Software Security Flaw Prediction Using Rich Contextualized Language Use Vectors: A Case Study on the Linux Kernel

by

Ghazal Fouladfard

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
March 2020

Abstract

One of the major threats to the security of software systems is the occurrence of security vulnerabilities, which can potentially cause a variety of problems including, but not limited to, information loss, privilege escalation, data breach, and system failure. Software vulnerability prediction is therefore a critical part of software engineering. A variety of approaches have been proposed to detect the most likely locations of vulnerabilities in large codebases. Many of the existing methods rely on traditional software metrics such as lines of code, complexity and code churn. In this study, we explored the possibility of using Rich Contextualized Language Use Vectors (RCLUVs) as a feature set for predicting vulnerabilities in the context of the Linux kernel.

The RCLUV of a source code file contains elements representing the frequency of each programming language feature being used, both individually and in the context of other features. This code profile is generated by parsing the source code of a program and analyzing the resulting parse tree.

We mined vulnerabilities reported in the National Vulnerability Database (NVD) and built a dataset containing all known vulnerable files in the 14-year history of the Linux kernel. We built and evaluated RCLUV-based prediction models using different machine learning algorithms under both experimental and realistic scenarios. Analysis of the learning curves of the models demonstrates that RCLUVs are effective

for training machine learning models to learn vulnerability patterns. Performance comparison of our models with four different popular vulnerability prediction models shows that our approach outperforms the models trained on *includes*, *function calls*, and *software metrics* in an experimental setup. Moreover, our models can successfully predict more than half of the future and unseen vulnerabilities in a real-life setting when given enough training data.

Acknowledgments

First of all, there are no proper words to convey my deep gratitude and respect for my research supervisor Prof. James R. Cordy for giving me the opportunity to work under his supervision, sharing his vision and providing me the necessary direction from the start of the research to the final phase. This accomplishment would not have been possible without his constant support, patience, and encouragement.

I would also like to thank my amazing husband, Nima, for always being by my side and helping me survive during my difficult times. Without his great companion, endless love and support, I could never sustain this hard journey.

Finally, I would like to deeply thank my parents, who have always encouraged me to follow my dreams. Their unconditional love and support made my journey far from home possible.

Statement of Originality

I hereby certify that the work of the author in this thesis is original, conducted under the supervision of Dr. James R. Cordy. Ideas and techniques that are not from the work of the author are fully acknowledged using citations or by paraphrasing if citations are not available, to indicate that these are the works of others.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we discuss the motivation and goal of this study. In addition, we explain our research questions and briefly go over our approaches to investigate each one. Finally, we discuss the contributions of our work followed by an outline of the rest of this thesis.

1.1 Motivation

One of the major threats to the security of software systems is the occurrence of security vulnerabilities. Detecting these vulnerabilities in the early stages is very important in software engineering because it would help reduce the costs of development and prevent damage to the reputation of the software companies. According to a recent report published by the *Information Technology Laboratory* (ITL), one of the major research components of the *National Institute of Standards and Technology* (NIST), the number of vulnerabilities reported almost doubled from 2016 to 2017, increasing from around 6,500 to over 14,000, and the number is continuing to rise dramatically [18] [8]. The same report also points out that the number of reported high and critical severity vulnerabilities are increasing rapidly.

Vulnerability prediction research has been very active and has achieved remarkable progress from about 2000 until now, but there are still many opportunities for further enhancements. Existing research shows that the performance of software vulnerability prediction is still far below the desired level.

Research has suggested that source code-related vulnerabilities are responsible for the majority of the exploits [21]. Consequently, many researchers have investigated various approaches to identify vulnerabilities in the software code base. A common approach is to use machine learning techniques to build vulnerability predication models and train the models to identify vulnerabilities based on different types of metrics. A large number of the existing techniques rely on common software engineering metrics such as source-code size, complexity, code-churn, and developer-activity. However, in this research, we present a novel approach for vulnerability prediction that leverages the analysis of raw source code files, which encodes the programming language feature use of source files into metric vectors as an alternative to traditional metrics to predict software vulnerabilities.

This research explores the use of Rich Contextualized Language Use Vectors (RCLUVs) in identifying vulnerable files, with the intention of helping software companies to detect vulnerabilities before deploying their software products to production. There are various reasons that motivated us to conduct this research. First, the effectiveness of using RCLUVs in defect prediction research recently published [27] encouraged us to utilize the same idea and apply it to the challenge of vulnerability prediction. Second, RCLUVs are based solely on the source code of the software systems, which means that there is no need to obtain any further information to build the feature vectors. Moreover, RCLUVs encode very useful and deep information of the

language feature use of the source code when compared them to other metrics such as Bag-of-Words (BoW) used in previous studies (e.g. [30]) that simply counts the frequency of code tokens. Clearly, a bag of code tokens does not necessarily capture the semantic structure of the code, especially its hierarchical and sequential nature. Therefore, we decided to conduct a vulnerability prediction study using RCLUVs as the predictive feature set.

1.2 Goal of the Thesis

The goal of this thesis is to explore software vulnerability prediction using RCLUVs in place of traditional metrics. We extracted RCLUVs by analyzing the parse tree of the source code, building vectors that encode the frequency of each programming language feature being used, both individually and in the context of other features in a program. We aimed to measure the effectiveness of utilizing RCLUVs as metrics to train machine learning algorithms, and compare the performance of such models to traditional metrics-based approaches.

Thesis statement: RCLUVs can be used as a practical and effective basis for software vulnerability prediction using machine learning.

In order to test this assertion, we first mined vulnerabilities reported in the National Vulnerability Database (NVD) and created a large dataset with all vulnerable components of Linux from 2005 to 2019. We then designed two research questions to explore our assertion, and designed a set of experiments to address each research question, which are discussed briefly in the following sections.

1.3 Research Questions

In order to address our thesis statement, we explore two specific research questions:

1.3.1 RQ1. How effective are machine learning models trained on RCLUVs in distinguishing vulnerable files from neutral files?

The answer to this research question demonstrates the actual prediction power of our models in discriminating vulnerable files from neutral files. We wanted to analyze whether RCLUVs are practical in predicting vulnerable files, independent of when the vulnerability was fixed. In order to serve this purpose, we built various machine learning models including random decision forest, gradient boosting, multi-layer perceptron neural networks, and logistic regression. Moreover, we performed a stratified 5-fold cross validation on the data extracted from the entire history of the Linux kernel and repeated this process 10 times. Finally, we calculated the average results of the precision, recall, accuracy and Matthews correlation coefficient (MCC) metrics per 10 iterations and compared our study with a recent study that replicated four different popular approaches for vulnerability prediction [15].

This is the most crucial research question that will enable the research community to decide whether the RCLUV-based code profile should be considered or not. All other possible research questions depend on the answer to this question.

1.3.2 RQ2. How effective are vulnerability prediction models trained on RCLUVs in predicting vulnerable files in future releases?

This research question aims to determine the practicality of our approach in real-world applications. In practice, in real-world applications, vulnerability prediction

models are used to detect new vulnerabilities in the future releases.

Another goal of this research question is to examine whether vulnerabilities share similar characteristics over time or not. Although the experiments done regarding the previous research question might demonstrate the capability of RCLUVs in distinguishing vulnerable files from neutral files, they can not confirm that our models are still effective in predicting future vulnerabilities. In order to explore this question, we decided to split our dataset according to 30 release points from release v2.6.25 to v4.10 of the Linux kernel. For each release point, we trained our models based on the files retrieved from the previous release points and tested our models based on the data retrieved after that release point. Therefore, we trained 60 machine learning models using random forest and XGBoost and analyzed their performance results.

1.4 Contributions

This thesis makes the following contributions:

- The new idea of using Rich Contextualized Language Use Vectors (RCLUVs) as a feature set for predicting vulnerabilities for the first time in the literature.
- A C programming language feature extractor tool for generating RCLUVs from the source code of a project.
- An RCLUV-based dataset of vulnerable and neutral files using the reports extracted from the Linux kernel section of NVD.
- Vulnerability prediction models based on RCLUVs and various machine learning algorithms along with the performance analysis of the models among themselves and in comparison to several existing approaches.

- A comparison of the results of vulnerability prediction models based on RCLUVs with four different popular vulnerability prediction approaches.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2 provides the background information needed to understand our work, as well as the literature review of related work in the vulnerability prediction area.

- Chapter 3 describes how we extracted and selected vulnerable and neutral files from NVD. We will also explain the process flow of the RCLUV feature extractor for building our dataset.

- Chapter 4 explores whether RCLUV-based code profiles can be used as features to effectively train machine learning algorithms for vulnerability prediction or not. Learning curves of several machine learning algorithms trained on RCLUV-based code profiles will also be analyzed. Finally, we will compare the performance of RCLUV-based models to four different popular vulnerability prediction approaches on a similar dataset.

- Chapter 5 evaluates the power of our proposed approach in predicting future vulnerabilities. We will compare the performance of our models with four popular existing approaches in the literature.

- Chapter 6 presents the contributions and limitations of this study along with discussions about the possible future work.

1.6 Summary

In this chapter we began with the motivation for this research. We briefly outlined the goal of this thesis and introduced our thesis statement. Then, we presented our research questions and provided a summary of our techniques to investigate each research question. Finally, we summarized the contributions of this thesis and gave an outline of the remaining chapters.

Chapter 2

Background and Related Work

In this chapter, we provide the relevant background information for this thesis and survey some of the related work. Initially, we describe some definitions of software vulnerability followed by a brief introduction to the National Vulnerability Database, the Linux kernel and programming language features. Subsequently, a brief overview of TXL and its recent applications is provided. Finally, we present a review of the related work on vulnerability prediction studies.

2.1 Background

2.1.1 Software Vulnerability

In this section we will explore the term "vulnerability" in the context of software security and provide definitions related to it. In 1998, Krsul in his PhD thesis [17] defined software vulnerability as:

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

Almost a decade later in 2007, Ozment [24] debated many proposed definitions

of software vulnerabilities and mentioned that Krsul's definition is the most accurate one. However, he suggested using the term “mistake" instead of “error" because using “error" is opposed to the definition in IEEE Standard Glossary of Software Engineering Terminology.

In addition, *Common Vulnerabilities and Exposures (CVE)* [19] offers an accurate explanation of vulnerability 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. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of a ected protocols or functionality in their entirety)."

Vulnerabilities can be considered as a subset of bugs in software systems. A defect in a computer program that results in an incorrect or unexpected outcome or unintended behaviour is regarded as a bug, while vulnerabilities are a category of bugs when one or more of the principles of security : *Confidentiality*, *Integrity* or *Availability* is impacted.

Software vulnerabilities exist since the advent of the computers. The history of security vulnerabilities started in 1965, when William D. Mathews from MIT found a vulnerability in the text editor of the CTSS running on an IBM 7094 [40]. The standard text editor was designed to be used by one user at a time, working in a directory. The issue occurred when multiple system programmers attempted to edit one file at the same time. As a result, the text editor swapped the message of the day file with the password file containing all the passwords. Therefore, the password file was displayed in clear text to each user that logged into the system.

2.1.2 National Vulnerability Database (NVD)

Significant advances have been made in understanding and management of vulnerabilities over the past two decades thanks to advent of the databases collecting publicly disclosed vulnerabilities. The National Vulnerability Database (NVD) [2] was established in 2005. It consists of a repository of data about software vulnerabilities along with vulnerability impact metrics such as vulnerability severity, encoded using the Common Vulnerability Scoring System (CVSS). NVD is built upon Common Vulnerabilities and Exposures (CVE) which is a common identifier that is assigned to a vulnerability to facilitate the sharing of relevant information about that flaw [2]. The search interface of NVD provides the opportunity for public and private sectors to gain insights about the nature and severity of the hundreds of new vulnerabilities discovered every month. Figure 2.1 shows the number of vulnerabilities reported by year in the CVE Database since 1999.

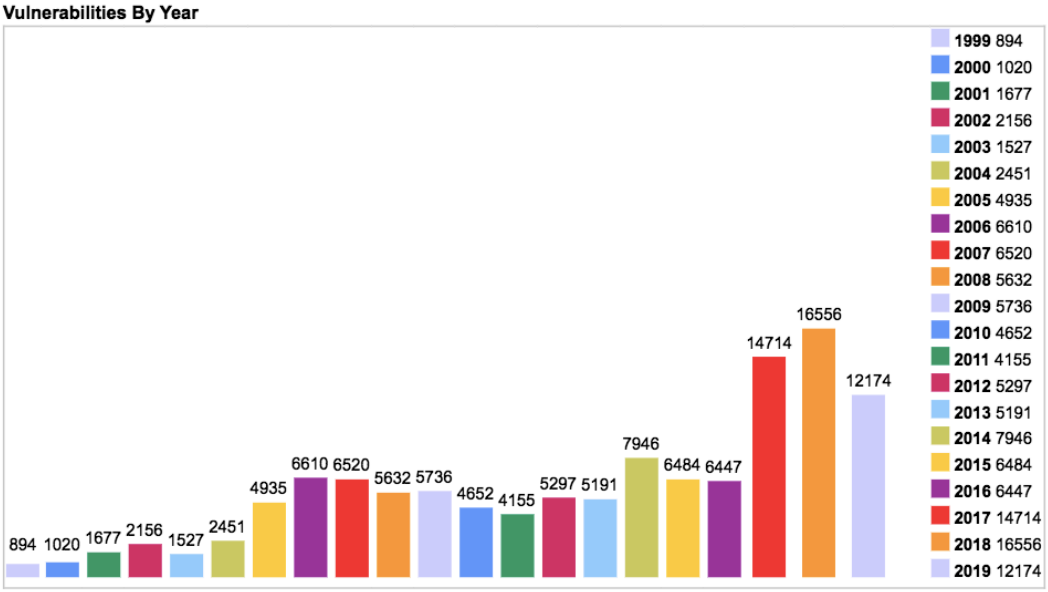

Figure 2.1: Number of vulnerabilities reported by year in the CVE Database since 1999.

In this thesis, we rely on the NVD as a source of information for the disclosed vulnerabilities in the Linux kernel.

2.1.3 The Linux Kernel

Linux is the best-known free and open source operating system. In 1991, Linus Torvalds created Linux kernel as a hobby for his personal computer. The Linux kernel is now shipped in billions of devices such as routers, wireless access points, all Android devices and mainframes. It consists of more than 36 million lines of code and has 19,000 contributing developers.

In this study, we decided to select the Linux kernel as our case study for various

reasons. The primary reason is that the Linux kernel has to deal with many security aspects. According to CVE, the Linux kernel is the software system with the third highest number of reported vulnerabilities (2357 vulnerabilities) after Debian Linux and Android [38]. Furthermore, what makes the Linux kernel a perfect candidate for our study is the well organized community behind it which facilitates getting relevant and reliable information on vulnerabilities.

Git is the most widely used modern version control system in the world that was created by Linus Torvalds in 2005 for development of the Linux kernel. In addition to the points mentioned above, the stability of the version control system in the Linux kernel provides us the access to more than a decade of history, which is used in this study to extract vulnerable files from different releases.

2.1.4 RCLUV and Related Definitions

Language Features

In this study, we consider each nonterminal in the programming language grammar, a feature of that language. For instance, we consider `additive_operator`, `if_statement`, and `for_statement` that are nonterminals in the C programming language, as features of the language.

Contextualized Language Features

Contextualized features of a language are combinations of the language features being used in the context of each other. For example, `switch_statement` inside `loop_statement` or `if_statement` inside `if_statement` inside `if_statement` are two contextualized features of C programming language. As these

two examples show, we can have multiple levels of contextualization. In this case, `switch_statement_loop_statement` is an example of level 1 and `if_statement_if_statement_if_statement` is level 2. The number of contextualized language features of a programming language depends on the level of contextualization we define.

Rich Contextualized Language Use Vectors (RCLUVs)

RCLUVs [39] are ordered vectors of language features and contextualized language features that encode the use frequency of each one in a piece of code as magnitudes. In other words, an RCLUV is an ordered array of integers in which each index corresponds to a language feature or a feature in the context of other features and the value is the feature count.

This concept is further explained and clarified using examples in Section 3.3

2.1.5 TXL-Based Language Use Studies

TXL [6] [7] is a rule-based language that was developed by J.R. Cordy and C.D. Halpern in the early 1980s. It is designed to support software source code analysis and source transformation tasks. TXL has proven to be an excellent choice for constructing language parsers which enable researchers to efficiently extract information on language feature use. TXL has been widely used in research applications in both industry and academia. We provide a brief overview of recent applications of TXL in language feature analysis below.

Luk et al. [20] conducted a language featrue analysis experiment on 20 open source PHP applications consisting of 51,871 PHP files. A precise parser written in TXL

was created by the authors to parse all of the PHP files in the dataset and count the use of PHP language features in them. In their study, three important research questions were explored:

1. Analysis of which PHP features are actually used in open source applications
2. Analysis of which PHP features are most frequently used in the selected applications
3. Analysis of the differences in the ways that different PHP applications use the features of the language

Shahjamali et al. [32] did a similar study on 51 open source Java applications with the aim of examining how Java language features are actually used by programmers. A parser was constructed in TXL to parse all open source Java projects and an extractor was written in TXL to extract and count the Java features. The results of this study can benefit language designers by suggesting how to evolve Java by concentrating on improving the most frequently used features, removing the most rarely used features, and finding opportunities to improve Java public libraries.

Vishwambar et al. [39] introduces the idea of Rich Contextualized Language Use Vector (RCLUV)-based code profiles. He automated the process of generating TXL language use extractors from a given language grammar. The extractors are used to extract a code profile containing the frequency of the programming language features being used from a program source code.

Rahman et al. [27] recently conducted a research on software defect prediction using RCLUV-based code profiles to train machine learning models. In this study, the authors utilized two benchmark bug datasets and compared the performance of their

models to traditional metrics-based approaches. The learning curves of the models created in their study depict the effectiveness of using RCLUV-based code profiles in defect prediction. In addition, the evaluation and comparison of performance metrics reveal that the RCLUV code profile-based approach outperforms traditional metrics-based approaches in terms of accuracy, precision, recall and f-score.

2.1.6 Vulnerability Discovery

Considering the critical importance of the software vulnerability analysis and discovery, a large amount of effort has been investigated by practitioners both in the academic community and the software industry in order to suggest and develop automatic vulnerability prediction techniques. Nowadays researchers are focusing on developing new approaches with the aim of outperforming the previous ones. Vulnerability discovery approaches can be classified in three categories: static analysis, dynamic analysis and hybrid analysis. We will discuss each category in the following subsections.

Static Analysis

Static Analysis involves direct analysis of the software program with no dynamic execution of the source code. The approaches under this category utilize a diverse abstraction to analyze the programs. Also, they are likely to find a high number of vulnerabilities but have the default of often raising too many false positives meaning that false vulnerabilities may be reported. Static code analysis have some advantages and disadvantages over other approaches for instance defects can be detected in the early development life cycle, reducing huge amount of cost. On the other hand,

automated static analysis tools cannot pinpoint weak points that may create troubles in runtime.

Dynamic Analysis

Dynamic Analysis involves executing a program with specific input data in order to monitor its runtime behavior. It supports analysis of applications without requiring the actual code. The resource consumption in this analysis technique is more as compared to static analysis. Dynamic program analysis tools may require loading of special libraries or even recompilation of program source code. The approaches under this category cannot guarantee the full test coverage of the source code since they utilize a finite number of input test cases to validate a program. Therefore, there is a possibility to miss some vulnerabilities that reside in unseen program states. Nevertheless, dynamic analysis approaches are greatly used in the software industry.

Hybrid Analysis

Hybrid Analysis combines both static source code analysis and dynamic analysis during application runtime. This type of analysis can integrate runtime data extracted from dynamic analysis into a static analysis algorithm to detect vulnerabilities in the applications. Although the combination of static and dynamic analysis can increase the accuracy of the detection rate, such techniques often suffer from the limitations of both approaches.

2.1.7 Machine Learning and Data Mining Techniques

The advent of machine learning (ML) has revolutionized the technology industries. Techniques based on machine learning have been applied successfully in diverse fields ranging from pattern recognition, computer vision, finance, entertainment, and computational biology to medical applications. Software vulnerability prediction is not an exception. ML in this context refers to techniques building complex models based on learning from the available data that can be used to make predictions. ML techniques can be divided in three categories: supervised learning, unsupervised learning and reinforcement learning.

Supervised Learning

Supervised learning refers to training the model on a previously labelled dataset. The dataset in this case consists of pairs of an input object (feature variable) and a desired output value. A supervised learning algorithm analyzes the training data and learns a function from the relationship between the provided feature variables and the target labels. The goal of the supervised algorithms is to predict the actual label of a sample given to them by utilizing the inferred function learned during the training time. Supervised learning problems can be grouped into *regression* and *classification* problems.

Unsupervised Learning

Unsupervised learning refers to training a model on an unlabeled dataset, meaning that the dataset only consists of input objects without any labels. The goal of the unsupervised algorithms is to explore the training data and find a structure or a

hidden pattern within it. Clustering is a typical unsupervised learning task.

Reinforcement learning

Reinforcement learning refers to taining the model to make sequence of decisions in a dynamic and potentially complex environment with a system of rewards and penalties. The goal is to maximize the total reward. Reinforcement learning has several applications ranging from robotics to health care and finance.

2.2 Related Work

In this section, after categorizing the vulnerability prediction studies, we present a review of the well-known related work in this area based on software metrics and leveraging machine learning techniques.

2.2.1 Vulnerability Prediction Studies

According to Ghaffarian et al., [11], in field of software vulnerability analysis and discovery, most of the approaches that use data mining and machine learning techniques can be classified into three main categories: anomaly detection approaches, vulnerable code pattern recognition and vulnerability prediction models based on software metrics.

Anomaly detection approaches

This category of methods utilize an unsupervised learning approach to automatically extract a model of normality from the source code and discover vulnerabilities as an unexpected behavior.

Vulnerable code pattern recognition

This category of work analyzes and extracts features from large samples of vulnerable code, and detects vulnerabilities using pattern matching.

Vulnerability prediction models based on software metrics

This category uses supervised learning approaches to build prediction models based on common software engineering metrics. These models are exploited to predict vulnerable software artifacts. Various metrics have been used for vulnerability prediction in the literature for example, *code-churn*, *code complexity*, *developer-activity* and *information flow*. While also a static analysis technique, this these falls in this category.

Shin et al. [33] [35] investigated whether complexity metrics can be effective to predict the location of security vulnerabilities and whether there are significant differences between vulnerable and faulty code in terms of code complexity. This study initiates all studies that use complexity metrics to build vulnerability prediction models (VPMs). The authors performed statistical analysis on nine code complexity metrics from the JavaScript Engine in the Mozilla application framework. The initial results demonstrated that there is a weak correlation between complexity metrics and vulnerabilities in the Mozilla JavaScript engine. They also found that vulnerable code tend to be more complex than the faulty ones.

Shin et al. [34] also performed a more extensive study on whether complexity, code-churn, and developer-activity metrics can be used for vulnerability prediction and are effective enough to detect vulnerable code locations. They performed empirical case studies on Mozilla Firefox web browser and Red Hat Enterprise Linux kernel. Their

results indicate that the metrics used in this study are discriminative and predictive of vulnerabilities to some extent. For Mozilla Firefox, the recall were over 70% for most of the metrics but the precision was very low (below 5%) for all the metrics. For Red Hat, the recall in the best case when using only complexity metrics was around 80% while the precision was still under 5%.

Chowdhury [4] in his thesis conducted an extensive case study on 52 releases of Mozilla Firefox to examine the relationship between the vulnerabilities and complexity, coupling, and cohesion metrics. The published results suggest that complexity and coupling metrics are strongly correlated with vulnerability while cohesion metrics have a medium correlation. Additionally, they diagnosed the same correlation to be stable across multiple releases of Firefox. In order to test the effectiveness of the selected metrics in vulnerability prediction on 52 releases of Firefox, they utilized four data mining approaches: Decision Trees, Logistic Regression, Random Forests and Naive-Bayes. The results showed that the algorithms based on decision trees outperforms the others by achieving about 75% recall and 28% of false positive rate. Furthermore, they performed the next release analysis on the Firefox versions using 32 releases as training set and 20 releases as the test set. The results depict an increase in the false positive rate and a decrease in the recall.

In 2010, Zimmermann et al [45] performed an empirical study on Windows Vista with the aim of finding out whether classical metrics like complexity, churn, coverage, dependency measures, and organizational structure of the company, are efficient metrics for predicting vulnerabilities or not. The results suggested that most of the metrics predict vulnerabilities with an average to good precision; however the recall is very low. The authors also measured the Spearman rank correlation [37] between the

metrics previously described and the number of vulnerabilities. Most of the metrics turned to have very low positive correlation with the vulnerabilities. Moreover, they identified that the files with frequent changes done by a great number of engineers are more prone to vulnerabilities. The other conclusion of this paper is that the number of lines of code and the complexity of a file have a direct relationship with the number of vulnerabilities in that file.

In 2012, Hovsepyan et al., [14] presented a preliminary study on vulnerability prediction based on text analysis techniques. In this study, 19 versions of the K9 mail client application have been selected. The authors suggested an approach to tokenize the raw source code and create bag of words representation of the code. The bag of words models, are used as features to train machine learning models. They used the first version of the K9 mail client application as their training set and test their models on the next releases. Their prediction models turned to achieve encouraging results with an average precision of 85% and recall of 88% . However, there are some concerns regarding the reported results since a static code analysis tool called *Fortify* was used for building the dataset and debates existed among these tools about the imprecise results that these tools might produce.

In 2016, Younis et al. [42] explored a new aspect of vulnerability prediction. They attempted to identify the attributes of vulnerable code that are more likely to be exploitable. In order to serve this purpose, a case study was established on 183 vulnerabilities extracted from NVD for Linux kernel and Apache HTTP server that only 82 of them were found to have an exploit. Eight metrics were used to inspect the existence of a vulnerability exploit from all the vulnerabilities. The results suggest that using the software metrics, the differences between a vulnerability that has no

exploit and a vulnerability that has an exploit can be characterized to some extent. However, they mentioned that recognizing the vulnerability exploitability is more complicated than predicting the presence of vulnerabilities.

A great many approaches have been proposed for vulnerability prediction, but each of them uses a custom dataset from various contexts and performs different data preprocessing techniques. Therefore, comparing these approaches to each other only based on the accuracy metrics is not reasonable. Jimenez et al. [15] made a reliable replication and comparison of the main vulnerability prediction models based on *software metrics*, *text mining*, *includes* and *function calls*. They created a large dataset containing all vulnerable files of Linux kernel from 2005 to 2016. Moreover, they assessed the power of vulnerability prediction models in distinguishing vulnerable and non-vulnerable files. The results depict that the models based on *text mining* outperforms the other approaches and code metrics related models perform poorly. It should be noted that we created our dataset similarly to the dataset used in this research in order to compare the results of our experiments with the four different vulnerability prediction models implemented and tested in this study. The details of how we built our dataset and designed our experiments are discussed in the following chapters.

2.3 Summary

In this chapter, we focused on introducing readers to the background needed to understand our work; and also presented an overview of the well-known related work in the vulnerability prediction area.

In the following chapter, we will briefly go over the goal of this thesis as a reminder

and discuss our process of data selection and feature extraction.

Chapter 3

Data Selection and Extraction

As outlined in Chapter 1, the goal of this thesis is to investigate the practicality of using RCLUVs and machine learning models to predict vulnerabilities in software systems. In the previous chapter, we covered some background information and discussed related work in this area. We believe the first step of exploring our research questions is to construct a solid database that can be used to simulate both experimental and real-life setups and has enough data to train and validate different machine learning models.

In this chapter, we present our approach for extracting vulnerable and neutral files from the Linux kernel. Afterwards, we describe the RCLUV extractor process flow and how vector-based code-profiles were generated from the selected vulnerable and neutral files.

An extensive search of the vulnerability prediction literature revealed that there is no standard dataset for studies on vulnerability prediction models. Although some authors have published their datasets [26] [41] [44] [12], they have not updated their datasets after the first release. On the other hand, some of the datasets contain only the metrics extracted from files and not the actual source files. Since, in this study,

we need to have access to the source code of the files, these datasets are not good choices for our work. Therefore we chose to construct our own large dataset from the original Linux vulnerability database.

Previous research in vulnerability prediction has shown that the models built on cross-projects are mostly less effective than the project specific ones [31]. This study lies in the context of the Linux kernel. We decided to select the Linux kernel as our case study for various reasons including the open source nature of Linux and the great number of vulnerabilities reported in the *National Vulnerability Database (NVD)* [18] for the Linux kernel.

The majority of previous research, i.e. [23], [33], [31] attempted to detect vulnerabilities in the source code files. We also make our evaluation at the file granularity level based on the fact that a file is a conceptual unit of development consists of a group of related entities such as functions, data types, etc. Furthermore, using a file as a logical level of analysis has been approved to be the most effective and feasible level of granularity [22].

3.1 Vulnerable File Extraction

In order to extract vulnerable files from the Linux kernel, we utilized the XML vulnerability data feed offered by NVD. This data feed contains a collection of all the vulnerabilities reported each year. We focused on the Linux kernel vulnerabilities that happened from 2005 (the date of adoption of git) to early 2019. First,the Linux kernel repository was cloned in a local folder and all the XML feeds were downloaded from NVD website. Then, the XML feeds were parsed and all vulnerabilities reported over the history were retrieved using our own modified version of Data7 tool [16]. Each

extracted vulnerability contains information including Common Vulnerabilities and Exposures Identifier (CVE ID), the affected versions, vulnerability severity, and reference to one or more commits that fixed the vulnerability. Figure 3.1 is a screenshot of the retrieved XML file containing information about the Linux vulnerabilities. We checked out the previous commit for each commit in the CVE and marked the files modified in that commit as vulnerable files. It should be noted that we only focused on the files with `.c` extension. This left us with 2282 vulnerable files accounting for 1345 vulnerabilities selected from different versions of the Linux kernel.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<data7 last_updated="2019-07-08 16:24:15 CEST" project="linux_kernel">
  <cve id="CVE-2011-3209" last_modified="1349262">
    <cwe>189</cwe>
    <score>4.9</score>
    <description>The div_long_long_rem implementation in include/asm-x86/div64.h in the Linux kernel before 2.6.26 on
    the x86 platform allows local users to cause a denial of service (Divide Error Fault and panic) via a
    clock_gettime system call.</description>
    <affectedVersions>
      <version>2.6.25.20</version>
    </affectedVersions>
    <bugs/>
    <patches>
      <linux_kernel>
        <commit hash="f8bd2258e2d520dff28c855658bd24bdafb5102d" timestamp="1209654238">
          <files>
            <file path="arch/mips/kernel/binfmt_elfn32.c"/>
            <file path="arch/mips/kernel/binfmt_elfo32.c"/>
            <file path="drivers/char/mmtimer.c"/>
            <file path="include/asm-x86/div64.h"/>
            <file path="include/linux/jiffies.h"/>
            <file path="kernel/posix-cpu-timers.c"/>
            <file path="kernel/time.c"/>
            <file path="kernel/time/ntp.c"/>
            <file path="mm/slub.c"/>
          </files>
        </commit>
      </linux_kernel>
    </patches>
  </cve>
```


Figure 3.1: A screenshot of the XML file extracted by Data7 tool containing information about the Linux kernel vulnerabilities

3.2 Neutral File Extraction

Once all vulnerable files had been gathered, the next step was to select a set of neutral files. In this study, we defined a file to be neutral if it has never been declared as a vulnerable file. We call these files neutral files, rather than non-vulnerable files, because they might contain unknown vulnerabilities that have not been detected yet.

Research suggests that approximately 3% of the files in the Linux kernel have the history of being vulnerable in at least one release [15]. In order to create a realistic dataset, for each vulnerable file, we randomly selected 32 neutral files from the same point of time with the intention of reflecting the actual ratios of vulnerable and neutral files in Linux. This resulted in the creation of a highly imbalanced set of files with 75,000 files in total.

The following section describes our code profiles, and how we extracted them from the selected vulnerable and neutral files.

3.3 Code Profile (Rich Contextualized Language Use Vectors)

In this study, we use RCLUVs as the code profiles of our dataset. Combining basic and contexualized language features together creates RCLUVs.

The features in our code profiles are derived directly from the formal grammar of the programming language. The identification of RCLUV features involves no subjectivity and no uncertainty because they correspond to syntactic features derived directly from the programming language grammar.

A program's code profile encodes how the program uses basic and contextualized features in its source code. Figure 3.1 shows an example of a C program that contains a simple function returning the maximum value in an array. Listing 3.2 shows a

snippet of the program's RCLUV code profile generated by our feature extractor.

```
int max(int array[], int size) {
    if (size==0)
        return 1;
    int max_value=array[0];
    for (int i=1; i<size; i++) {
        if (max_value<array[i])
            max_value=array[i];
    }
    return max_value;
}
```

Listing 3.1: An example of C program

```
...
function_definition 1
type_specifier 7
if_statement 2
for_statement 1
return_statement 2
integer_constant 4
for_statement if_statement 1
function_definition type_specifier 7
function_definition if_statement 2
parameter_declaration type_specifier 4
compound_statement if_statement 3
compound_statement expression_statement 2
...
```

Listing 3.2: A part of the RCLUV vector for Listing 3.1

3.4 Process Flow of the RCLUV Feature Extractor

The extractor that we designed in our work is a refinement of a recent contribution by Vishwambar [39], using the same idea for extracting the features of the language. Vishwambar et al. automated the process of generating a new extractor for each programming language. However, we designed a generic feature extractor for any language by using TXL polymorphism as long as the TXL grammar for the language is available. In order to use the generic extractor, first we needed a TXL grammar for the language that correctly identifies all the syntactic language features of the language (list of nonterminals). Having the syntactic features of the grammar and the TXL grammar for the language enables the extractor to parse, count the language feature use and produce the code-profile of the input program. Figure 3.2 shows the general structure and abstract view of the RCLUV feature extractor. In the following paragraphs we will describe this process in detail.

Our generic feature extractor takes a list of nonterminals in addition to the TXL language grammar. The nonterminal list is generated automatically from the TXL

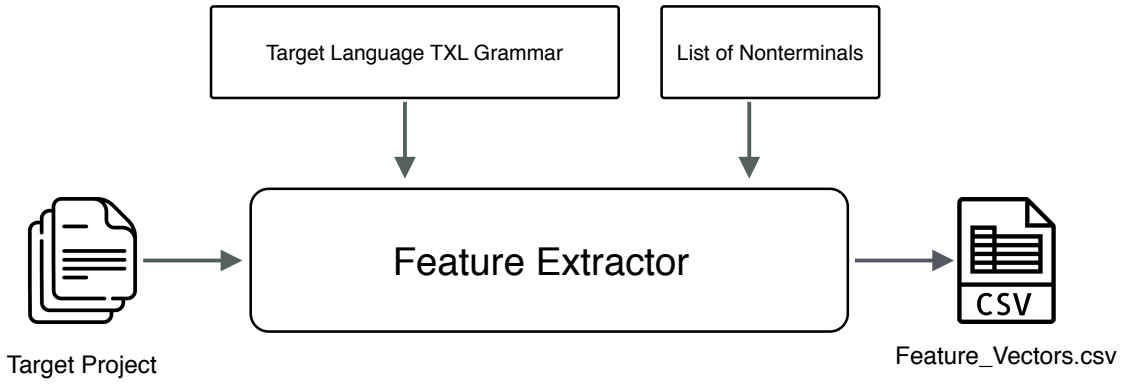

Figure 3.2: An overview of the proposed generic feature extractor

grammar of the language using a shell script that extracts all the nonterminal names from the grammar. Then, a filtering process is applied to the list in order to remove the redundant nonterminals which always have the same count as another or whose frequency can be derived from other features. To illustrate the point, consider Listing 3.3 which shows a part of C programming language TXL grammar. Running the shell script identifies the `if_statement` and `structured_statement` as nonterminals. The filtering process, then, filters `structured_statement` out since its frequency can be derived from the sum of nonterminals inside it.

Having a TXL grammar and a list of non-redundant nonterminals, our extractor can take a target project and generate the RCLUVs. The first step is to parse the target project and construct a parse tree using TXL and the language grammar. Afterward, the extractor loops over the list of nonterminals and counts the number of nodes in the parse tree that are of the type of each nonterminal. We call the result of

```
define if_statement
        'if '( [condition]
            ') [sub_statement]
        [opt else_statement]
end define

define structured_statement
        [if_statement]
    |   [for_statement]
    |   [while_statement]
    |   [switch_statement]
    |   [do_statement]
    |   [compound_statement]
    |   [asm_statement]
end define
```

Listing 3.3: A part of C TXL grammar

this step *level 1* LUVs. For instance, consider {a, b, c} as the "useful" nonterminals used in the target programming language. In this case, the LUVs depict how many times feature a, b and c were used in the project individually.

Although *level 1* LUVs contain useful information, we are interested in taking them one step further and obtaining more in-depth information from the parse tree. In order to serve this purpose, our feature extractor also draws the frequency of the usage of the features in the context of other features. This requires knowing which nonterminals can potentially be used inside each nonterminal. This information is generated automatically from the TXL grammar for the language using an expander program that analyzes the grammar to find all the nonterminals which can possibly be contained in each nonterminal. The results are then filtered to remove all the contained nonterminals that are not in the "useful" nonterminal list. These concepts are further explained in the following example.

Continuing on the previous example, the output of the expander program before filtering is shown in Listing 3.4. In this example, `a = a,b,c,d` means that the nonterminal a can contain notnterminals a,b,c, and d.

The output of the expander program is then filtered in order to remove the redundant features. This process removes the redundant nonterminals from both sides of equal sign. In this example, since the nonterminal d is considered redundant, the output of the filtering process will look like Listing 3.5.

```
a = a , b , d , c
b = d , c
c = c
d = c
```

Listing 3.4: Expander program output before filtering

```
a = a , b , c
b = c
c = c
```

Listing 3.5: Output of the filtering process

In this example, the contextualized features that the extractor will be looking for are: `a-a`, `a-b`, `a-c`, `b-c` and `c-c`.

In order to calculate the frequency of the usage of `a-b`, for example, the extractor detects all nodes of type `a`, and then it searches in the subtrees of those nodes, trying to find and count children of type `b`. Therefore, the count value of feature `a-b` represents how many times feature `b` was used inside feature `a` in the target file. The value count for all the other contextualized features will be calculated similarly.

The output of the feature extractor is a comma-separated values (CSV) spreadsheet which is called the code profile of the target project. Each row in the CSV file represents a source file in the dataset, and each column shows the frequency of the usage of a basic or contextualized language feature.

3.5 The Linux Kernel Feature Extraction

3.5.1 C TXL Grammar

In order to be able to parse an input program using TXL, the TXL grammar for the programming language that the program is written in is required. The Linux kernel is written primarily in C and as mentioned before, we considered and analyzed only files named with the `.c` extension in our work. We created a C TXL grammar semi-automatically from the published reference grammar in the language documentation.

Appendix A: Language Syntax Summary, from the draft ISO standard for C18 [1] was used as the reference to the C language documentation. An initial grammar for C was built by a semi-automatic extraction from the reference document, and then tested and hand-tuned to handle a wide selection of large open source C applications such as the Linux kernel. Figure 3.3 shows an excerpt of the resulting C TXL grammar.

```
% Comments are /* */ and // to end of line
comments
    //
    /* */
end comments

define program
    [translation_unit]
end define

define translation_unit
   [repeat function_definition_or_declaration]
end define

define function_definition_or_declaration
   [function_definition]
    |  [declaration]
    |  [compound_statement] [NL]    |
[unknown_declaration_or_statement]
end define

define function_definition
    [NL] [function_header]
    [opt KP_parameter_decls]
    [compound_statement] [NL]
end define

define function_header
    [opt declaration_specifiers] [declarator]
[function_declarator_extension]
end define
```

Figure 3.3: An excerpt of the C TXL grammar

[1] ISO standard ISO/IEC 9899:2017

3.5.2 Nonterminals of the C Language

The C programming language has 257 nonterminals extracted from the C grammar, 158 of which are redundant based on our filtering criteria. Adding the contextualized features to the nonterminal features resulted in 1,047 features in total.

3.5.3 Code profile of the Linux Kernel

We ran the automatic RCLUV generator on two separate folders of vulnerable and neutral files of the Linux kernel. As a result, two datasets in the form of csv files each containing 1,047 columns were created. We added a vulnerability label to each dataset and concatenated them. The final dataset contains only numeric (integer) features for approximately 75,000 files in total, selected from the different stages of the history of Linux.

3.6 Summary

In this chapter, we described our data selection and extraction procedures. In addition, we explained the code profile used in this research and how we generated it from the selected source files. In the following chapter, we will examine whether RCLUV-based code profiles are effective features for training machine learning models to predict security vulnerabilities or not.

Chapter 4

How effective are machine learning models trained on RCLUVs in distinguishing vulnerable files from neutral files?

The previous chapter described how RCLUVs are generated by analyzing the parse tree of source code, and how we created our dataset. This chapter investigates whether RCLUV-based code profiles can be used as features to train machine learning models to distinguish vulnerable files from neutral files and, if so, how effective they are. In order to serve this purpose, several machine learning models were trained using RCLUV code profile and the learning curve of the models were analyzed. Additionally, the performance of the models was evaluated by calculating the precision, recall and f1-score to assess the ability of RCLUVs in vulnerability prediction.

4.1 Introduction

In order to train a supervised machine learning algorithm, a feature vector is required. During the training process, an algorithm learns the relationship between the feature and the target variables which results in the creation of a model. Using that model,

unseen test set can be predicted in the next phase.

Before we begin discussing how we explored RQ1, it is important to mention that using RCLUVs might pose some potential risk on the performance of machine learning algorithms. For instance, the number of elements in an RCLUV can easily exceed thousands depending on the number of features of the target programming language and the level of feature contextualization. The large number of features can contribute to creation of very complex machine learning models which results in overfitting. For example, in our experiments, each sample has 1,047 elements in its RCLUV. In addition, RCLUVs contain redundant information by nature. They may contain very correlated features, or features that have no correlation with the target label. These challenges might affect the machine learning algorithms' ability to discover the true relationship between the feature vectors, and the target labels.

In the next section we discuss our study design to make sure our models are not getting affected by the challenges of using RCLUVs and to answer our first research question:

- RQ1. How effective are machine learning models trained on RCLUVs in distinguishing vulnerable files from neutral files?

4.2 Study Design

To determine the effectiveness of RCLUVs in vulnerability prediction, we designed a study to measure and compare the performance of different machine learning models in detecting vulnerable files by analyzing their learning curves.

4.2.1 Dataset

In Chapter 3, we described our approach for creating our main dataset of approximately 75,000 files selected from the different stages of the history of Linux. For each file, we extracted 1,045 numeric features. The dataset is designed to determine if the models are able to mark vulnerable files in a realistic setting, i.e., in an environment where vulnerable files would be uncommon. Therefore, the ratio of vulnerable files to neutral files in our main dataset (that is 3:97) reflects the actual ratios in Linux.

The problem with the imbalanced datasets is that because of the dominance of the majority class, the minority class often will be ignored as noise and therefore the classifiers predict the majority class far more often. In order to put more weight on the minority class cases, there are numerous methods which address the imbalanced classification problem, such as over-sampling the minority class and under-sampling the majority class.

In the experiments of this chapter, we decided to use a balanced training set that has the same number of vulnerable and neutral files (1:1 ratio). This decision was made because the purpose of this research question is to understand how effective RCLVs are in distinguishing vulnerable and neutral files in general, therefore, keeping the same portions as the Linux kernel is not desirable and might affect the conclusion.

4.2.2 Machine Learning Algorithms

Four Different machine learning algorithms have been used in this study:

- eXtreme Gradient Boosting (XGBoost)

  XGBoost is an implementation of the Gradient Boosted Decision Trees(GBT) algorithm that is widely used by data scientists and provides state-of-the-art

results on many machine learning competitions [3]. In GBT, trees are grown sequentially, with each successive one grown to correct the errors contained in the previous trees using gradient descent for optimizing the loss function. The algorithm is highly effective in reducing the computing time and provides optimal use of memory resources [3].

- Random Forest (RF)

  The random forest technique is an ensemble method which consists of multiple unpruned decision trees trained on different parts of the same training set to obtain better prediction performance. Each tree provides a classification for the input data, then the classification results are aggregated and the majority voted prediction is chosen as the outcome.

  Dittman et al. [9] in their work inferred that random forest is relatively robust in terms of handling imbalanced data. This finding was one of the primary motivations behind deciding to choose the RF algorithm in our work.

- Multi-Layer Perceptron Neural Networks (Deep Learning)

  A multi-layer perceptron (MLP) is a feed-forward supervised neural network containing an input layer, an output layer, and one or more hidden layers. MLP is often used for classification, utilizing a supervised learning technique called backpropagation for training [28].

- Binary Logistic Regression (LR)

  LR is one of the most common learning algorithms according to *Hands-on machine learning with Scikit-Learn and TensorFlow* [13]. LR is a case of Generalized Linear Model that uses a logistic function to model the probabilities for classification problems.

4.2.3 Evaluation Metrics

Vulnerability prediction is a binary classification problem. The Confusion Matrix is a technique for summarizing the performance of a classification algorithm that, in our case, consists of two columns and two rows that together represent: true positives (TPs), true negatives (TNs), false positives (FPs) and false negatives (FNs).

TP True Positive: Number of files correctly classified as vulnerable.

FP False Positive: Number of neutral files incorrectly classified as vulnerable.

TN True Negative: Number of files correctly classified as neutral.

FN False Negative: Number of vulnerable files incorrectly classified as neutral.

There are several performance measures which can be directly computed from the confusion matrix including *Accuracy*, *Precision*, *Recall* and *F1-Score*. In this study, we use the following terminology:

- **Accuracy**: Represents the probability that the model correctly predicts the observations. When dealing with a severe class imbalance, Accuracy does not precisely demonstrate the performance of a model.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) \tag{4.1}$$

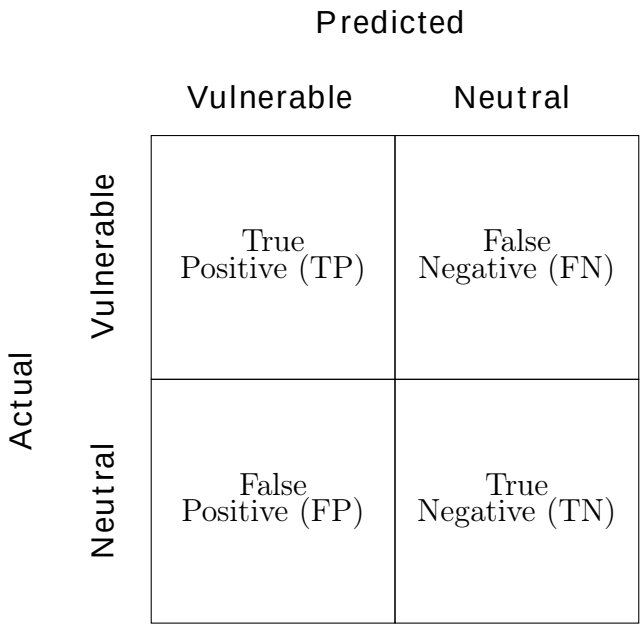

Figure 4.1: Confusion Matrix

- Precision: Represents the probability that a file that is classified as vulnerable is indeed a vulnerable one.

$$Precision = TP/(TP + FP) \tag{4.2}$$

- Recall: Represents the probability that a vulnerable file will be classified as vulnerable by our classifier.

$$Recall = TP/(TP + FN) \tag{4.3}$$

- F1-score: Represents the harmonic mean of the precision and recall. This metric gives a better measure of the incorrectly classified cases than the Accuracy

metric.

$$F1 = 2 \quad \frac{precision \quad recall}{precision + recall} \tag{4.4}$$

- MCC (Matthews Correlation Coe□cient): Represents the binary classification quality by calculating the correlation coefficient between the actual and predicted class of an observation. MCC takes all the elements of the confusion matrix into account and returns a coefficient between -1 and +1, a score of 1 demonstrates an ideal prediction whereas a score of -1 shows that all data were wrongly predicted. A coefficient of 0 shows that the classifier performed very similar to random guessing.

$$MCC = \frac{TP \quad TN \quad FP \quad FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.5}$$

4.3 Experiments and Results

In our first experiment, we trained four types of machine learning models including XGBoost, random forest, MLP neural networks and logistic regression on our dataset. All machine learning models were developed using Python and the Scikit-learn library. To enhance classification performance, we used 5-fold cross validation in conjunction with Grid-search optimization to find the optimal hyper-parameters for each learning algorithm.

In order to assess the performance of our models, we plotted and analyzed learning curves individually. Learning curves are plots that show changes in the learning performance of training and evaluation data over a varying number of training instances. If the performance of a model on the unseen data improves, then it can be inferred

that the model is learning from the provided feature set. Therefore, the purpose of drawing the learning curves was to get an idea of how well an algorithm learns from RCLUV-based code profiles and whether the estimator suffers from a variance or bias error.

Figure 4.2 shows an example of a learning curve based on the accuracy score. When an algorithm is given a small sample of training data, it can easily fit a model on the training set with a very high training score. As the training set size grows, the training score decreases because it becomes difficult for the algorithm to fit all of the training samples perfectly. However, adding new training instances is very likely to lead to better models that perform better on the validation set. The notable feature of the learning curve is the convergence to a particular score as the number of training samples grows. Once the validation score curve and the training score curve reach a plateau, adding more training data will not help.

Figure 4.2: A sample learning curve

Overfitting and underfitting are two common problems that occur when an algorithm fails to generalize beyond the training set, and can be detected from the learning curve plots. Overfitting happens when the model performs accurately on the training set, but very weakly on the test (or validation) set. This occurs when the model is very complex and there exist too many features. On the other hand, underfitting happens when the model cannot learn a sufficiently complex representation of the data and does not fit the training set well. Therefore, it cannot be generalized to unseen data.

For plotting the learning curves, we used the Scikit-learn library in Python. First, stratified 5-fold cross-validation was performed on the entire dataset meaning that the dataset was split into five folds, each fold having the same ratio of vulnerable and neutral files in it. Each fold was given a chance to be the held-out validation set and four other folds were assigned as the training sets resulting in five different experiments (Figure 4.3). For each experiment, eleven different subsets of the training set (1%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%) were used to train a model and for each subset, several performance metrics were computed. Afterward, the scores were averaged over all five runs for each training set size. Figure 4.4 demonstrates our approach to construct vulnerability prediction models for the purpose of plotting the learning curves.

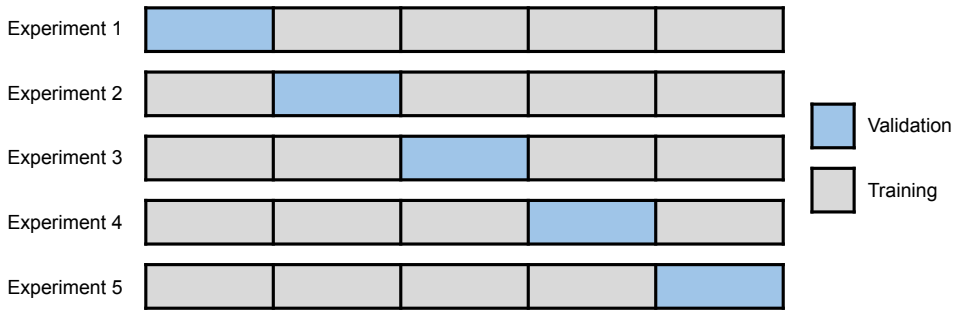

Figure 4.3: Training and validation split for each experiment

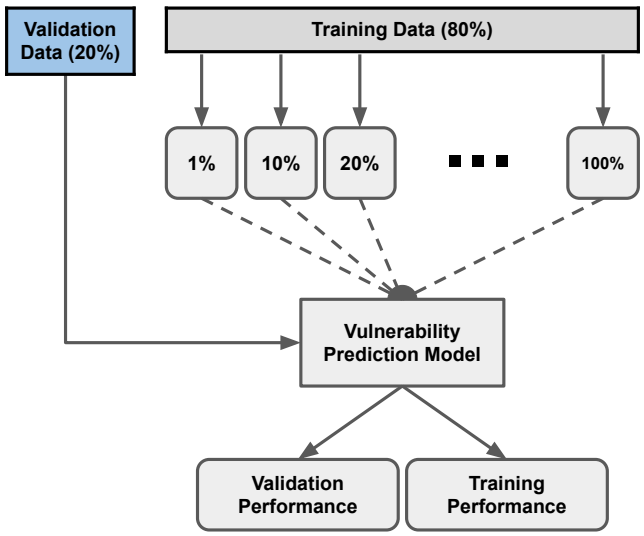

Figure 4.4: Overview of the approach for creating learning curves in each experiment

4.3.1 Analysis of the Learning Curves

The learning curves of four different machine learning models trained on the Linux dataset are shown in Figures 4.5, 4.6, 4.7 and 4.8. All of the curves reveal a similar pattern. The training and validation score curves started with a large gap at the

beginning. As the number of training samples increases, all the training score curves show a downward trend while the validation curves show an upward trend over time. Evidently, the models are learning from RCLUV-based code profiles and the algorithms are benefiting from the increased amount of training data. The decreasing nature of the gaps between the training and validation score curves suggests that adding more training data will improve the models. Eventually, there exist tiny gaps between the curves in the XGBoost, random forest and MLP models. The gap in the logistic regression model is a little bit larger but it is still narrow, so we can safely conclude that the variance is not high since the validation accuracy is reasonably good and the models are generalizing and learning pretty well.

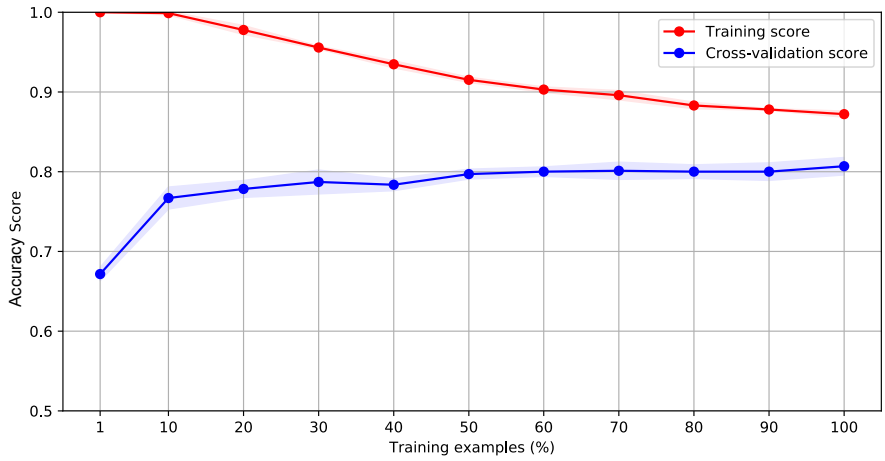

Figure 4.5: Learning curves of XGBoost models

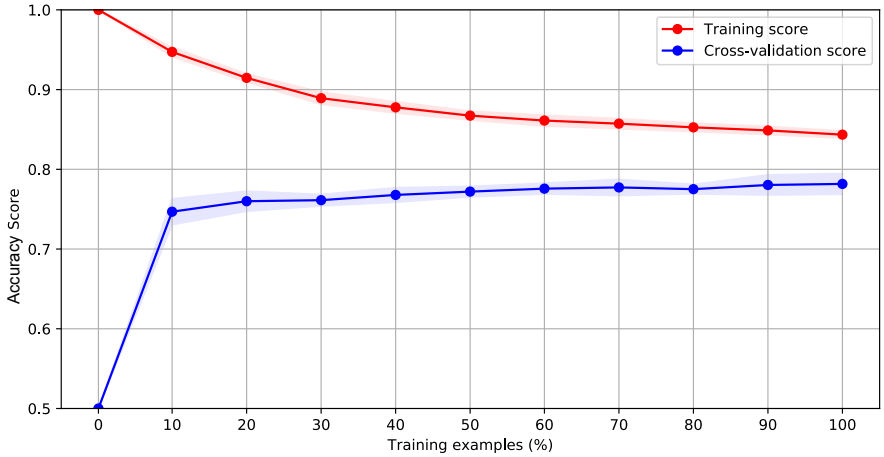

Figure 4.6: Learning curves of Random Forest models

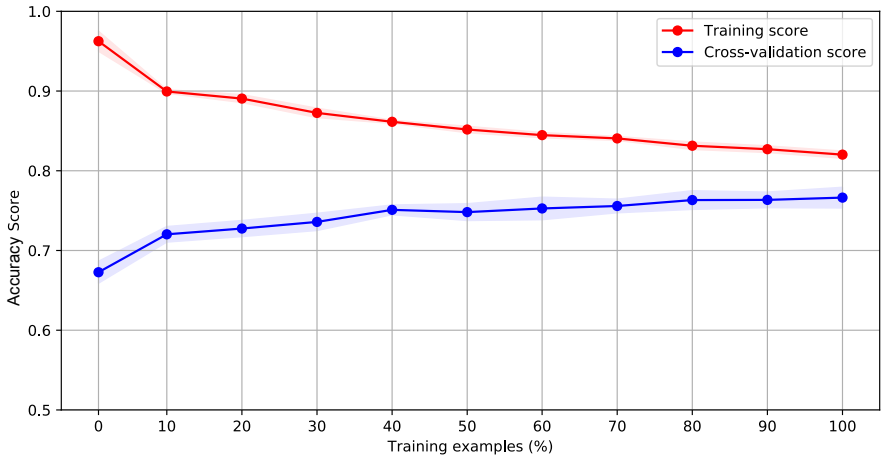

Figure 4.7: Learning curves of Multi-Layer Perceptron Neural Networks models

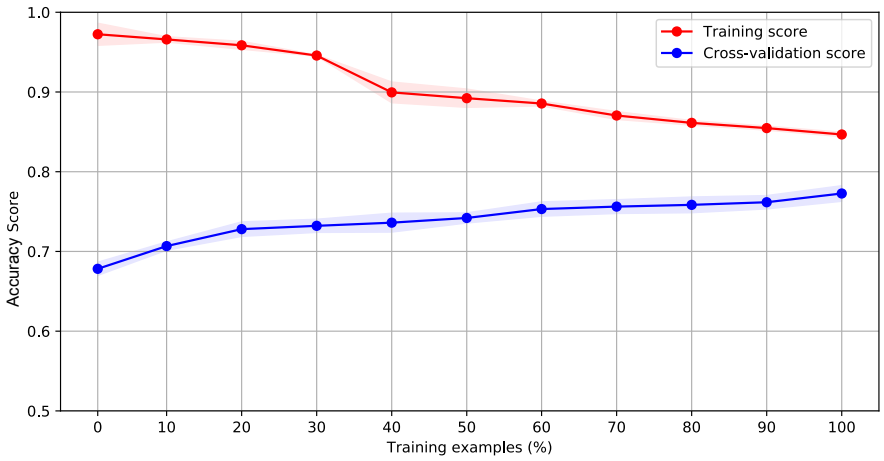

Figure 4.8: Learning curves of Logistic Regression models

4.3.2 Performance Evaluation

In the previous section, we showed that machine learning models can learn the relationship between RCLUV-based code profiles and vulnerabilities based on the results of the learning curves. The question that comes next is how well RCLUV-based models perform in distinguishing vulnerable files from neutral files compared to the existing prediction methods. Therefore, we set up four experiments to evaluate the performance of the RCLUV-based models and compare them with each other. Next, we compared our results with four popular vulnerability prediction methods from the literature.

We examined the efficiency of four machine learning algorithms including XGBoost, random forest, MLP neural networks and logistic regression in these experiments. Finding the best hyperparameters for the models is computationally intensive,

and hence, there is a need to severely limit their search space. In our case, we short-listed hyperparameters manually first and then selected them using the Grid-search optimization technique in scikit-learn Python library. The pattern of the learning curves ensured that the models are a good fit for the given dataset, meaning that they do not overfit or underfit the data although they could benefit from more training examples.

Dataset Splitting

To split the data, we used stratified 5-fold cross-validation and repeated this process 10 times. Using stratified sampling, the dataset is split into 5 folds of equal size (20% of the entire dataset), each has approximately the same ratio of vulnerable and neutral files. Each fold is used as the test set and the remaining folds are used for training. The performances of 5 different prediction models were calculated and averaged. Moreover, we repeated the entire process 10 times meaning that, for each ML algorithm, we built 50 different prediction models in total (10 times 5-fold stratified cross-validation) and calculated the averaged results of the precision, recall, accuracy, MCC and F1-score metrics.

Sampling

As discussed in the previous chapters, the ratio of vulnerable files in Linux is about 3% of the total files. Such a dataset is heavily unbalanced. In this experiment, we used under-sampling to balance the training set with the intention of building more accurate models. With under-sampling the training set, all the vulnerable cases in the dataset are retained, while only a subset of the neutral cases are selected

Table 4.1: Number of vulnerable and neutral files in each experiment

	Vulnerable Files	Neutral files	Total
Training Set	1825	1825	3650
Test Set	456	14519	14975

randomly so that the number of both cases matches each other. The change to the class distribution did not apply to the test set. As a result, all models were evaluated on an imbalanced test set. Table 4.1 shows the number of vulnerable and neutral files in each experiment. On the other hand, since random undersampling can produce a different set of training data each time, as mentioned in the previous section, we repeated the 5-fold cross-validation 10 times. Therefore in each run, a new dataset was generated as the neutral file samples in the training were selected randomly.

Results

Figure 4.9 and Table 4.2 indicate the performance of our models in predicting vulnerabilities of the Linux kernel.

Comparison among the four popular supervised algorithms demonstrated that they perform quite similarly using RCLUVs as the feature set. Little variation can be observed in their performance on the same train-test data. XGBoost models outperform all other models in terms of recall with an average recall of 55%. Random forest models performed better than other models with an average precision of approximately 40%. Surprisingly, MLP models failed to perform as well as expected. This could be due to limited training data since it is suggested that MLP models require a very large amount of training data. Apparently, random forest and XGBoost models achieved slightly better performance results than models based on MLP and logistic regression. In general, all models performed far better than random guessing based

on the computed MCC scores.

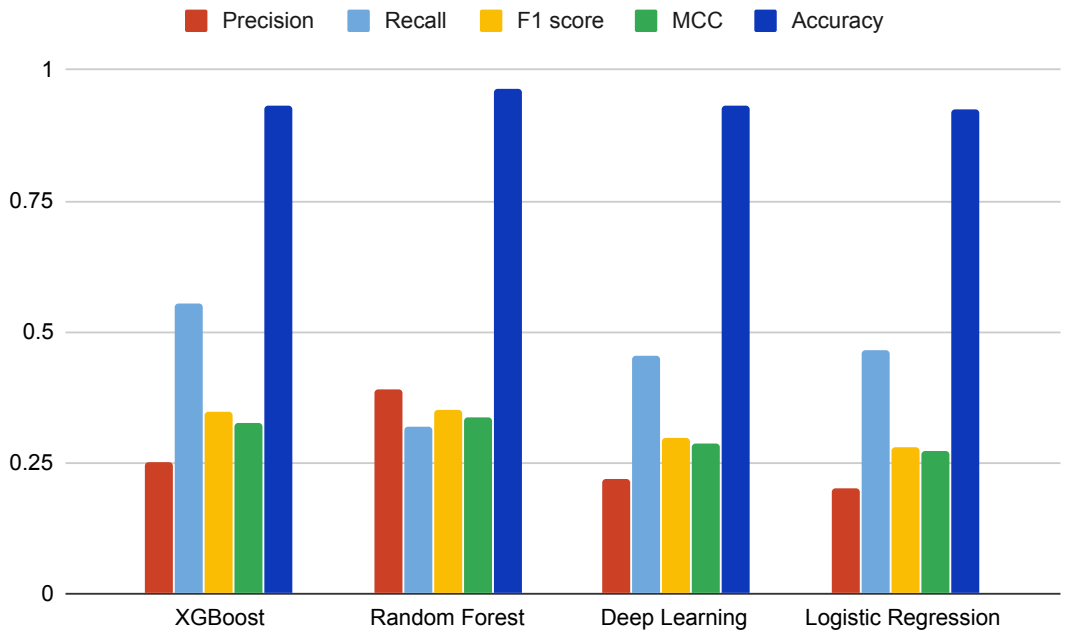

Figure 4.9: Performance results for all machine learning models using RCLUV code profile

Performance Comparison and Discussion

One of the goals of this study is to compare an RCLUV-based vulnerability prediction approach with other existing approaches. We firmly believe that comparing the results of a study with others is not meaningful unless they use the same dataset.

	Precision	Recall	F1-score	MCC	Accuracy
XGBoost	0.252	0.553	0.347	0.325	0.933
Random Forest	0.391	0.318	0.351	0.335	0.965
MLP Neural Networks	0.22	0.454	0.297	0.286	0.932
Logistic Regression	0.202	0.465	0.278	0.271	0.925

Table 4.2: Performance results of models using RCLUV code profile

Unfortunately, there is no standard dataset for vulnerability prediction. Defining a reliable dataset is not an easy task since it heavily relies upon the information that is extracted and considered as the ground truth. Some researchers have built their vulnerability prediction datasets based on different sources of information including static analysis tools such as *Fortify* [14], [31], [29]. Austin et al. [1] found that although static analysis tools have been commonly used as part of the security testing process, they generate a large volume of false positives. In addition to this finding, training a model based on the predictions of a static analysis tool with a questionable performance, in the best case, will result in a model that performs as well as the tool.

Jimenez et al. [15] recently conducted a vulnerability prediction study on the Linux kernel. In their study, they replicated and compared the main vulnerability prediction models that are based on (a) software metrics such as complexity, code churn and developer activity metrics [33], (b) text mining [31], (c) includes and (d) function calls [23]. To be able to compare our results with the results published in their study, we tried to use the same source of information (National Vulnerability Database) and approach to mark the vulnerable and neutral files in our dataset. Also, we kept the same ratio of vulnerable files and neutral files similar to their study. The only difference between the two datasets is that our dataset contains the vulnerabilities reported from 2005 to 2019 (consisting of 2,282 vulnerable files accounting for 1,345 vulnerabilities) while the other dataset was constructed based on the vulnerabilities reported from 2005 to 2016 (consisting of 1,640 vulnerable files accounting for 743 vulnerabilities).

The reported results of their study indicated that an approach based on text mining performed better than the others. Also, they concluded that models based on code

metrics performed poorly. Figure 4.10 and 4.11 demonstrate the clear comparison of the results achieved by our best models trained on RCLUV code profiles and the other four vulnerability prediction models replicated in the above-mentioned study. Although the precisions of our models are not always the best, they are still higher than the approaches based on *includes* and *function calls*. Additionally, in terms of recall, our XGBoost models outperform all other models. Since there is a trade-off between precision and recall values depending on the classifier's threshold, it is not a fair way to compare each of these metrics separately. MCC and F1-score are suitable metrics for this comparison.Fortunately, MCC results have also been reported in the paper that we are comparing our results with. Based on Figure 4.10, RCLUV-based models outperformed the models that were trained on *includes*, *function calls*, and *software metrics*. However, the text mining model still performed the best.

An overview of the limitations of this study, including threats to validity will be discussed in detail in Chapter 6.

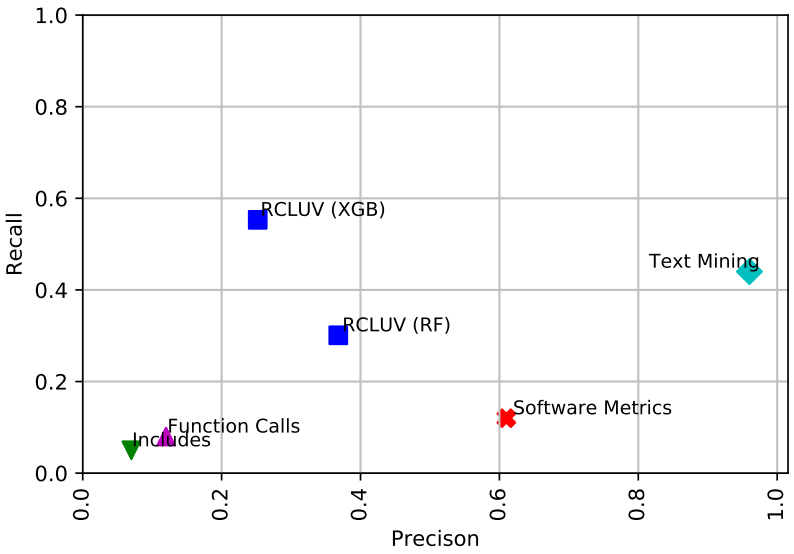

Figure 4.10: Precision-Recall comparison chart

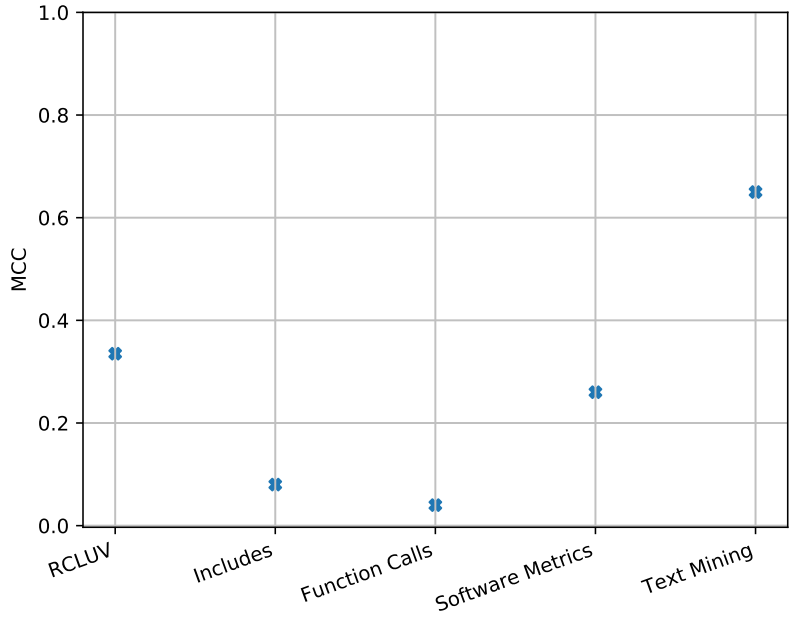

Figure 4.11: MCC comparison chart

4.4 Conclusion

In this chapter, first, we investigated whether RCLUV-based code profiles can be used as features to train machine learning models to distinguish vulnerable files from neutral files. By training machine learning models and analyzing their learning curves, we concluded that the machine learning models were able to learn the patterns of vulnerabilities from RCLUV-based code profiles. Also, we calculated the performance of four different machine learning models trained on RCLUVs and compared them with vulnerability prediction models based on *includes*, *function calls*, *software metrics*, and *text mining*. The overall direction of our results showed that the RCLUV-based models can learn to distinguish vulnerable files from neutral files with a precision close to 40% and MCC of 33%. Also, RCLUV-based models outperformed the models that were trained on *includes*, *function calls*, and *software metrics*. In the next chapter, we will investigate the effectiveness of RCLUVs in predicting future vulnerable files of Linux based on the historical data retrieved from the previous releases.

Chapter 5

How effective are vulnerability prediction models trained on RCLUVs in predicting vulnerable files in future releases?

In the previous chapter, we investigated whether RCLUV-based code profiles are effective features for training machine learning algorithms to distinguish vulnerable files from neutral files, or not. Our results indicated that RCLUV-based code profiles are practical features for detecting vulnerabilities in the Linux kernel to some extent. Additionally, we compared our results with previous studies and concluded that our approach outperformed vulnerability prediction models based on *software metrics*, *includes* and *function calls* in terms of MCC and F1-score. In this chapter, we evaluate the practicality of RCLUV-based code profiles in real-world applications by predicting future vulnerabilities based on the data available in the past.

5.1 Introduction

Previously, we explored whether RCLUV-based code profiles can be utilized as a feature set for vulnerability prediction research. We examined the power of our feature

set in distinguishing vulnerable files from neutral files. The pattern of the learning curves and the models' evaluation results demonstrated that machine learning algorithms can be trained using RCLUV-based data to predict vulnerabilities.

Now that we are confident about the strength of our feature set, the next step is to explore the practicality of our approach in real-world applications. Although in the previous experiments, we used a dataset that was very close to real-world settings, we have not yet confirmed the predictability power of our developed models when evaluating them on future vulnerabilities. In those experiments we did not take the time factor into account, meaning that both test and training data were randomly selected from the entire history of the Linux kernel. In practice, at a certain point in time, prediction models only have access to the vulnerabilities that have been reported up to that time. We can simulate a similar situation by choosing a point in time, training the models on the previous data, and testing them on the future and unseen data. In this chapter, we conducted experiments to answer the following research question:

- RQ2. How effective are vulnerability prediction models trained on RCLUVs in predicting vulnerable files in the future?

The main goal of this research question is to examine whether vulnerabilities share similar characteristics over time or not. In the following sections, first, we provide a brief review of related studies on future release analysis, followed by a discussion of our approach in detail. Finally, we review the results of our experiments and compare them to a similar experiment in vulnerability prediction research from the literature.

5.2 Related Work

Beyond the classic cross-validation, future prediction is also a very common setup for the validation and performance assessment of a prediction model, especially in the vulnerability prediction area. In future prediction or next-release prediction, an historically earlier version/versions of a program is employed as a training dataset, while later versions are employed as a test dataset. Generally, future prediction is considered to be a better validation technique than cross-validation as the former simulates the real-life scenario of a prediction model. In this section, we review several studies that performed next-release validation as part of their experiments. Although the authors defined various training and testing sets in their next-release experiments, all of them tested their models on future releases (unseen data).

In 2012, Shin et al. [36] published a paper on vulnerability prediction to evaluate whether complexity, code churn and developer activity metrics are discriminative and practical metrics for predicting vulnerable locations of code or not. In order to evaluate their univariate and multivariate models, they performed next-release validation on 34 releases of Mozilla Firefox. Based on their definition, next-release validation at release N uses all of the data previous to N as the training data and release N as the test data. After performing the experiments, they reported the average of the results across all releases. Their prediction results indicated that the recall values are over approximately 70% while the precision was very low (below 5%) for all univariate and multivariate models. Based on the release-based chart they provided in their paper, their results were slightly improved over time. Hence, they suggested that updating models over time was helpful.

Chowdhury et al. [5] investigated whether complexity, coupling and cohesion metrics could be used to predict vulnerabilities. The authors empirically validated the proposed approach on fifty-two releases of Mozilla Firefox. They performed a next release validation using the data from the first 32 releases for training their models and the remaining 20 releases for testing purposes. Their reported average recall is approximately 70% with a 30% false-positive rate. They justified the increase in false-positive rate by the fact that they had balanced the training set and tested their models on an unbalanced test set to mimic the real-life situation.

Scandariato et al. [31] investigated the possibility of predicting the vulnerability status of software components using text mining techniques. The authors performed an extensive set of experiments on 10 open-source Android applications, including: within-project experiments, future release experiments, and cross-project experiments. For the prediction in future releases experiment, they trained their models on the first version of each android application and tested their models on the rest of the versions. Their motivation was to evaluate how far in the future a prediction model would work. The reported recall and precision of all the experiments are both above 80%. The authors also noted that in the majority of the cases, the average performance in future prediction is above the benchmark when using the random forest technique. They also stated that the performances of their models tested on the future releases were stable for at least 13 months on average.

In 2016, Jimenez et al. [15] conducted a vulnerability prediction study on the Linux kernel. In their study, they replicated and compared the main vulnerability prediction models that are based on (a) *softwaresoftware metrics* [33], (b) *text mining* [31], (c)

includes and (d) *function calls* [23]. In this paper, the authors performed next-release evaluation on various releases of the Linux kernel and reported the results. As mentioned in the previous chapter, we have used the same source of information (National Vulnerability Database) and approach to create and label our dataset. In the following sections, we will explain the similar approach we took for the next-release validation and compare our results with the results of this paper.

5.3 Experiment Design

In this experiment, we were interested to see how vulnerability prediction models trained on RCLUVs perform in real-life scenarios when predicting future and unknown vulnerabilities. A good vulnerability prediction model should be able to detect vulnerabilities that have not been reported yet. In order to simulate a realistic setup, we had to choose reference points in time and, for each point, split our data into two groups, "past" and "future". We could choose any set of points in time from 2005 to early 2019, however, for the purpose of making these experiments similar to how our solution will be used in real-life, we decided to choose the release dates of the Linux kernel. Therefore, to answer our second research question, we ran 30 experiments on 30 Linux releases from 2005 to early 2019.

It should be noted that Linux's vulnerable files that had been reported from 2005 to early 2019 are associated with 55 releases from the v2.6.12 (released 6 May 2005) to v5.2 (released 7 July 2019). Figure 5.1 shows the frequency of vulnerable files that have been fixed in these releases. As stated before, we decided to split our dataset according to 30 release points from release v2.6.25 to v4.15 of the Linux kernel. For each release point, we trained our models based on the historical data

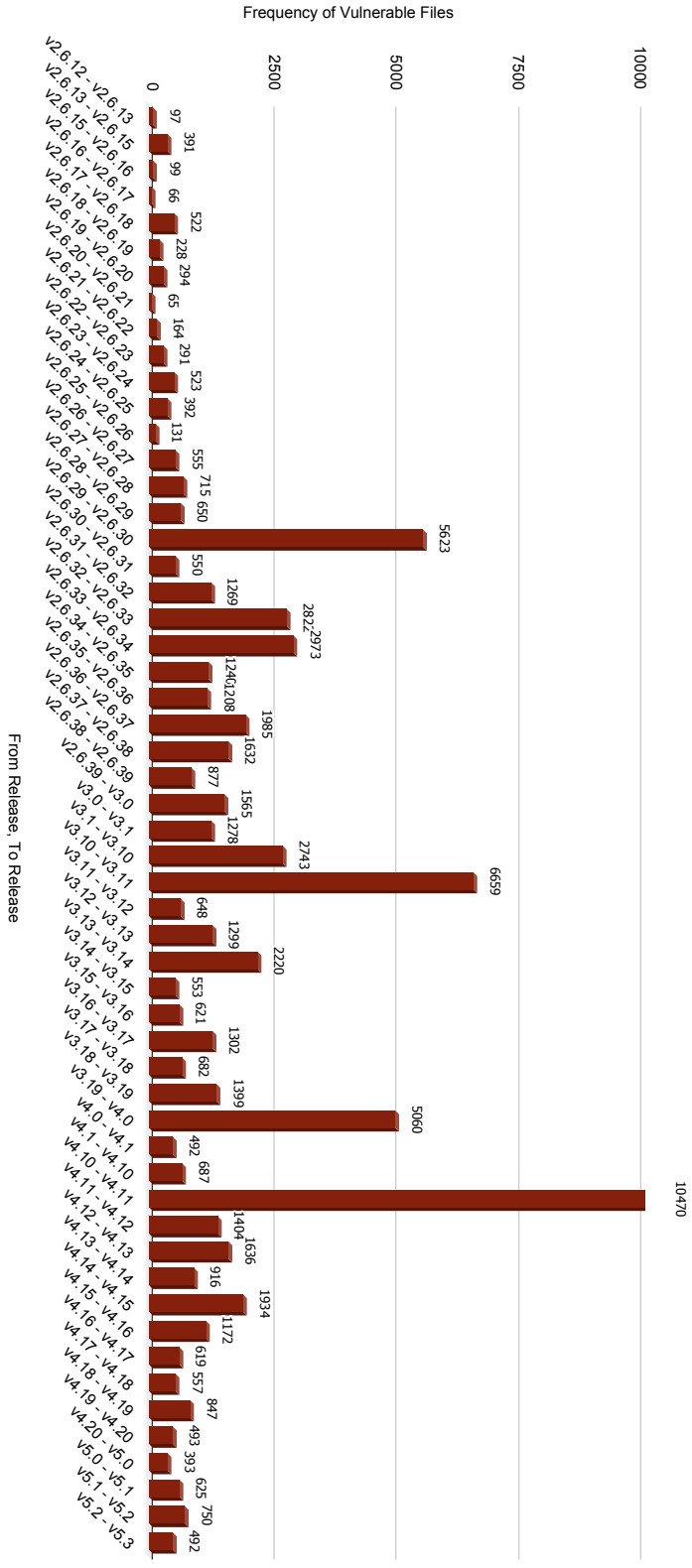

Figure 5.1: Number of vulnerable files that have been fixed over time

retrieved before the release point and evaluated our models based on the data retrieved after that release point. For instance, for release v3.1, the training set consisted of all vulnerable files that have been fixed by the time of the release date (October 24 2011) along with neutral files randomly selected from the previous releases. Moreover, the test set contained all vulnerable files that have been reported after the release date (October 24 2011) as well as neutral files selected from the same period. Figure 5.2 demonstrates two examples of how we divided the training and testing data in different experiments.

It is important to note that, at the time of writing this thesis, 55 releases of the Linux kernel with at least one vulnerability were available in the Git repository. However, we decided to limit our experiments to 30 of these releases in the middle since we wanted to make sure that we have adequate training and test data in all of the experiments. If we chose to begin our experiment from the early releases, we would not have a reasonable amount of historical data to train our models, and similarly for test data in recent releases we would not have enough future release data.

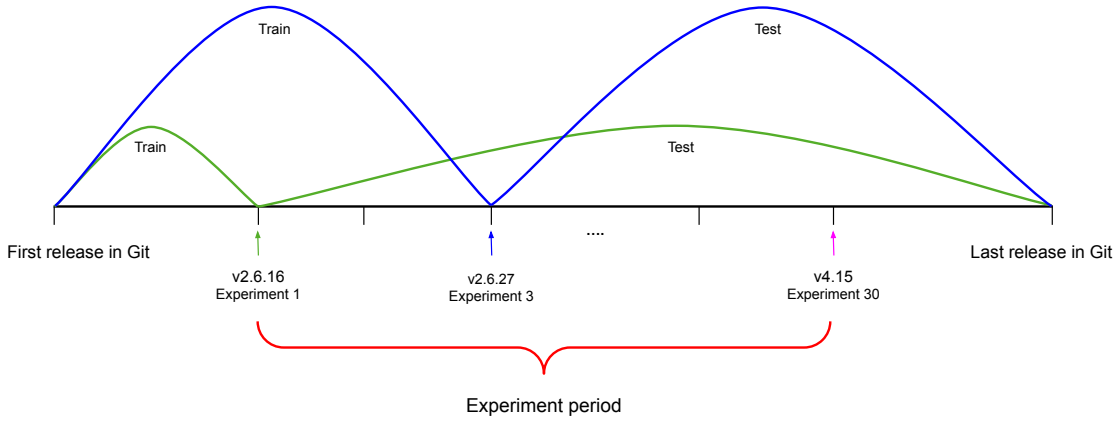

Figure 5.2: Train and test data split in Experiment 2

We used random forest and gradient boosting classifiers to train 60 models since these algorithms outperformed other ones in our experiments for RQ1. In addition, we performed under-sampling by randomly removing the data instances in the major class (neutral files) from the training set until the number of vulnerable and neutral files became equal. We only under-sampled the training data, not the test data, since the latter should reflect the real distribution of vulnerabilities in the Linux kernel, which is approximately 3%. In all 60 experiments, we used the same set of hyper-parameters that we chose in RQ1 experiments. Consequently, at this stage, we did not focus on tuning hyper-parameters for each of the 60 experiment separately because of it being computationally expensive.

5.4 Results

Figures 5.3, 5.4, and 5.5 demonstrate the precision, recall, and f1-score of the random forest models in predicting future vulnerabilities. As mentioned before, we also ran the same experiments using XGBoost and compared the results of these statistical learners. Figures 5.6, 5.7, and 5.8 present how XGBoost performed.

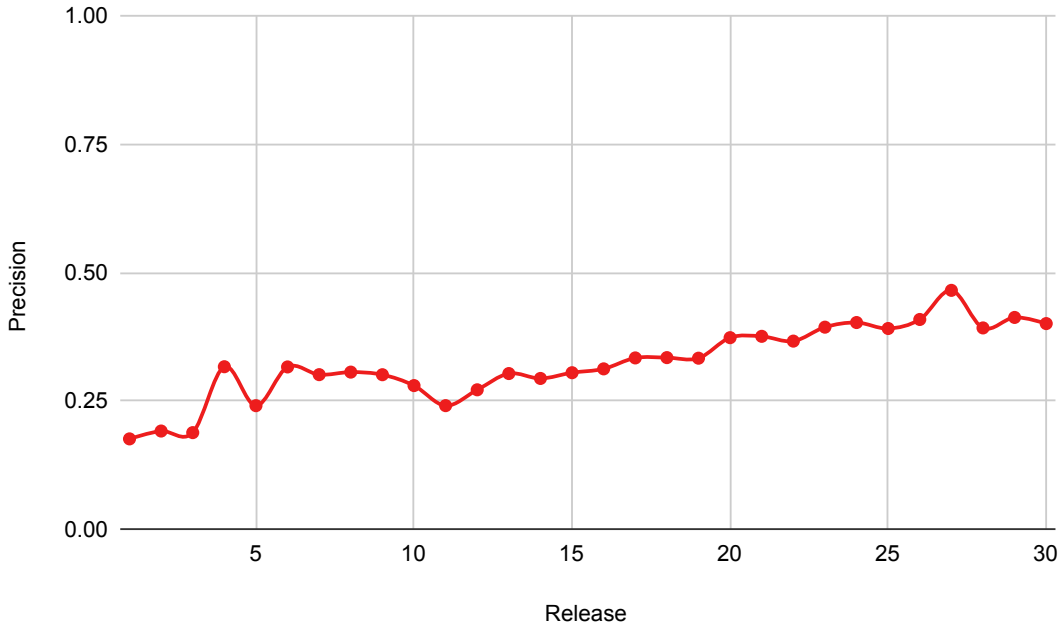

Figure 5.3: Precision in future prediction experiments using random forest

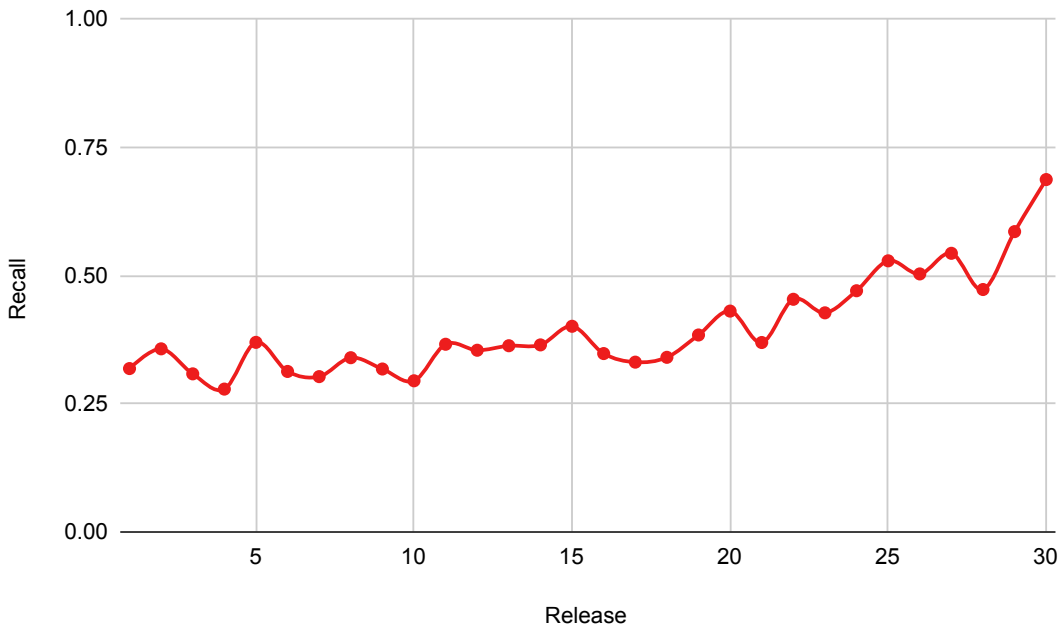

Figure 5.4: Recall in future prediction experiments using random forest

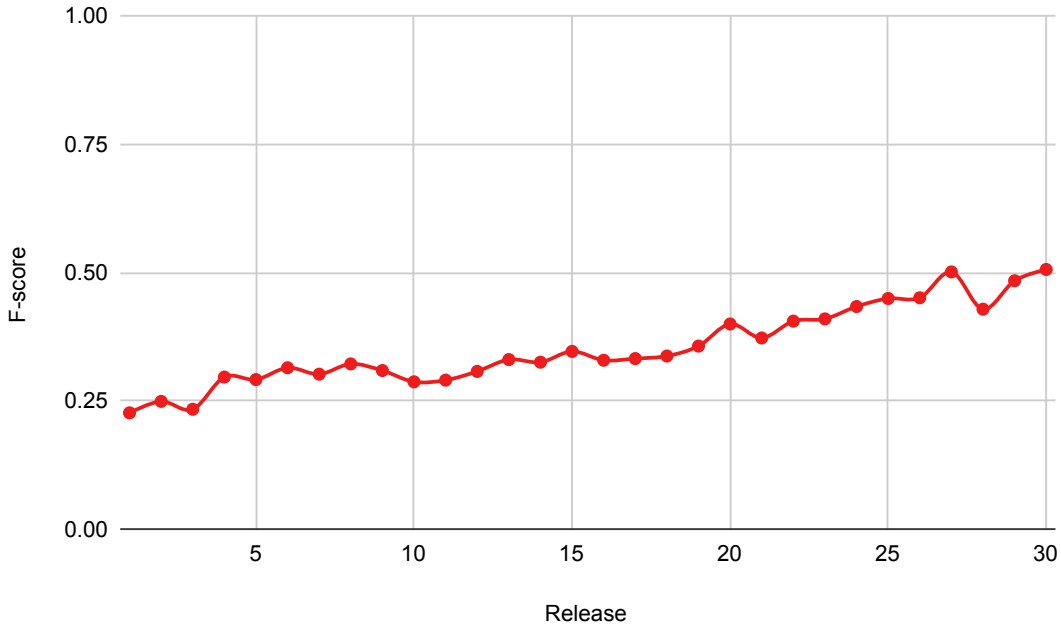

Figure 5.5: F1-score in future prediction experiments using random forest

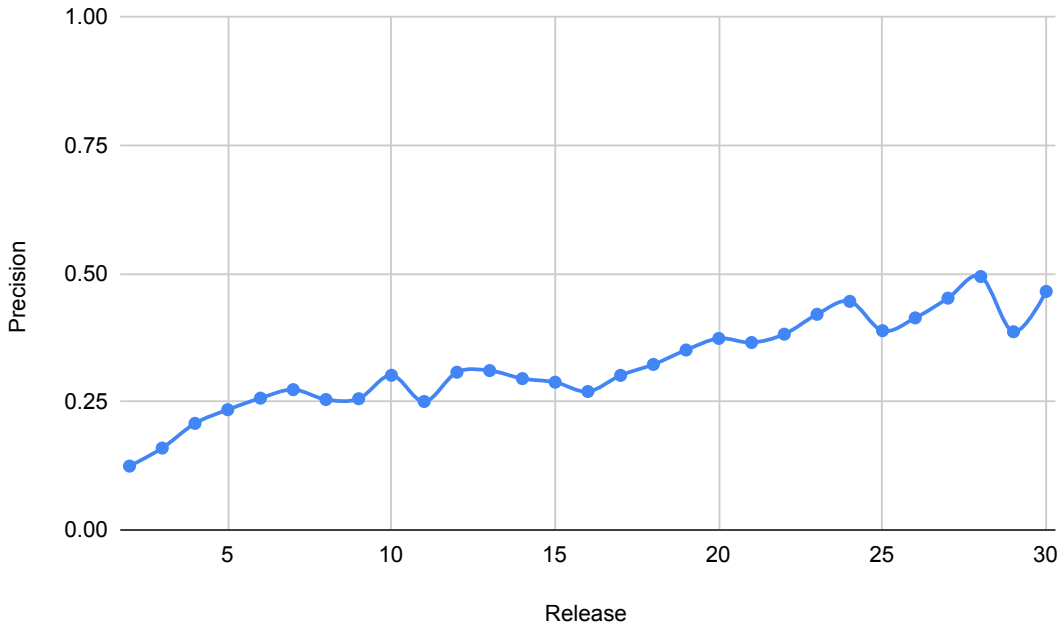

Figure 5.6: Precision in future prediction experiments using XGBoost

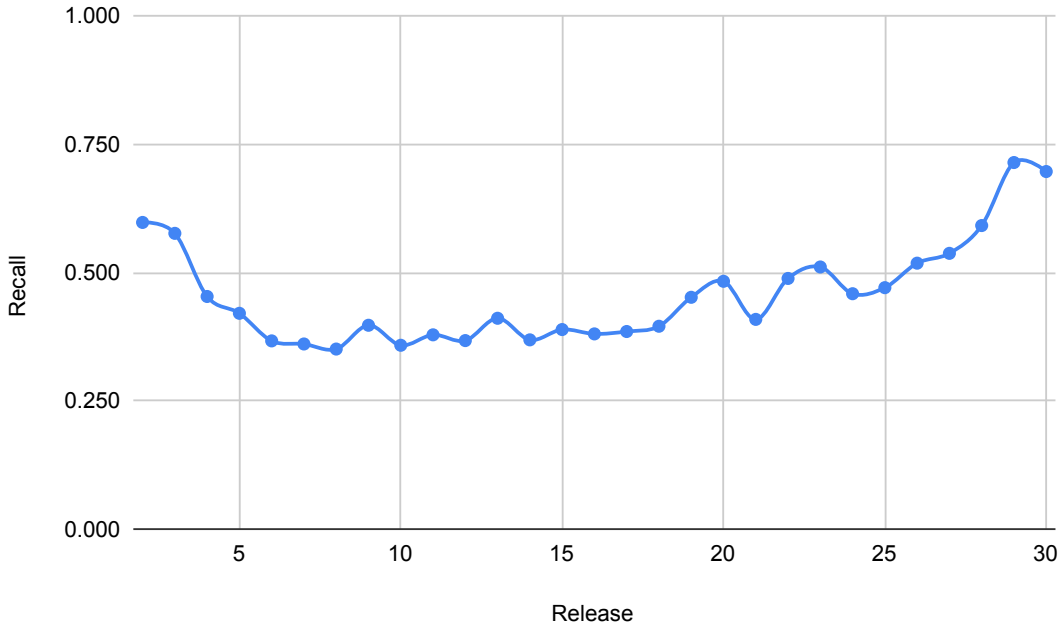

Figure 5.7: Recall in future prediction experiments using XGBoost

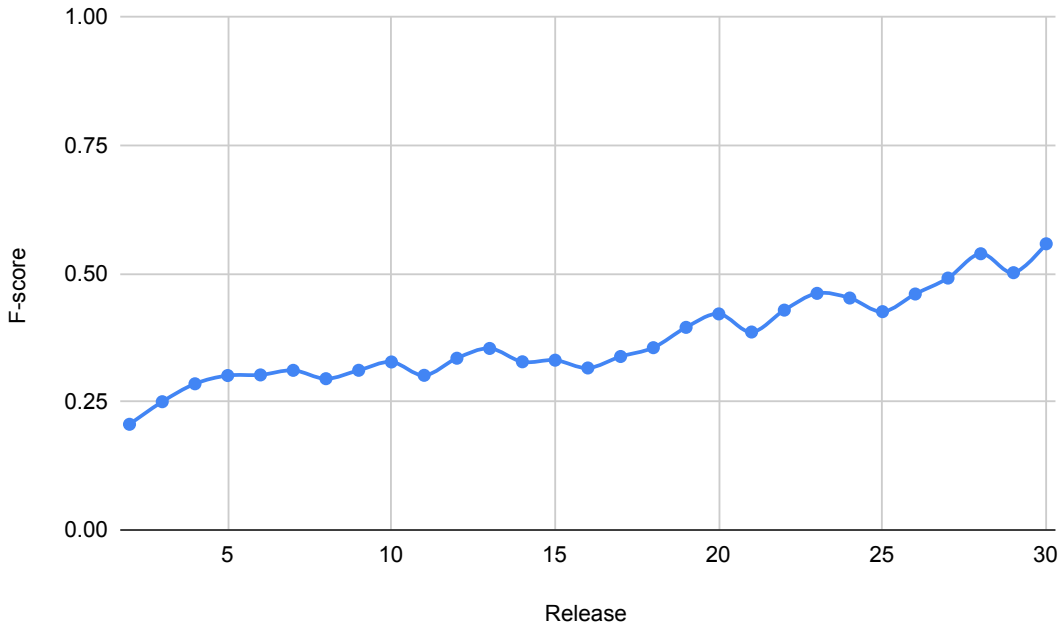

Figure 5.8: F1-score in future prediction experiments using XGBoost

5.5 Analysis and Conclusions

In random forest models, precision has an upward trend over time starting from 17% and reaching to the maximum of 47%. The recall values also increase noticeably over time but with some ups and downs. The best recall was obtained in the last experiment reaching to approximately 70%. Indeed looking at precision and recall individually is not enough because of the trade-off between the two metrics. Therefore, we drew the f1-score charts to make sure about our conclusions. Figure 5.5 depicts the f1-score graph rising from 22% to 51%.

For the XGBoost models, the trend of the precision graph is upward with values ranging from approximately 11% to 49%. The recall graph fluctuates gently started from nearly 60%, fell slightly in the middle, then reached to its maximum value (71%). For the first three release points, the recall values are relatively high, however, precision values are low. Therefore, to have a better understanding, we should consider the f1-scores. Figure 5.8 confirms that, over time, the f1-score of the XGBoost models increased 36% starting from 19% and ending with 55%.

From the results and the patterns of precision, recall and f1-score charts, we drew four important conclusions:

First, we found out that the choice of statistical learner between random forest and XGBoost does not make a significant difference in the performance results for our dataset.

Second, we can confidently confirm that vulnerabilities share similar characteristics over time. This can be inferred from the fact that utilizing historical data to build prediction models was proved to be effective in detecting new and unseen vulnerable files reasonably well.

Third, the noticeable improvement in precision, recall, and f1-score over time suggests the importance of keeping vulnerability prediction models up to date with the recent data. This can be justified by the fact that refreshing the models allows them the chance to observe and learn new patterns of vulnerabilities. We evaluated all of the models on the data related to the entire future vulnerabilities starting from the model's associated release point. Although the difference between the test sets of the adjacent experiments is very limited, it is clear that in most cases the f1-score was enhanced. We believe the data of the recently found vulnerabilities, even though very small in size, is the main reason for the improvement. This conclusion supports Shin et al. [36]'s finding that updating the models will help improve the results.

Fourth, it can be inferred that RCLUV code profiles are effective features for predicting future vulnerabilities using historical data. Although, at first, the efficiency of using this feature set was completely unknown, results show that RCLUVs are capable of detecting future vulnerabilities for the Linux kernel. The average f1-score for both random forest and XGBoost models is 37%. Average recall for the XGBoost and random forest models is 47% and 41%, respectively. In addition, based on the recall results, our models were able to identify more than half of the vulnerabilities in the latest experiments.

5.6 Comparison

Jimenez et al. [15] also performed future release prediction for Linux in their research. They used four different sets of features based on *software metrics*, *text mining*, *include* and *function calls*. They ran the experiment on 20 releases of Linux selected from a similar (but different) period of time (from v2.6.28 to v3.7). The performance

results of their experiment, which was done in a similar setup to our experiment, shows that *function calls* performed the best among their feature. Since they did not provide the f1-score in their paper, we had retrieve precision and recall from a chart in their report and calculate the f1-score based on those. Based on our calculations, the f1-score in their experiment using *function calls* starts from 47% in the first release and increases up to 63%.

Comparing our results with [15] shows a huge difference in f1-score in the first (earliest) experiment. Our results show an average of 20% while their *function calls* results indicate 47%. We think this gap might be the result of some differences in how vulnerable and neutral files are assigned to a release point in addition to the fact that the time frame of the two studies are different. However, due to lack of details about this in their report, we could not investigate our assumptions further.

5.7 Summary

In this chapter, we investigated the possibility of predicting future Linux kernel vulnerabilities based on historical data using the RCLUV code profiles as a feature set. We performed 60 experiments on 30 releases of the Linux kernel using two supervised machine learning algorithms. Random forest and XGBoost experiments showed similar predictive performance. From this we concluded that the choice of statistical learner between these two models does not heavily impact the results. We also found evidence that vulnerabilities share similar characteristics over time, since the models were able to successfully predict future vulnerabilities. More importantly, our results suggested that RCLUV code profiles are effective features for predicting future vulnerabilities, with an average recall of 47% and f1-score of 37%. All of the performance

metrics improved with more training data, which emphasizes the importance of updating the models over time. In the next chapter, we provide a brief review of the contributions and conclusions of our research. Subsequently, we explore the threats to validity and discuss possible future work.

Chapter 6

Conclusion and Future Work

This thesis has presented a vulnerability prediction study on the Linux kernel. As a first research question, we have analyzed the potential of using RCLUVs, a formal encoding of the language feature usage, of programs for predicting software vulnerabilities. The answer to RQ1 is that RCLUV-based code profiles can be used to train machine learning algorithms to detect software vulnerabilities. A comparison of results with four different popular vulnerability prediction models on a similar dataset indicate that RCLUV-based models perform better than approaches based on *includes*, *function calls*, and *software metrics* in terms of MCC and f1-score. In RQ2, we explored the idea of using RCLUV-based code profiles to predict future vulnerabilities based on historical data. Results suggest that the models learned from past vulnerabilities can predict more than half of the new and unseen vulnerable files successfully by having enough training data. We have several contributions in this thesis, which we will discuss in the next section.

6.1 Contributions

In this thesis we have five main contributions.

6.1.1 The idea of using RCLUVs as a feature set for vulnerability prediction

In this thesis, we have implemented the idea of using Rich Contextualized Language Use Vectors as a feature set for predicting security vulnerabilities for the first time in the literature.

6.1.2 RCLUV Extractor for C programming language

To extract RCLUV features from C source file, we developed an RCLUV extractor using TXL [6]. We had to update and tune the existing TXL grammar for the C programming language and filter out redundant language features. The resulting RCLUV extractor can be used in any future research to extract features of projects in the C programming language. The RCLUV extractor is available in our project repository on Github [10].

6.1.3 RCLUV Dataset of Vulnerable and Neutral Files from the Linux Kernel

Using the reports from NVD, we labeled vulnerable and neutral files throughout the history of the Linux kernel and ran our RCLUV extractor on them. The result is a labeled dataset of 1,047 features, including the release point of the file, extracted from approximately 75,000 files. The dataset might potentially be useful in future vulnerability prediction research. This dataset is available on Github [10].

6.1.4 Vulnerability Prediction Models

To answer our two research questions, we developed, trained and validated machine learning models. We explored whether it is possible to predict vulnerabilities using RCLUVs. We ran the experiment in 2 different setups. First, without considering the time factor; we trained models based on vulnerabilities reported thoroughout the history of the Linux kernel and validated them using a similarly sampled subset of the dataset. Second, we choose 30 points of time in the history of the Linux kernel, trained the models based on the data reported before each point and validated them on the data reported afterward. The latter simulates a real-life situation in which the model might be used. The performance metrics of each model were reported in this thesis and might be used by other researchers to compare with their results.

6.1.5 A Comparison of Vulnerability Prediction Approaches

We compared the results of vulnerability prediction models based on RCLUVs with four different popular vulnerability prediction approaches based on *includes*, *function calls*, *software metrics*, and *text mining* in both experimental and realistic scenarios.

6.2 Discussion and Threats to Validity

It is possible that our experiments do not demonstrate the power of RCLUVs in vulnerability prediction at their full strength. There are some limitations regarding our research and in vulnerability prediction research in general which might affect the results. In this section, we discuss some of the limitations and threats to validity.

We generated our dataset automatically, based on data extracted from the CVE-NVD database using Git commits. This method ensures that the previously known

and fixed vulnerabilities are retrieved completely. However, in reality, there might exist some undiscovered or non-fixed vulnerabilities that can potentially affect our performance measurements. For example, our models might detect unknown vulnerabilities correctly but since the vulnerabilities have not been discovered yet, they will be counted as false positives.

There exists another issue regarding the automatic labeling of our dataset. We marked the vulnerable files of our dataset using Git commits, such that for all commits that fixed a vulnerability, we checked out the previous commit, and marked the files modified in that commit as vulnerable files. However, there is a chance that some of those files might not be actually vulnerable. For instance, the actual vulnerability might exist in one file, however, in order to fix that file, some other files may have been changed as well. In our approach, we marked all of those files as vulnerable. This issue is not limited only to our research. All other research that builds its dataset based on Git commits might encounter the same problem. We think that this issue might be addressed by introducing a manual step in the labeling process in which a specialist would look at commits with several files individually and decide which files should be marked as vulnerable.

One probable factor affecting the predictive power of our models may be the quality of the data from the National Vulnerability Database. Zhang et al. [43] discussed some limitations of the NVD dataset such as missing information, vulnerability release time and data errors. Ozment [25] also performed a number of studies on analyzing NVD and pointed out several limitations of this database. Thus, imprecise information in the NVD could generate noise in our data. However, to the best of our knowledge, NVD is the leading complete and standard free source of vulnerability

information available. Considering the lack of options available for retrieving information on Linux vulnerabilities and the fact that the research community has chosen NVD as the main source of information for building their dataset, we decided to take the same approach.

Our work in this thesis was limited to the C programming language and Linux kernel since we wanted to compare our results with existing work in the literature. While our approach can be generalized to be used in other programming languages and other projects, the performance of the models might vary due to the nature of the projects, the quality of the vulnerability report databases and the programming language of the project.

6.3 Future Work

This study forms a first step towards building security vulnerability prediction models using RCLUVs. Results show that the proposed approach performs reasonably well but there is a huge space for improvement. Therefore, more research is needed to enhance the performance of code profile-based vulnerability prediction models in order to get the maximum benefit out of them. Our research can be extended in different directions. in this section, we describe the potential next steps of this research.

Currently, our models predict whether an input file is vulnerable or neutral based on historical data. Finding the vulnerable files in a software system can help companies to focus their efforts only on the suspicious parts of the systems which leads to reduce the developers' time and effort. However, in many projects, developers might need more help to pinpoint the location of vulnerabilities in a file. One way to make our prediction results more helpful and understandable is to take one step

further by developing explainable machine learning models using RCLUVs. Building explainable machine learning models will bring various benefits. For example, it will enable developers to understand why and how the prediction was made. Specifically, based on the nature of RCLUVs, we think it would be possible to attach vulnerabilities directly to program structures. Developing explainable AI models using RCLUVs facilitates the developers' time to find and fix the vulnerabilities by knowing which features or contextual combinations of features of the language used in their code might be correlated with the vulnerability predictions.

We are interested in applying the presented approach to other types of projects and programming languages (e.g. Java or C++) where the source code of the project and information about vulnerabilities are available.

In this study, we used an intra-project setting, meaning that training and test data came from the same project. In the future, we would like to consider cross-project settings by testing the models on different projects that have not been used for the training phase.

Another extension of this work can be through further analysis in combining different features and learning methods to build more powerful and composite vulnerability prediction models that can benefit from the strength of all approaches.

In addition, finding an appropriate *level* to explore the parse tree of a program is significantly important. Based on the *level* value that is selected, the length of an RCLUV vector can vary for the same program meaning that the number of features representing each file (feature set) can increase exponentially if we explore the parse tree of a program in more depth. In this study, we used a level 1* (single level contextualized) extractor rather than a level 2, 3 or higher. Our feature extractor

draws the frequency of the usage of the features in the context of other features (one level). Increasing the *level* value may generate many elements in the RCLUV of the files that might be rarely used and leads to an increase in the training time of the algorithms. However, at the same time, additional features might contain unique and valuable information which can be utilized to train more accurate models. In the future, we will investigate further how to select the most effective *level* of contextualization for creating RCLUVs.

We did not differentiate vulnerabilities into different types in this study. However, we are eager to build prediction models that are specialized to specific vulnerability types. It is possible to build such models using the information from the NVD database about the types of vulnerabilities.

Finally, we think that to make a more precise dataset, domain knowledge should be integrated into the process of labeling the dataset. Therefore, it might be helpful to define manual processes to label the vulnerable files; for example, by checking all the commits individually.

Bibliography

[1] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.

[2] Harold Booth, Doug Rike, and Gregory Witte. The National Vulnerability Database (NVD): Overview. Technical report, National Institute of Standards and Technology, 2013.

[3] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[4] 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.

[5] 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.

[6] James R Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[7] James R Cordy. Excerpts from the TXL Cookbook. In *Generative and Transformational Techniques in Software Engineering III, Lecture Notes in Computer Science 6491*, pages 27–91, 2011.

[8] IT Security Database. CVE Details. `https://www.cvedetails.com/browse-by-date.php`, 2019. [Online; accessed 7-October-2019].

[9] David J Dittman, Taghi M Khoshgoftaar, and Amri Napolitano. The effect of data sampling when using random forest on imbalanced bioinformatics data. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 457–463. IEEE, 2015.

[10] Ghazal Fouladfard. Vulnerability Prediction Using RCLUVs. `https://github.com/Ghazall1993/Vulneability-Prediction-Using-RCLUV.git`, 2020.

[11] 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.

[12] 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.

[13] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017.

[14] 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.

[15] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Vulnerability prediction models: A case study on the linux kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10. IEEE, 2016.

[16] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Enabling the Continous Analysis of Security Vulnerabilities with VulData7. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2018, Madrid, Spain, September 23-24, 2018*, 2018.

[17] Ivan Victor Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[18] Information Technology Laboratory. CVSS Severity Distribution Over Time. `https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time`, 2019. [Online; accessed 6-October-2019].

[19] Information Technology Laboratory. Understanding Vulnerability Detail Pages. `https://nvd.nist.gov/vuln`, 2019. [Online; accessed 30-October-2019].

[20] Patrick Luk. An Empirical Analysis of PHP in Open Source Applications. Master's thesis, Queen's University, Kingston, Canada, 2017.

[21] Gary McGraw. Automated code review tools for security. *Computer*, 41(12):108–111, 2008.

[22] 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.

[23] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, 2007.

[24] Andy Ozment. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, pages 6–11. ACM, 2007.

[25] James Andrew Ozment. *Vulnerability discovery & software security*. PhD thesis, University of Cambridge, 2007.

[26] 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.

[27] Ashiqur Rahman. *Software Defect Prediction Using Rich Contextualized Language Use Vectors*. PhD thesis, Queen's University, Kingston, Canada, 2019.

[28] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[29] Riccardo Scandariato and James Walden. Predicting vulnerable classes in an Android application. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 11–16, 2012.

[30] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting Vulnerable Software Components via Text Mining. *IEEE Trans. Software Eng.*, 40(10):993–1006, 2014.

[31] 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.

[32] Fatemeh Shahjamali. An Empirical Analysis of Java Language Use in Open Source Applications. Master's thesis, Queen's University, Kingston, Canada, 2019.

[33] 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, 2010.

[34] 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*, pages 315–317. ACM, 2008.

[35] 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.

[36] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[37] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, (3/4):441–471, 1987.

[38] The ultimate security vulnerability datasource. Top 50 Products By Total Number Of "Distinct" Vulnerabilities. `https://www.cvedetails.com/top-50-products.php`. [Online; accessed 2-November-2019].

[39] Karthik Vishwambar. Automated Generation of Language Use Vector Extractors from TXL Grammars. Master's thesis, Queen's University, Kingston, Canada, 2019.

[40] Tom Van Vleck. The IBM 7094 and CTSS. `https://multicians.org/thvv/7094.html`, 2019. [Online; accessed 30-October-2019].

[41] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33. IEEE, 2014.

[42] Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit. In *Proceedings of the Sixth ACM Conference on Data*

and Application Security and Privacy, CODASPY '16, pages 97–104, New York, NY, USA, 2016. ACM.

[43] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the National Vulnerability Database (NVD) to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

[44] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. Combining software metrics and text features for vulnerable file prediction. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 40–49. IEEE, 2015.

[45] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010.