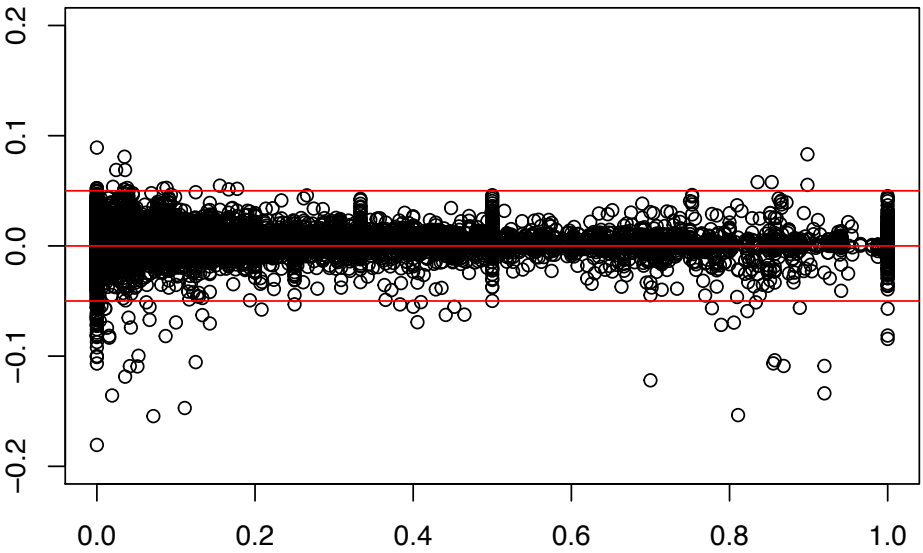


Figure 13.4: Fault Revelation probabilities and naturalness

The x-axis records fault revelation probability of each mutant (measured by counting the number of tests that kill the mutant and at the same time expose a real fault) and the y-axis records the naturalness value of every mutant.

13.7 Threats to Validity

The generalizability of the results is a common threat to the external validity of experimental study. To mitigate this threat, the use of real-world projects with real faults was considered.

A potential threat affecting the internal validity of this study stems from the sets of mutants and test suites which were used. We used state-of-the-art mutation testing tools supporting all the mainstream mutation operators [102].

To compose the test pools, multiple test suites that were generated by state-of-the-art test generation tools were used, *i.e.,* Randoop [142] and EvoSuite [60]. Although it is possible that different tests may lead to different results, the practice employed reflects what current test case generation research has to offer in large-scale experiments.

Threats that affect the construct validity of our study concerns the metrics we used. To evaluate mutant ranking we used fault-revealing mutants and fault revelation probabilities approximated by multiple test executions. These metrics are appropriate since fault revelation forms the purpose of testing.

Regarding the cost of mutation we measured and controlled the number of mutants. This is an intuitive choice that assumes that the cost of mutation testing is dependent on the number of the mutants involved. Yet, it might not represent the actual cost of the mutation testing practice. Here it must be mentioned that the cost of computing the naturalness is not considered as this practice requires one-off computation, common to the whole targeted code-based.

13.8 Related Work on Mutation Testing

Mutation Testing is a well studied, fault-seeding technique with a rich background both in theoretical and practical advances. A summary of these advances can be found in the recent survey of Papadakis *et al.,* [147] which summarizes the advances in the area between 2007–2016, complementing previous surveys [138, 87].

The quality of mutants and how to generate "good" mutants have concerned researchers for many years. This problem has many facets. First, the question "what changes should be applied to the program under test" can be posed. This is directly related to the mutation operators that one should use. Although mutation research has expanded the available mutation operators to handle multiple and diverse artefacts, ranging from mobile applications [54, 112] to models [34, 55], and includes specialized sets of operators, *e.g.,* energy-[85], security-[116] and memory-related [190] ones, it is not clear what constitutes a "good change" for mutant creation. Ultimately, a "good" mutant will be a fault-revealing one (for testing purposes), *i.e.,* it will be killed by test cases that reveal underlying faults in the program under test [46].

Offutt *et al.,* [139] introduced a theoretical model of the "size" of program faults which makes the separation between its syntactic and semantic characteristics. The syntactic size of a fault is related to the source code of the program under test and how it differs from its correct counterpart and the semantic size to the divergence between the program under test and its specification due to the presence of the fault. Thus, the authors suggest that mutants having a small semantic size are more valuable to testing.

Semantic mutation testing has been proposed as a different way to generate mutants that affect the semantics of the language of the artefact under test rather than the syntax [50, 53]. Thus, the semantic mutants simulate a different category of faults than the traditional ones and are more useful in several scenarios. Since semantic mutants are realized by syntactic changes in the source code of the program under test, naturalness can be used to evaluate their quality.

Sridharan and Namin [172] attempt to rank mutants by focusing on mutation operators that are likely to generate mutants that will not be killed by a specific test suite. Their approach is based on a probabilistic, Bayesian model which analyses a small portion of the generated mutants and the available test suite to rank the whole set of mutants. The basic difference between this approach and the previous ones is that they depend on the available test suite whereas our approach leverage mutants' naturalness and is applied statically.

Other studies attempt to create mutation operators that resemble real faults by analysing previous faults that developers have made. Brown *et al.,* [41] mine fault-fixing commits from the version control history of projects and extract fault-fixing patterns. Based on these patterns, they propose new mutation operators that reverse the patterns, thus, creating faulty program versions (mutants). Linares-Vásquez *et al.,* [112] created a taxonomy of faults found in Android applications and propose a new set of Android mutation operators based on these patterns. However, the utility of these techniques have not yet been evaluated wrt. to their ability to reveal faults.

13.9 Conclusion

Naturalness has been used for a large variety of Software Engineering tasks, in this chapter we addressed the fourth challenge by evaluating its use in the context of "fault-revealing" mutant selection. We investigated whether a link exists between the naturalness of mutants and the semantic alterations of real faults and found no evidence of such link. We also demonstrated that the naturalness-based mutant selection performs similar (slightly worse) than the random mutant selection. The findings also suggest that mutants (and their locations) coupled with faults are both natural and unnatural.

These results are of interest for both the software engineering and testing community. They indicate for the first one that as promising as naturalness of software seems, it has its limit. While for the second, it increases the understanding of the interconnections of program syntax, program semantics and software faults, as it confutes the natural intuition.

14

Wrapping Up Part IV

The previous part largely focused on the evaluation, replication and comparison of VPM. In this part, we developed new approaches based on the naturalness of software and therefore addressed the third challenge of the thesis. Yet to develop this new approaches the right parameters needed to be found. Thus in Chapter 11 an empirical study on the effect of parameters on naturalness was presented. This study revealed that the choice of the code representation was of utmost importance as models trained with different code representations were learning different things. Based on this experience, the study of Chapter 9 was extended in Chapter 12 with 2 novel VPM approaches, one using solely naturalness values as features and one combining code metrics values with naturalness. The results of the first one were disappointing and validated that naturalness could not be used alone. The results of the second, on the other hand, were really interesting as the precision compared to Code metrics alone was improved in most of the case. Moreover, it turned out that the combination with naturalness can reduce the sensitivity of the approaches to issues such as already fixes vulnerabilities and mislabeling noise, which are two of the main issues that were pointed out in the original study. Naturalness can be used for a variety of tasks and as the results obtained out of the first two chapters of this part, Chapter 13 attempted to apply it to a different context, mutation testing, which is in line with the fourth and last challenge of this thesis. More precisely, this chapter investigated whether there was a link between naturalness and the fault revelation utility of mutants. The results turned out to be negative with no evidence of a link. Still, they were considered as interesting as they confute a natural intuition and indicates that despite its great potential naturalness of software is no "pot of gold", which the results of the VPM approach based only on naturalness confirm.

Part V

Conclusion

15

Conclusion

This chapter concludes the dissertation and presents future research directions.

Contents

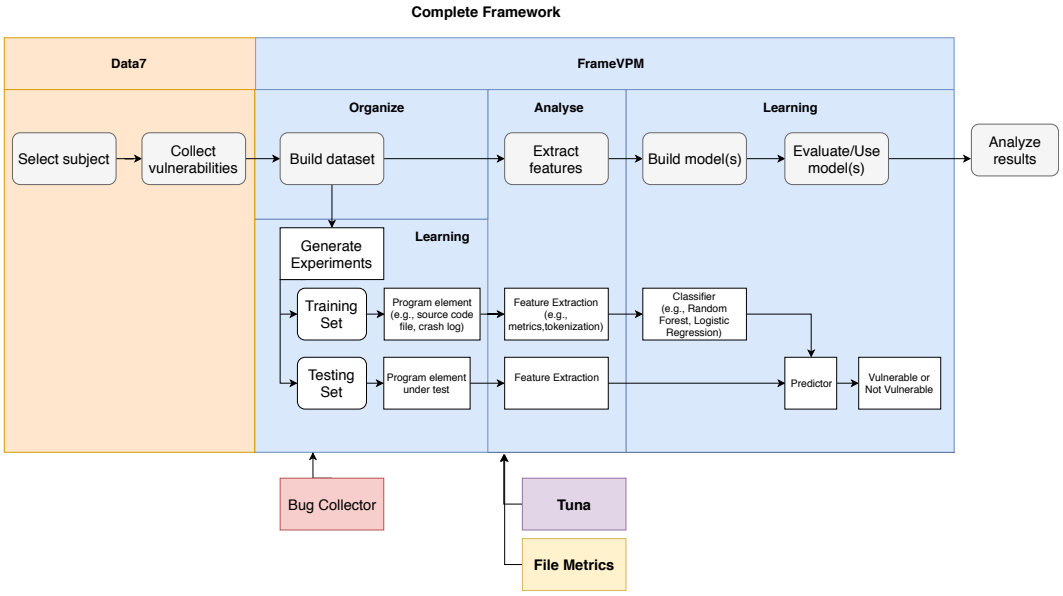


Figure 15.1: Complete Framework

This chapter is organised as follows. Section 15.1 summarises the contributions of this dissertation before Section 15.2 discusses potential directions for future work.

15.1 Summary

The threat posed by vulnerabilities is increasing as software becomes pervasive. To deal with this issue, a number of strategies have been devised over time. Still, the best cure remains prevention. Thus many software vendors integrated security policy and inspection to their development process. Yet, uncovering vulnerabilities requires an "attacker mindset" [120], which is not evident for most developers. Additionally, covering in a continuous way the entire code base is impractical and too costly especially for projects of millions LoC. In order to guide security inspection efforts, researchers suggested and evaluated various types of methods. Among them, vulnerability prediction modelling techniques offer promising results. Vulnerability Prediction Modelling is an active research area, appeared in 2007 by the seminal work of Neuhaus *et al.*, [133]. VPM uses supervised learning techniques to build prediction models based on a set of features and then used these models to evaluate the likelihood, to be vulnerable, of software components. Yet and despite, the growing attention paid to this area of research, there is a clear lack of replication and comparison studies of the approaches that have been suggested over time, which hinders the development of the area. One of the possible reasons for that is the absence of established publicly available datasets and the difficulty to create them. Additionally, as most of the existing studies do not provide any replication framework, such a comparative study is hard to make.

In this context, we identified and addressed two main challenges: (1) Collecting vulnerabilities in order to make a reliable, evolutive, multi-projects and large dataset enabling vulnerability prediction modelling, (2) Replicating and Comparing existing approaches.

Additionally, we proposed and investigated the use of naturalness (cross entropy measure) as a potential candidate feature. This proposition was inspired by its successful application on various software engineering tasks especially DPM [156].

This led to identification of two additional challenges addressed in this dissertation: (3) Developing a new VPM approach based on the Naturalness of Software, (4) Exploring the use of the Naturalness of Software for other Software Engineering tasks.

This work presented a series of frameworks that can be combined to form a complete framework for the development, evaluation, replication and comparison of vulnerability prediction model approaches along with studies that motivated and validated those frameworks. The architecture of the complete framework is depicted in Figure 15.1.

In the first part, we started by introducing the context, technical background and the challenges in Chapter 1. Then, we discussed the state of the art in Chapter 2.

The second part focused on addressing the first challenge of collecting the vulnerabilities. Chapter 3 first presented a manual analysis of Android vulnerabilities. The study emphasized on three properties of vulnerabilities, their root causes, complexity and location. The studies revealed that vulnerabilities are almost always located among the most complex function, which confirmed the choice of Shin *et al.,* [169] to use it as a feature for vulnerability prediction. More importantly the study highlightened the issue faced by researching when trying to collect information on vulnerabilities.
Chapter 4 offered a solution to this issue, which is as well the first challenge by introducing an extensible framework that automatically collects vulnerability information and fixes from the NVD and the version history of software projects. The framework currently supports 4 projects, *i.e.,* the Linux kernel, Wireshark, OpenSSL and Systemd and can seamlessly update its information.
Chapter 5 then presented an analysis of OpenSSL and the Linux Kernel vulnerability based on the data collected by the framework. In this analysis, a profile of each category of vulnerability present in those systems was built using information such as severity, code metrics and location. The results of this profiling were full of insight opening new perspective for vulnerability prediction. Hence, the profiles showed that the profiles are type-specific meaning that vulnerability type prediction modelling. The results also suggested that severity could be used as a target for prediction. Finally, the "expected" discovery of the fact that these profiles are system specific cast a shadow on the long-lasting dream of vulnerability prediction modelling, cross project prediction.
Finally, Chapter 6 concluded this part by summarizing its content.

The third part was guided by the second challenge of replicating and comparing the existing VPM approaches. Chapter 7 presented a replication study of three of the main VPM approaches, *i.e.,* includes and function calls as suggested by Neuhaus *et al.,* [133], Code Metric suggested by Shin *et al.,* [167] and Text Mining proposed by Scandariato *et al.,* [161]. Results demonstrated the importance of the evaluation methodology, a parameter often ignored by previous studies. They also showed that when training on past vulnerabilities to predict "future" ones, the models based on Includes and Function Calls offers an interesting trade-off time to compute/results, while Text Mining clearly outperforms the others in terms of "raw" performances, *i.e.,* not considering time to compute.

This replication study highlighted the difficulty to replicate (finding all the right settings) and compare (using the same evaluation methodology) existing VPM approaches. Thus based on the experience built upon the previous chapter, Chapter 8 introduced a framework to evaluate, replicate and compare VPM approaches. This framework is built on top of the Data7 framework from Chapter 4 and is composed of three distinct parts each one handling specific aspect of the evaluation models. The framework supports the integration of new approaches and currently offer 6 different approaches.
Chapter 9 then presented using this framework the largest empirical study on VPM. This study confirmed the good performance of the different approaches when trained on sufficient and accurately labelled data. However, it was as well observed that the performances become poor when the models are placed against more realistic data, *i.e.,* partial and miss-labelled data. This difference was demonstrated to be mainly due to the lack of vulnerability example referred as the "needle effect" by Shin *et al.,* [164] and not to the fact that data are mislabelled as one could expect. Finally, the poor performance of the models to detect a fix of vulnerabilities were confirmed, which supports the idea that VPM should be used as a guide and not as detectors.
Similarly to the previous part, Chapter 10 concluded this part with a summary of it.

The fourth part investigated the use of the Naturalness of Software. Chapter 11 presented a study on the impact of the choice of smoothing, tokenizer, unknown threshold and n values on the predicting ability of the N-Gram models that are used to compute naturalness. In particular, the study investigated whether this choice has the potential to impact previous and future findings and it turned out that the answer was positive. The major impacting factor found was the way to feed the source code to the N-Gram model, *i.e.,* the code representation. Interestingly, it appeared that the most interesting code representation to find defects was the closest to the human representation. These results led to the conclusion that in order to make the most out of the computation of naturalness, several code representations need to be considered.
Using these findings, Chapter 12 tackled the third challenge by introducing two novel VPM approaches in an extension of the study from chapter 9. The first one used only naturalness values computed using different settings, while the second one combined this approach with the code metric one. It turned out that naturalness cannot be used alone but is interesting when combined to Code Metrics, with observed effect such as an increase in precision and a reduction in sensitivity to the problem of vulnerability fixes and mislabelling noise. Overall, it seems like naturalness can be a good complement to model helping them to generalize better.
Finally, Chapter 13 addressed the last challenges by investigating whether a link existed between the naturalness of mutant and their "fault-revealing" capability. Following the findings of Chapter 11 several configurations were considered but the results turned out to be negative. These results remain nonetheless interesting as they contradict the natural intuition and as when put together with the negative results of the naturalness only VPM approaches remind to researchers that as promising as "Naturalness" seems it is no "Pot of Gold".
In a similar way to Chapter 6 and 10, Chapter 14 concluded this part.

15.2 Future research directions

15.2.1 Extending the Data7 Framework

Chapter 4 introduced the Data7 framework that is used all along the dissertation. The framework is mature but still face some limitations. First of all, the data are stored in a flat way, which means that no navigation is possible in the dataset which is not problematic for vulnerability prediction modelling but is more troublesome in the case of vulnerability analysis. Second, the data are not versioned. The state of knowledge on vulnerabilities evolves in time, *i.e.,* new exploits and fixes can be reported. In the current implementation every update on the vulnerability information overwrite the previous entry. This is not especially problematic but when investigating the problem of mislabelling noise it would be preferable to know the exact state of knowledge at the time we desire to investigate. These two concerns can be easily handled through the use of temporal graph databases such as Greycat [74]. The navigation would be solved by the transposition of the flat data into a graph, while the versioning would be ensured by the temporality. A first step in this direction, not yet available, has already been made.

15.2.2 Broadening the Scope of VPM

Chapter 5 showed that metric profile differs depending on the type of vulnerability. These findings open the door to approach predicting the type of vulnerability instead of just vulnerability. This type of approach could take several forms, a first possibility would be an approach that target a specific type of vulnerability alike what we did with the severity. Yet, this is likely to come at the price of an increased "needle effect" as we reduce the number of elements in the minority class. A second possibility would be to directly try to predict the type of vulnerability. This doesn't solve the needle effect but transform the classification from a binomial to a multinomial problem which complicates the measures of performances, still it might be an interesting setting to experiment. A last solution would be to chain a traditional VPM with a type specialized one. This would reduce the "needle effect" but the impressions of both models would then add up. None of these solutions are perfect, but they should nonetheless be investigated.

Another way to broaden the scope of VPM is to combine it with DPM to perform a trinomial classification, *i.e.,* creating an approach building model that predicts defects and vulnerabilities. Such approach could potentially build better model as the spread between the different class would be more balanced. Additionally, it would likely reduce the number of components that are predicted as clear when they are vulnerable. This could be of great use for security inspection effort, as practitioners could first focus on components that are predicted as vulnerable and then the one predicted as defect. The evaluation of such models is already possible in FrameVPM and initial evaluation have already been made.

15.2.3 Analyzing Vulnerability Fixes

The Data7 framework stores the fixes of the vulnerabilities, in this dissertation those are not fully exploited, *i.e.,* the vulnerable form is almost always the only one studied. Yet, vulnerability fixes are essential to the understanding of what makes a vulnerability a vulnerability. Thus, further analysis would be of great interest. This could take the form of a standard analysis like the one presented in chapter 5, but could also lead to the creation of a model detecting them. Indeed, not every vulnerability has declared fixes but they are still supposed to have been fixed. Thus predicting whether a fix is a vulnerability fix could help detect those fix or even better detect unknown vulnerability corrected by the fix. To create this kind of model, some feature specific to code changes would be required. Naturalness could be an interesting source of features, either by computing the diff between the naturalness of the vulnerable and fixed version of the component alike what was done for mutants in Chapter 13, either by usual textual representation of the patch like line diff or AST diff [57]. ged introduced in Chapter 5 might as well be an interesting feature to explore.

List of Papers and Tools

Published papers included in the dissertation

- Profiling android vulnerabilities (QRS 16)
  M Jimenez, M Papadakis, TF Bissyandé, J Klein [91]

- Vulnerability Prediction Models: A case study on the Linux Kernel (SCAM 16)
  M Jimenez, M Papadakis, Y Le Traon [93]

- An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel(APSEC 16)
  M Jimenez, M Papadakis, Y Le Traon [92]

- Enabling the Continuous Analysis of Security Vulnerabilities with VulData7 (SCAM 18)
  M Jimenez, Y Le Traon, M Papadakis [90]

- On the impact of tokenizer and parameters on n-gram based code analysis (ICSME 18)
  M Jimenez, M Cordy, Y Le Traon, M Papadakis [89]

- Are mutants really natural? A study on how "naturalness" helps mutant selection (ESEM 18)
  M Jimenez, T Checkam, M Cordy, M Papadakis, M Kintis, Y Le Traon, M Harman [94]

Published papers not included in the dissertation

- Analyzing complex data in motion at scale with temporal graphs (SEKE 17)
  T Hartmann, F Fouquet, M Jimenez, R Rouvoy, Y Le Traon [74]

Paper currently under Submission

- Do Prediction Models Find Important and Severe Vulnerabilities?

Tools and Dataset developed during the thesis

- Data7 Framework `https://github.com/electricalwind/data7`
- VPM Framework `https://github.com/electricalwind/framevpm`
- TUNA: TUning Naturalness-based Analysis (ICSME 18)
  M Jimenez, M Cordy, Y Le Traon, M Papadakis [88]
  `https://github.com/electricalwind/tuna`
- FileMetrics `https://github.com/electricalwind/FilesMetrics`
- Android Vulnerabilities dataset `https://github.com/electricalwind/ReasearchData/tree/master/AndroidVulnerabilities/Android`

Bibliography

[1] Android source code on github. https://github.com/android.

[2] Android source code on googlesource. https://android.googlesource.com.

[3] Apache commons statement on deserialization vulnerability. https://blogs.apache.org/foundation/entry/apache_commons_statement_to_widespread.

[4] cloc tool. http://cloc.sourceforge.net/.

[5] Cve-2018-10840 fixing commit:. https://github.com/torvalds/linux/commit/8a2b307c21d4b290e3cbe33f768f194286d07c23.

[6] Cve website. https://cve.mitre.org.

[7] Cwe home page. https://cwe.mitre.org/data/.

[8] Find bugs security. https://find-sec-bugs.github.io.

[9] Findsecbugs. https://find-sec-bugs.github.io/.

[10] Fortify static code analyzer. https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview.

[11] Fuzzing. https://www.owasp.org/index.php/Fuzzing.

[12] Graph edit distance implementation. https://github.com/haakondr/NLP-Graphs/.

[13] Hacking and blue boxes. https://infostory.com/2011/11/20/hacking-and-blue-boxes/.

[14] Heartbleed. http://heartbleed.com.

[15] National vulnerability database:. https://nvd.nist.gov.

[16] Openssl usage statistics. https://trends.builtwith.com/Server/OpenSSL.

[17] Scitool complexity metric. https://scitools.com/support/metrics_list/?metricGroup=complex.

[18] Shellshock: Cve-2014-6271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

[19] A software bug taxonomy. http://www.cs.nyu.edu/~lharris/content/bugtaxonomy.html.

[20] Source code analysis tools. https://www.owasp.org/index.php/Source_Code_Analysis_Tools.

[21] Top levels of boris beizer's bug taxonomy. http://c2.com/cgi/wiki?SourcesOfBugs.

[22] Vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss.

[23] Weka: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/.

[24] O.H. Alhazmi, Y.K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security*, 26(3):219 – 228, 2007.

[25] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[26] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on*, pages 151–156. IEEE, 2016.

[27] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Software metrics and security vulnerabilities: Dataset and exploratory study. In *Dependable Computing Conference (EDCC), 2016 12th European*, pages 37–44. IEEE, 2016.

[28] Paul Ammann, Marcio Eduardo Delamaro, and A. J. Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.

[29] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[30] Negar Armaghan and Jean Renaud. Evaluation of knowledge management in an organisation. *Journal of Information & Knowledge Management*, 16(01):1750006, 2017.

[31] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 97–106. IEEE, 2011.

[32] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pages 804–810. IEEE, 2009.

[33] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[34] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa. Mutation testing of "go-back" functions based on pushdown automata. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 249–258, March 2011.

[35] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, 2013.

[36] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 105–115, 2014.

[37] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 257–268. ACM, 2014.

[38] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu. Mutation-aware fault prediction. In *Procs. of the 25th International Symposium on Software Testing and Analysis*, ISSTA'16, pages 330–341. ACM, 2016.

[39] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 105–114, New York, NY, USA, 2010. ACM.

[40] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114. ACM, 2010.

[41] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 511–522, New York, NY, USA, 2017. ACM.

[42] K. K. Chaturvedi and V. B. Singh. Determining bug severity using machine learning techniques. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–6, Sept 2012.

[43] K. K. Chaturvedi and V.B. Singh. An empirical comparison of machine learning techniques in predicting the bug severity of open and closed source projects. *Int. J. Open Source Softw. Process.*, 4(2):32–59, April 2012.

[44] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357, 2002.

[45] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *CoRR*, abs/1803.07901, 2018.

[46] 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 *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 597–608, Piscataway, NJ, USA, 2017. IEEE Press.

[47] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Comput. Speech Lang.*, 13(4):359–394, October 1999.

[48] Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1963–1969. ACM, 2010.

[49] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[50] J. A. Clark, H. Dan, and R. M. Hierons. Semantic mutation testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 100–109, April 2010.

[51] William W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, ICML'95, pages 115–123, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[52] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the'security vulnerability likelihood'of software functions. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 266–274. IEEE, 2003.

[53] H. Dan and R. M. Hierons. Smt-c: A semantic mutation testing tools for c. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 654–663, April 2012.

[54] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015.

[55] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 655–666, New York, NY, USA, 2016. ACM.

[56] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007.

[57] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[58] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 65–73. ACM, 2016.

[59] Apache Foundation. Apache commons. `http://commons.apache.org`, 2017. Accessed 2017-10-20.

[60] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[61] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information & Software Technology*, 76:135–146, 2016.

[62] William A. Gale and Kenneth W. Church. Poor estimates of context are worse than none. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '90, pages 283–287, Stroudsburg, PA, USA, 1990. Association for Computational Linguistics.

[63] William A. Gale and Kenneth W. Church. What's wrong with adding one. In *Corpus-Based Research into Language. Rodolpi*, 1994.

[64] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13, 2010.

[65] Michael Gegick, Pete Rotella, and Laurie Williams. Predicting attack-prone components. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 181–190. IEEE, 2009.

[66] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification throughcode-level metrics. In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 31–38. ACM, 2008.

[67] Michael C Gegick, Laurie Ann Williams, and Mladen A Vouk. Predictive models for identifying software components prone to failure during security attacks. Technical report, North Carolina State University. Dept. of Computer Science, 2008.

[68] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.

[69] Cem Kaner Giri Vijayaraghavan. Bug taxonomies: Use them to generate better tests. *Star East*, pages 1–40, 2003.

[70] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 18–21. ACM, 2018.

[71] Maggie Hamill and Katerina Goseva-Popstojanova. Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. *Software Quality Journal*, 23(2):229–265, 2015.

[72] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[73] M. Harman, S. Islam, Y. Jia, L. L. Minku, F. Sarro, and K. Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In *Procs. of the International Symposium on Search-Based Software Engineering (SSBSE'14)*, pages 240–246. Springer, 2014.

[74] Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *The 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17)*, page 6. KSI Research, 2017.

[75] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, 15(2):147–165, 2010.

[76] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 157–167, Piscataway, NJ, USA, 2015. IEEE Press.

[77] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[78] Aram Hovsepyan, Riccardo Scandariato, and Wouter Joosen. Is newer always better?: The case of vulnerability prediction models. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 26. ACM, 2016.

[79] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10. ACM, 2012.

[80] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[81] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM.

[82] Xing Hu, Yuhan Wei, Ge Li, and Zhi Jin. Codesum: Translate program language to natural language. *CoRR*, abs/1708.01837, 2017.

[83] IDC. Android market share. `https://www.idc.com/promo/smartphone-market-share/os`, 2018. Accessed 2017-10-20.

[84] inquirer. Pc sales in 2016. `https://www.theinquirer.net/inquirer/news/3002448/pc-sales-down-in-2016-and-will-continue-declining-in-2017-say-analysts`, 2018. Accessed 2017-10-20.

[85] Reyhaneh Jabbarvand and Sam Malek. μDroid: An energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 208–219, New York, NY, USA, 2017. ACM.

[86] Frederick Jelinek and Robert L. Mercer. Interpolated estimation of markov source parameters from sparse data. In *In Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, The Netherlands: North-Holland, May 1980.

[87] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[88] Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. Tuna: Tuning naturalness-based analysis. In *34th IEEE International Conference on Software Maintenance and Evolution, Madrid, Spain, 26-28 September 2018*, 2018.

[89] Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. On the impact of tokenizer and parameters on n-gram based code analysis. In *34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29 2018*, 2018.

[90] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. Enabling the continous analysis of security vulnerabilities with vuldata7. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2018.

[91] Matthieu Jimenez, Mike Papadakis, Tegawende F. Bissyandé, and Jacques Klein. Profiling android vulnerabilities. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 222–229, Aug 2016.

[92] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, pages 105–112, 2016.

[93] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Vulnerability prediction models: A case study on the linux kernel. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 1–10, 2016.

[94] Matthieu Jimenez, Thierry Titcheu Checkham, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural? a study on how "naturalness" helps mutant selection. In *12th International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, 11-13 October 2018*, 2018.

[95] Karen Sparck Jones. *Natural Language Processing: A Historical Review*, pages 3–16. Springer Netherlands, Dordrecht, 1994.

[96] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440, 2014.

[97] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 612–615, 2011.

[98] Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 35(3):400–401, 1987.

[99] Taghi M Khoshgoftaar, Ruqun Shan, and Edward B Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 301–310. IEEE, 2000.

[100] 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.

[101] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *International Working Conference on Source Code Analysis and Manipulation*, pages 147–156, 2016.

[102] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, 2018.

[103] Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume I, pages 181–184, Detroit, Michigan, May 1995.

[104] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning*

and Programming Languages, MAPL 2017, pages 35–42, New York, NY, USA, 2017. ACM.

[105] Igor Kononenko. On biases in estimating multi-valued attributes. In *IJCAI'95*, pages 1034–1040, 1995.

[106] Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, West Lafayette, IN, USA, 1998.

[107] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 571–582, 2016.

[108] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10, May 2010.

[109] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 249–258, March 2011.

[110] Michele Lanza, Andrea Mocci, and Luca Ponzanelli. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software*, 33(6):102–105, 2016.

[111] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, March 2017.

[112] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 233–244, New York, NY, USA, 2017. ACM.

[113] Richard R. Linde. Operating system penetration. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, pages 361–368, New York, NY, USA, 1975. ACM.

[114] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 599–609, 03 2016.

[115] Han Liu. Towards better program obfuscation: Optimization via language models. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 680–682, New York, NY, USA, 2016. ACM.

[116] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans. Towards security-aware mutation testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 97–102, March 2017.

[117] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. A genetic algorithm to configure support vector machines for predicting fault-prone components. In *Product-Focused Software Process Improvement - 12th International Conference, PROFES 201. Proceedings*, pages 247–261, 2011.

[118] 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.

[119] Thomas J. McCabe. A complexity measure. In *ICSE '76*, pages 407–, 1976.

[120] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[121] Gary McGraw. Automated code review tools for security. *IEEE Computer*, 41(12):108–111, 2008.

[122] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security & Privacy*, 2(5):81–85, 2004.

[123] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[124] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355, Sept 2008.

[125] Katsiaryna Mirylenka, George Giannakopoulos, Le Minh Do, and Themis Palpanas. On classifier behavior in the presence of mislabeling noise. *Data Min. Knowl. Discov.*, 31(3):661–701, 2017.

[126] 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*, page 4. ACM, 2015.

[127] Sara Moshtari and Ashkan Sami. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1415–1421. ACM, 2016.

[128] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17, 2013.

[129] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[130] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[131] Graham Neubig. Kyoto language modeling toolkit. `https://github.com/neubig/kylm`, 2017. Accessed 2017-10-25.

[132] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *USENIX Annual Technical Conference*, 2009.

[133] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *CCS'07*, page 529, 2007.

[134] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8:1–38, 1994.

[135] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[136] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[137] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, page 3. ACM, 2010.

[138] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In W.Eric Wong, editor, *Mutation Testing for the New Century*, volume 24 of *The Springer International Series on Advances in Database Systems*, pages 34–44. Springer US, 2001.

[139] A. Jefferson Offutt and J. Huffman Hayes. A semantic model of program faults. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 195–200, New York, NY, USA, 1996. ACM.

[140] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 572–588, New York, NY, USA, 2015. ACM.

[141] Andy Ozment. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 6–11. ACM, 2007.

[142] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 815–816, 2007.

[143] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*, pages 543–548. IEEE, 2015.

[144] Yulei Pang, Xiaozhen Xue, and Huaying Wang. Predicting vulnerable software components through deep neural network. In *Proceedings of the 2017 International Conference on Deep Learning Technologies*, pages 6–10. ACM, 2017.

[145] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *13th International Workshop on Mutation Analysis (MUTATION'18)*, 2018.

[146] 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, pages 354–365, New York, NY, USA, 2016. ACM.

[147] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2018.

[148] 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 *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 537–548, 2018.

[149] Java Parser. Java parser github. `https://github.com/javaparser/javaparser`, 2017. Accessed 2017-10-20.

[150] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437. ACM, 2015.

[151] Marcus Piancó, Baldoino Fonseca, and Nuno Antunes. Code change history and software vulnerabilities. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 6–9. IEEE, 2016.

[152] Rene Pickhardt, Thomas Gottron, Steffen Staab, Paul Georg Wagner, Till Speicher, and Typology Gbr. A generalized language model as the combination of skipped n-grams and modified kneser-ney smoothing. In *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

[153] Nawa Raj Pokhrel, Hansapani Rodrigo, and Chris P Tsokos. Cybersecurity: Time series predictive modeling of vulnerabilities of desktop operating system using linear and non-linear approach. *Journal of Information Security*, 8(04):362, 2017.

[154] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. pages 39–40, 07 2016.

[155] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *ACL*, 2017.

[156] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.

[157] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[158] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Trans. Systems, Man, and Cybernetics*, 13, 1983.

[159] Juliana Saraiva, Christian Bird, and Thomas Zimmermann. Products, developers, and milestones: How should i build my n-gram language model. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 998–1001, New York, NY, USA, 2015. ACM.

[160] Federica Sarro, Sergio Di Martino, Filomena Ferrucci, and Carmine Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012*, pages 1215–1220, 2012.

[161] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[162] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 18–27, New York, NY, USA, 2006. ACM.

[163] Claude Elwood Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30:50–64, January 1951.

[164] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.

[165] Abhishek Sharma, Yuan Tian, and David Lo. NIRMAL: automatic identification of software relevant tweets leveraging language model. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 449–458, 2015.

[166] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. on Softw. Eng.*, 40(6):603–616, 2014.

[167] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[168] 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, New York, NY, USA, 2008. ACM.

[169] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50. ACM, 2008.

[170] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7. ACM, 2011.

[171] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[172] Mohan Sridharan and Akbar Siami Namin. Prioritizing mutation operators based on importance sampling. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 378–387, 2010.

[173] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Trans. Reliability*, 66(1):17–37, 2017.

[174] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Ken-ichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 812–823, 2015.

[175] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 321–332, 2016.

[176] Terrier. Terrier open source search engine. `http://terrier.org`, 2017. Accessed 2017-10-20.

[177] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 199–208. IEEE Press, 2015.

[178] Christopher Theisen, Brendan Murphy, Kim Herzig, and Laurie Williams. Risk-based attack surface approximation: how much data is enough? In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 273–282. IEEE Press, 2017.

[179] Y. Tian, D. Lo, and C. Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *2012 19th Working Conference on Reverse Engineering*, pages 215–224, Oct 2012.

[180] Yuan Tian, Nasir Ali, David Lo, and Ahmed E. Hassan. On the unreliability of bug severity data. *Empirical Software Engineering*, 21(6):2298–2323, Dec 2016.

[181] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[182] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Jrnl. Educ. Behav. Stat.*, 25(2):101–132, 2000.

[183] James Walden and Maureen Doyle. Savi: Static-analysis vulnerability indicator. *IEEE Security & Privacy*, 10(3):32–39, 2012.

[184] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 23–33. IEEE, 2014.

[185] Shaohua Wang, Foutse Khomh, and Ying Zou. Improving bug localization using correlations in crash reports. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 247–256. IEEE Press, 2013.

[186] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 708–719, New York, NY, USA, 2016. ACM.

[187] Laurie Williams et al. Toward the use of automated static analysis alerts for early identification of vulnerability-and attack-prone components. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, pages 18–18. IEEE, 2007.

[188] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inf. Theor.*, 37(4):1085–1094, September 2006.

[189] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[190] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Information and Software Technology*, 81(Supplement C):97 – 111, 2017.

[191] Shir Yadid and Eran Yahav. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 98–111, New York, NY, USA, 2016. ACM.

[192] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 13–13. USENIX Association, 2011.

[193] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. ACM, 2012.

[194] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[195] G. Yang, T. Zhang, and B. Lee. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 97–106, July 2014.

[196] Zhe Yu, Christopher Theisen, Hyunwoo Sohn, Laurie Williams, and Tim Menzies. Cost-aware vulnerability prediction: the harmless approach. *arXiv preprint arXiv:1803.06545*, 2018.

[197] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

[198] Tao Zhang, Geunseok Yang, Byungjeong Lee, and Alvin T. S. Chan. Predicting severity of bug report by mining bug repository with concept profile. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1553–1558, New York, NY, USA, 2015. ACM.

[199] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. Combining software metrics and text features for vulnerable file prediction. In *Engineering of Complex Computer Systems (ICECCS), 2015 20th International Conference on*, pages 40–49. IEEE, 2015.

[200] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.

[201] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 421–428. IEEE, 2010.