Commit-Level vs. File-Level Vulnerability Prediction

by

Michael Chong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Helping software development teams find and repair vulnerabilities before they are released and exploited can prevent costs due to loss of data, availability, and reputation. However, while general defect prediction models exist to help developers find bugs, vulnerability prediction models currently do not achieve high enough prediction performance to be used in industry [43]. Prediction of vulnerabilities in commits and files has been explored by previous work, and while commit-level prediction, at a finer granularity, may offer more useful results, there exists no clear comparison in predictive performance to justify this assumption.

To inform further research in vulnerability prediction, we compare commit and file-level prediction, across 7 projects, using 6 classifiers, for 8 different training dates. We evaluate the performance of each prediction model using 'online prediction' for ensuring an evaluation in line with practical usage of the prediction model. We evaluate each model using four different metrics, which we interpret as representing two different practical usage scenarios. We also perform an analysis of the data and techniques for evaluating prediction models. We find that despite achieving a low absolute prediction performance, file-level prediction generally tends to outperform commit-level prediction, but in a few outstanding cases, commit-level performs better.

Acknowledgements

I would like to thank my advisor, Lin Tan, for guidance, advice, and patience with my work on this research. I would like to thank fellow members of the asset team (present and past) who have helped contribute to this research, particularly Thibaud Lutellier, but also Song Wang, Ming Tan, and Nasir Ali, as well as other member who provided helpful advice, feedback, and inspiration. I would also like to thank the VCCFinder team for sharing their results and source code, which facilitated the replication of their results.

Dedication

This is dedicated to my parents, friends, and family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

High-profile disclosures of software vulnerabilities continue to make headlines with increasing frequency. Software vulnerabilities (permission errors, buffer overflows, or SQL injections, for example) are bugs that a malicious attacker may "exploit to cause loss or harm" [52]. The public disclosure of a vulnerability can take a financial toll on a company's stock prices [74], and the cost of developing and deploying a patch can be expensive [63]. Additionally, vulnerabilities expose users to attacks that may reveal or disrupt sensitive data or services, and these attacks can also have a significant financial impact—it has been estimated that they cost the global economy more than $400 billion annually[5]. Because of the significant impact and cost of vulnerabilities, research on new tools to help developers identify and repair vulnerabilities is crucial.

Vulnerability prediction techniques, based on related work in defect prediction, have been developed to predict elements of software (files, binaries, changes, or commits) that are likely to contain vulnerabilities. They do so by leveraging 'features', or attributes that characterise the elements they are predicting on. Features such as code complexity, developer activity metrics, or source code keyword metrics are used for this purpose. These prediction models can be applied at varying levels of granularity, including binary [79], component [48], file [68], commit [50] and change [31, 73]. At each level of granularity, the classification performance and utility (or cost effectiveness) of vulnerability prediction may vary. These variations may be due to differences in element size, semantics and distribution of element attributes ('features'), and frequency of vulnerable elements.

Vulnerability prediction techniques proposed at different levels of granularity have obtained various degrees of success, and some recent studies have been investigating which level of granularity is the most promising to work with [54, 43]. In particular, Morrison

et al. [43] examine vulnerability prediction at *binary level* and *file level*. They suggest that vulnerabilities predicted at *file level* provide information that is more useful to developers than those predicted at binary level, but find that when compared, file-level has worse (i.e., lower) prediction performance results. Posnett et al. [54], examine defect prediction in general and compare *package-level* prediction with *file-level* prediction. In line with Morrison et al.'s results, they find that prediction at file level provides comparable or *worse* performance than at *package level*, measured in terms of traditionally used metrics measuring classification performance. However, they also evaluate classification using *'cost effectiveness'* metrics, and find that *file-level* prediction is significantly more cost effective than prediction at a higher level of granularity. Based on these results, one might conclude that the finer granularity of file-level prediction is more useful, but harder to do, than the coarser grained package and binary-level prediction.

While the *file-level prediction* models discussed above predict which files are vulnerable in a given version of a project, *commit-level prediction*[50] predicts which commits, made over a project's history, are vulnerable. File-level prediction models are often trained using past versions of the project, where each file is labelled as "vulnerable" (containing a vulnerability), or "clean" (no vulnerability yet found) [68]. On the other hand, commit-level models are trained using past commits labelled as "vulnerable" (commits introducing a vulnerability), or "clean".

One might expect based on the studies [43], and [54], discussed above, that commit-level prediction, at a finer granularity, would be more useful for developers, but have worse prediction results. However, a recent study, VCCFinder [50], proposed a method to predict vulnerabilities at commit level that obtained a comparable performance to previous vulnerability prediction at file level [68]. However, they achieve this performance on a different dataset, and using a different machine learning algorithm. A direct comparison between commit- and file-level vulnerability prediction, to the best of our knowledge, has yet to be performed.

As discussed by Morrison et al., while general defect prediction has achieved high enough performance that it is currently being employed in industry by software development companies like Microsoft, vulnerability prediction has yet to achieve acceptable performance for practical use in industry [43]. For this reason, our work seeks to provide insight to further inform the development of vulnerability prediction towards the development of practical industry tools. In order to find out which level of granularity it would benefit us most to develop further with improved vulnerability-specific attributes and prediction techniques, a direct comparison of the two prediction granularities is required.

Is commit-level vulnerability prediction better than file-level vulnerability prediction? Commits and files have different semantics and attribute sets that may benefit one over the other, and which make direct comparison of the two prediction granularities nontrivial. For example, in mature projects, commits generally contain fewer lines of code than files, while there exist many more commits over the project's history than there are files in a given version. In addition, while files, packages, and projects ("binary level"), share a hierarchical relationship (e.g., projects, packages contain a set of files), and analogous features can be calculated through aggregation, the same is not true for commits and files. This means that there are some differences in the semantics and distribution of 'features', or attributes, used at file level versus those used at commit level. Because of these differences, a set of features that may work well for one granularity level (file or commit) may not map directly to the other. For example, features of the commit message, or the commit time, may contribute to prediction performance for commit-level prediction [31, 50, 73], but these have no direct analogy at file level. On the other hand, cyclomatic complexity may be less meaningful at commit level, as a commit contains additions and deletions to lines in a file, and does not intuitively have a program path.

Our research seeks to develop as fair a comparison as possible, built on the same projects over the same time periods, and using a variety of features and their analogies at both commit and file level, in order to directly compare the two prediction granularities. By doing so, we hope to inform further research in vulnerability prediction by providing insight into which level of granularity performs better, and how each level of granularity is impacted by current techniques used for evaluating prediction performance.

1.1 Developing a Practical Technique

In the end, this research is motivated to facilitate the development of techniques that can be used practically in industry in order to aid software development teams find and fix vulnerabilities before they are released and can be exploited. To that end, we design both our experiment and our evaluation to reflect, as much as possible, the effectiveness of prediction for such practical, real-world use. We choose an experiment design based on the *Online Prediction* [73] technique, which simulates the practical use of a classifier for real-world prediction on a software project. We argue that Online Prediction is a more rigorous process of evaluation, due to the challenges it poses through the introduction of undiscovered vulnerabilities, as discussed in Section 2.4. For our evaluation, we explore a variety of different classifier performance metrics to compare two different usage scenarios, similar to the evaluation performed by [54].

When evaluating prediction models, traditional techniques, such as k-fold cross validation can result in evaluations with artificially high accuracy and precision when compared with performance of models used in production [73]. As such, *Online Prediction* was developed in order to accurately evaluate prediction performance in a real-world setting. When training a prediction model, instances (files or commits) from before a "training date" are collected to be used as a training set. However, if vulnerabilities that have been discovered after the training date are used to label training instances for an evaluation, the resulting performance of the classifier will be at an advantage from this use of "future data" (this also occurs in k-fold cross validation). This means that a production classifier built to detect vulnerabilities in recent project versions or commits may not perform as well as in an evaluation.

Previous work [68, 50, 43] then evaluates the results of the prediction based on criteria derived from the *confusion matrix* of the classification results, which present the classification errors, *False Positives (FPs)* and *False Negatives (FNs)*, as well as the correctly classified *True Positives (TPs)*, and *True Negatives (TNs)* (as described further in Section 2.7). Posnett et al. argue that evaluating defect prediction based on 'cost effectiveness' metrics, that is, the number of vulnerabilities discovered versus the number of lines of code inspected to find those vulnerabilities, is a more rigorous and higher standard with which to measure performance. They also argue that it is more accurately represents the practical utility of the classifier. In our evaluation, we propose two *usage scenarios* represented by different metrics and capturing different possible uses of the classifier. The 'classification performance' usage scenario, measured by traditional classification performance metrics F_1 score and AUROC (discussed in Section 2.7), evaluates the quality of prediction if all vulnerable instances as predicted by the classifier are able to be examined. On the other hand, the 'cost effectiveness' usage scenario, measured by AUCEC and TPR10k (again discussed in detail in Section 2.7), evaluates how many vulnerable instances are discovered by examining instances in order of likelihood, and at a cost equivalent to the instance size. We use these two usage scenarios to describe how the evaluation metrics we use can be interpreted. The comparison of usage scenarios is discussed in more detail in Section 3.3.

1.2 Contributions

In this project, we compare the performance of vulnerability prediction at file level and commit level using an analogous set of features for each level of granularity. Our evaluation uses the online prediction method in order to present results that represent actual practical usage of a classifier. We perform the comparison on 7 projects, for 6 classifiers, and for

8 training dates, and present both 'classifier performance' and 'cost effectiveness' results. We also present an in-depth analysis of the data used in our evaluation and the process used to extract it, in order to gain insight on the techniques we used, (techniques based on those used in previous work), the data we collected, and its impact on our evaluation of vulnerability prediction performance. We find that for both 'classifier performance' and 'cost effectiveness', file-level prediction usually outperforms commit-level prediction. We also find that for some projects and classifiers, particularly the PHP project and J48 classifier, median AUROC and AUCEC scores are higher at commit level than file level. Finally, we note that for some ROC and CEC curves, one level of granularity may perform better below certain low thresholds (5% false postive rate for ROC curve or under 20% LOC for CEC curve), while overall the other level of granularity shows a higher prerformance.

This project makes the following contributions:

- A comparison of the effectiveness of commit-level and file-level vulnerability prediction using analogous features for each. We perform our comparison on 7 different projects, with vulnerability labelling data collected using an automated technique, using independently tuned project-specific heuristics to match vulnerabilities with fixing commits. We use four different metrics describing two different usage scenarios to evaluate the performance of our classifiers
- We find that file-level prediction generally outperforms commit-level under all metrics. However, we also find that for certain projects, in particular the PHP project, commit-level sometimes outperforms file-level under AUROC and AUCEC metrics.
- We produce a dataset including features/attributes of files and commits from the history of each of the 7 projects, labelled with vulnerability data from the National Vulnerability Database using variable training dates. It can be easily extended to include other projects, features/attributes, and training dates. It can also be extended to compare commit- and file-level general defect prediction on the same projects.

The remainder of the work is organized as follows: Section 2 describes important background information on vulnerability prediction. Section 3 describes our experimental design, and the research question we set out to explore. Section 4 details the specifics of our experimental setup, including which projects, classifiers, and training dates we used, and the data preprocessing we used for classification. Section 5 describes and discusses our findings, while Section 6 presents an analysis of the data used for prediction, as well as the effectiveness of the techniques used to collect the data. Sections 7 describes threats to the validity of our research. Section 8 discusses related work, and finally Section 9 discusses take away points, possible future work, and our conclusions.

Chapter 2

Vulnerability Prediction

Machine learning classification algorithms can be used to classify components (binaries, files, commits or changes) as likely or unlikely to contain a bug or vulnerability [31, 73, 68, 79]. Developers can then focus their limited resources on testing the subset of components classified as vulnerable or buggy.

Recall that classification algorithms are given a set of labelled instances as a 'training set', and build, or 'learn' a classification model based on the features in the data. For example, the J48 classifier constructs a tree of attributes that best divides instances into subsets containing either vulnerable or clean instances. Given an unlabelled instance, the classifier examines the tree of attributes, and follows each branch based on the values of that instance, assigning it a label upon reaching a leaf node. That is, the built classification model takes as input unlabelled testing instances, and assigns them a probability of containing a vulnerability, based on the learned patterns. In our evaluation, we consider instances with a probability above a certain threshold as 'vulnerable' (as predicted by the classifier), and measure predictive performance by comparing the predicted results to the actual labels for each test instance.

The evaluation of vulnerability prediction models follows the methods outlined in the following subsections. First, the data is divided into two parts: the training set, used to build the classifier, and the testing set, used for evaluation. Then, components are labeled as vulnerable or clean (not containing a known vulnerability). Finally, we extract 'features' (or attributes) for each instance, such as complexity metrics or commit metadata. The model is then evaluated using a set of metrics that highlight certain attributes of its performance (e.g. F_1 score, AUROC or cost effectiveness, discussed in Section 2.7).

2.1 Generating a Training Set

To build a vulnerability prediction model, we need to generate a training set of instances. The training instances represent software elements such as binaries, files, or commits. Each instance consists of a set of features and a label. Features represent characteristics of software elements extracted from various software metrics such as code complexity, code churn, developer activity, and keywords in source code [31, 73, 68, 79]. These software metrics can be readily collected from version control systems such as git. By linking a software element in a version control system with a reported vulnerability, we can label instances as vulnerable or clean. The National Vulnerability Database (NVD) maintains the identified vulnerabilities from widely used software products [49] and provides reference information to security issue reports and vulnerability-fixing commits of the products affected by the vulnerabilities. Thus, by the version control system and the NVD, we can label the training instances.

2.2 Training a Prediction Model

With the training set, we can build a prediction model by using various machine learning algorithms such as Naive Bayes or Random Forest. The model built by these algorithms can classify the software elements as vulnerable or clean [31, 73, 68, 79]. Before training a model, we may apply data preprocessing approaches used in machine learning such as feature selection and sampling to build a better prediction model [39, 10].

2.3 Predicting Vulnerabilities

When the prediction model is ready, we feed unlabelled instances into the model. Then, the model classifies these unlabelled instances as vulnerable or not. With the vulnerability prediction results, developers can effectively allocate their limited resources on reviewing and testing the subset of elements classified as vulnerable.

2.4 Online Prediction

Recall that in order to evaluate a classifier, we need a training set to build the classification model, and a testing set to evaluate it on. In order to simulate real-world use of a classifier,

we choose a training set of data taken from before a given 'training date', and evaluate on a testing set of data taken from after that date. In our evaluation, we aim to simulate 'online prediction' [73], meaning that we do not use vulnerabilities that have been reported after the 'training date' to label instances in our training data.

Using 'future data', or information learned after the training date, can falsely improve results [73]. For example, consider the evaluation of a classifier built using a classifier training date of January 2013. While relying on vulnerabilities discovered after January 2013 to label our data would be 'correct', it gives the classifier information it would not have known if it were truly trained on the data available in January 2013. In a sense, these undiscovered bugs lead to 'undiscovered vulnerable instances' in the training set, which can affect the classification performance of the classifier [73]. Undiscovered vulnerable instances are possible in any project at any point in time, but the amount of undiscovered vulnerable instances in a given period decreases as labelling information is used from further in the future, as can be seen in Figure 6.1, discussed in Section 6.3.

Lowering the number of positive instances is of particular concern in the context of vulnerability prediction, given the already low positive rate found in most data. For example, Shin and Williams[68] found that 21% of files in the Firefox 2.0 contained general bugs while only 3% of those files were vulnerable. For additional comparison, Tan et al. [73] find that 6 common open source projects have a general defect buggy rate of 15-37%, while our work on vulnerabilities, as well as that of Morrison et al. [43] have vulnerability rates of less than 10%. Table 4.1 (Discussed in detail in Section 4) shows the vulnerability rates shown in our data at commit and file levels. Given such a low rate of positive instances (i.e., vulnerable files), it is challenging to learn and evaluate accurate models [73, 28, 68].

In the testing set however, we rely on all available data in order to minimize the number of undiscovered vulnerable instances. Traditional techniques for the evaluation of classifiers, such as ten-fold cross validation, involve the use of future data. For this reason, they do not evaluate classifiers in a manner that is comparable to the real-world application of a classifier for vulnerability prediction.

2.5 Labelling instances

To identify commits and files that contain vulnerabilities, we rely on vulnerabilities listed in the National Vulnerability Database (NVD) [49]. The NVD maintains a database of publicly disclosed vulnerabilities for a number of software products. It also maintains information about the vulnerability, including a brief description, vulnerability type, severity

identifier, and links to external resources such as bug reports, commits, or security advisories.

To link the NVD information to vulnerable components in the software, we identify *vulnerability fixing commits*. Because the NVD is not always consistent in reporting vulnerabilities and fixing commits, we use three different methods to link vulnerabilities to fixing commits. For example, we look for mentions of the vulnerability identifiers used by NVD in commit messages, and identify commits with these messages as fixing these vulnerabilities. Further details of our labelling technique, as well as the effectiveness of our identification, is discussed in detail in Section 6.1.

Figure 2.1 depicts how commits and files are identified as vulnerable from vulnerability fixing commits. Figure 2.1 shows two 'vulnerability fixing commits' discovered at different points in time, and depicts how they are used, or left unused for labelling vulnerable instances in training sets at file and commit level. The upper timeline represents commits made over time. The lower timeline shows a series of versions, the files of which are used as the training set (V_1 to V_9), and testing set (V_{Test}) for file-level prediction at the given 'training date'. Figure 2.1 shows how files in these versions are identified as containing a vulnerability, as we are about to describe further.

C_{fix_1} is a vulnerability fixing commit, identified as linked to a vulnerability in NVD as described above (and in further detail in Section 6.1). The lines changed by the C_{fix_1} are considered to be the vulnerability. C_{blamed_1} is then identified as a 'blamed commit', by finding the most recent commit (before C_{fix_1}) to modify a line fixed by the fixing commit C_{fix_1}. We use the version control system (specifically, 'git blame' [2] or an equivalent) to identify C_{blamed_1} from the modified line in C_{fix_1}. Following the techniques used in previous work [31, 50, 71, 47], we consider C_{blamed_1} as a 'vulnerability introducing commit', or simply a 'vulnerable commit'. Because C_{fix_1} was made before the 'training date', meaning the vulnerability is known at time of prediction, we label C_{blamed_1} as vulnerable in the training set at commit level. At file level for the same 'training date', the file containing the line modified in C_{blamed_1} is labelled as vulnerable in Versions V_1 and V_2, and used in the training set. A file is considered to be vulnerable until it is fixed by the 'vulnerability fixing commit', and only after it is modified by the 'vulnerability introducing commit'.

On the other hand, the vulnerability fixing commit C_{fix_2} is made after the 'training date', meaning that at the time of prediction, the vulnerability may not have been discovered, and a researcher performing prediction at that time would have no knowledge of the vulnerability without access to 'future data'. For this reason, C_{fix_2} is not used to label blamed commit C_{blamed_2} as vulnerable in the training set at commit level for the given 'training date'. The file modified by C_{blamed_2} in version V_9 is also left unlabelled (by this

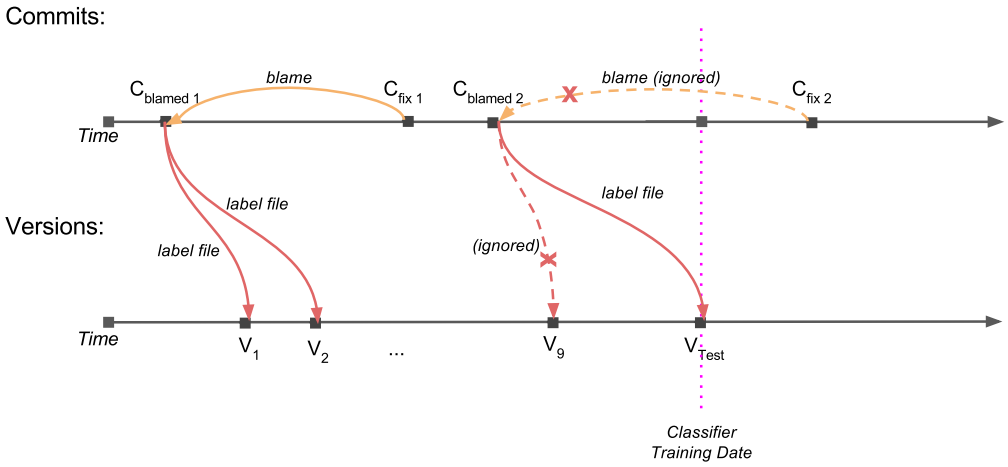

Figure 2.1: Labelling vulnerable ('buggy') instances. C_{fix_1} is a vulnerability fixing commit. A line fixed by C_{fix_1} was last modified by vulnerable commit C_{blamed_1}, which is considered vulnerable in the training set at commit level. At file level, the file containing the line in modified in C_{blamed_1} is labelled as vulnerable in Versions V_1 and V_2. The vulnerability fixing commit C_{fix_2} is made after the 'training date', and so is not used to label blamed commit C_{blamed_2} or the file modified by C_{blamed_2} in version V_9 as vulnerable in the commit-/file-level training sets. However, it is used to label the file modified by C_{blamed_2} in the *testing set* at file level, version V_{Test}.

vulnerability) in the file-level training set for the given 'training date'. However, C_{fix_2} *is* used to label the file modified by C_{blamed_2} in the *testing set* at file level, version V_{Test}.

Following this technique, we label known vulnerable instances in a projects history for a model's training set (based on a given 'training date'), and label all possible vulnerable instances in the testing set. We then use these labellings to build and evaluate our prediction models.

2.6 Feature extraction

Classification algorithms take as input a set of labelled instances, where each instance is described using a vector of *features*, or attributes of the given commit or file. We choose a set of features over a broad range of categories similar to the ones used in previous work [50, 54, 43, 63] for both commit and file levels.

1. Complexity Metrics (i.e. cyclomatic complexity, nesting, lines of code) are a set of features we use for prediction. The intuition behind this kind of features is that complex entities are more likely to be vulnerable. These metrics have been widely used in previous work [63, 43, 42].

2. Code Churn Metrics, such as the number of changes or number of lines added or deleted, have been shown to be some of the most relevant features [56, 44] for prediction, and have been extensively used in previous work, both at commit and file level [63, 43, 50].

3. Developer Activity metrics contain features related to the authors of each commit and file (e.g. author contribution). For example, it is possible that authors with a small contribution to the projects to be more prone to adding vulnerabilities, as they are not familiar with the project. These features have been used in previous work [63, 31, 73, 50].

4. Keyword features consist in counting how many times specific keywords appear in the commit or file. We use the list of keywords provided in previous work [50].

Some features need to be adapted to be used at different levels of granularity. For example, the commit-level features 'additions' and 'deletions' listed in Table 3.1 are analogous at file-level to the features 'sum additions' and 'sum deletions'. We describe these differences and how we mitigate this threat in Section 3.1.

2.7 Evaluation Measures

To evaluate the performance of the classification models in various aspects, we use four measures, or evaluation metris. In terms of practical use of vulnerability prediction models, we can consider two usage scenarios, i.e., a prediction performance scenario, and a cost effectiveness scenario. In the prediction performance scenario, the goal of prediction models

is to precisely predict as many vulnerabilities as possible. The cost effectiveness scenario aims at minimizing cost for software quality activities such as code review and testing while maximizing high detection rate of vulnerabilities by the quality activities. Thus, vulnerability prediction models often cannot be evaluated by a single measure for both scenarios.

For the prediction performance scenario, we use F_1 score [43] since it is computed by precision and recall together. However, F_1 score is known as unstable since it varies depending on probability threshold [56]. For this reason, we use AUROC (Area Under receiver operating characteristic Curve), that is independent from the probability threshold [54].

In terms of cost effectiveness, we use the area under the cost effectiveness curve (AUCEC), which plots the recall against the percentage of lines of code 'inspected' [56, 42]. Since AUCEC represents how early vulnerabilities can be detected by inspecting the entire LOC, it does not fairly estimate inspection cost between two projects with different LOC. For this reason, we also use TPR10k (Percentage of vulnerable instances in 10K Lines of Code), similar to the cost effectiveness measure used by Jiang et al. [31].

The F_1 score is calculated as the harmonic mean of the *precision* and *recall* of a classifier. Precision measures the number of true positives (tp - correctly predicted buggy instances) out of all predicted buggy instances (true and false positives), and indicates how likely a predicted buggy instance is likely to be an actual buggy instance. Recall measures the number of true positives out of all actual buggy instances (true positives and false negatives), and indicates what portion of the buggy instances were identified by the classifier. F_1 can be calculated directly from confusion matrix values as $\frac{2tp}{2tp+fp+fn}$, where fp is the number of false positives (instances incorrectly predicted as buggy), and fn is the number of false negatives (instances incorrectly predicted as clean).

AUROC is computed as the area under the receiver operating characteristic curve (ROC curve), which plots the true positive rate (recall) against the false positive rate, and visualizes the tradeoff between true positives and false positives as the threshold for prediction is varied. A randomly guessing classifier would have an AUROC of 0.5, while a perfect classifier would have an AUROC of 1.

Similarly, AUCEC is the area under the cost effectiveness curve, which plots the recall against the percentage of lines of code 'inspected'. This metric indicates how useful the model is when developers have limited time and resources and can only analyse a specific percentage of the elements flagged as vulnerable by the prediction model. Menzies et al. [42] suggest the use of AUCEC over AUROC as a metric to optimize in the context of defect prediction, in order to locate more vulnerabilities within a smaller amount of code.

The TPR10k metric is similar to the NofB20 metric used in [31], however, because the number of lines of code can vary between the test sets for commit and file level, we use an absolute number (10KLOC), rather than a percentage (20%) in order to facilitate a fair comparison. Based on a developer review rate of 200 LOC/hr (as recommended by Kemerer et al. [33]), 10kLOC would take 50 man hours, or between 1-2 weeks of inspection time.

In this section, we have outlined several key background concepts for vulnerability prediction, including how a prediction model is built using data from a training set, and then evaluated on a testing set. We first discussed how prediction can be designed to emulate a real-world use case by collecting training and testing set data following online prediction methods. We then discussed how the instances in the data are labelled as vulnerable or not vulnerable using information from NVD and the project's version control history. We outlined the categories of features that are used to represent instances in our data to the classification algorithms. Finally, we discussed several common metrics that are used in the evaluation of the performance of a classification algorithm and detail how they each capture varying aspects of a classifier's performance.

Chapter 3

Study Design

In order to compare the predictive performance and cost effectiveness of vulnerability prediction at file and commit levels, we build classifiers for both levels of granularity on the same set of projects, using an analogous set of features. We then evaluate the quality of each classifier using the metrics discussed in Section 2.

It is nontrivial to fairly compare file- and commit-level prediction since commits and files do not share a hierarchical relationship. In previous work [54, 43] comparing defect or vulnerability prediction at different levels of granularity that are hierarchically related, such as file and binary levels, a comparison can be made by selecting the set of files that makes up the binary. No such hierarchy or similar subset relationship exists between file and commit levels. Indeed, many of the features extracted, and the entities predicted on are semantically different at file and commit levels. In this section, we describe the major differences between file- and commit-level vulnerability prediction and how we proceed to design a *fair* comparison between commit and file levels. We start by describing the set of features used to describe each instance for prediction, then describe in detail our technique for selecting training and testing sets for the construction and evaluation of our predictive models. Finally, we discuss the research question which we set out to answer in our experiments.

3.1 Features

Features at file and commit levels are different because files and commits are very different entities. While a file can be considered as a chunk of code that exist in a specific snapshot

of the software, a commit represent changes made to a software. Some type of features are more easily derived for a file (e.g. complexity metrics) while other types of features are extracted from commit related information (e.g. churn features). In this section, we describe the differences between features we used at file level and commit level. Table 3.1 lists the features for commit and file levels respectively. The features are aligned by row such that features in the same row are roughly analogous at commit and file level. Note that in some cases, one commit-level feature may correspond to multiple file-level features.

3.1.1 Code Churn features

The first set of features listed in Table 3.1 represent code churn metrics at file level, or are representative of commit size at commit level. 'Additions', and 'deletions' count the number of added or deleted lines in a commit. To adapt such features at file level, we aggregate such information. For example, we sum the code churn features of all commits that modified the file in the past. Because of the assumption that higher churn (i.e., larger commits) are more likely to be vulnerable, for each file, we also keep the maximum of each attribute associated to past commits.

3.1.2 Developer Activity features

In addition to the above described features, we collect metrics that measure developer activity, similar to those used by previous work [31, 73, 50, 63]. These include the number of past changes, or distinct past authors to a file, the day of week or hour of day the commit was authored, and the percentage of total commits the author has made to the project. Because a file was likely modified at different times, we aggregate the timing features (day of the week, hour of the day, day of the months). For past change and past authors of the file, we sum the number of changes and past authors at the time of the version the file was extracted.

3.1.3 Complexity Metrics

Complexity metrics were collected using the Understand tool [3], which supports the collection of various metrics of source code files and projects, and has been used in previous studies for defect and vulnerability prediction, such as Shin et al. [63]. Most of these metrics are easy to obtain at file level for specific version of a project (e.g. max nesting,

Table 3.1: List of features extracted.

Feature type	**Commit level**	**File level**
Metadata	-	File Age
Churn/Size	Files Changed	-
Churn/Size	Additions	Sum Additions
Churn/Size		Max Additions
Churn/Size	Deletions	Sum Deletions
Churn/Size		Max Deletions
Churn/Size	Hunk Count	Sum Hunk Count
Churn/Size		Max Hunk Count
Understand (File Size)	New File LOC	LOC
Understand	New File Function Declarations	Function Declarations
Understand	New File Declarative Lines	Declarative Lines
Understand	New File Preprocessor Lines	Preprocessor Lines
Understand	New File Essential Complexity (sum)	Essential Complexity (sum)
Understand	New File Cyclomatic Complexity (sum)	Cyclomatic Complexity (sum)
Understand	New File Max Nesting	Max Nesting
Understand	New File Comment/Code Ratio	Comment/Code Ratio
Developer Activity	Sum Past Changes	Past Changes
Developer Activity	Sum Past Authors	Past Authors
Developer Activity	Authored Day of Week	Mode DOW
Developer Activity	Authored Day of Month	Mode DOM
Developer Activity	Authored Hour of Day	Mode Hour
Developer Activity	Author Contribution %	Sum Author Contribution %
Keywords	Keywords	Keywords

cyclomatic complexity), but cannot be used to measure the complexity of a commit because they require at least a full function to be computed. Because we need to work on complete files, we consider the complexity of the commit as being the complexity of the file after the commit was applied.

We do not collect dependency metrics (for example, FanIn, the number of calls in the project to functions in a file) for scalability reasons. Indeed, it would require applying the tool to the entire project for each commit to measure the number of calls. For the 8 projects under study, it would be equivalent to running Understand more than 600,000 times on complete projects.

3.1.4 Keywords

Finally, we collect counts for a set of keywords at both commit and file level. We use the same set of keywords as used in previous work [50], a list of 68 common C/C++ keywords and common standard library functions. At file level, we count how many times each keyword appears in the file. At commit level, we count how many time each keyword appears in the patch (lines added, lines deleted and context lines) and do not consider the commit message. We did not consider keywords in the commit messages because the same keyword might have a different meaning in the code and the commit message. For example, the keywords 'for' in the source code (patch or file) indicates that the code contains a loop. The semantic of this keyword in the patch message is different because these messages are written in plain English. The full list of extracted keywords can be found in Appendix E.

3.2 Training and Test Sets

In order to perform a fair comparison between granularity levels, we go to length to ensure we use an analogous set of data for training our file- and commit-level models. Figure 3.1 depicts this process, as described in detail in the following section.

In order to perform an evaluation of the performance classifier, instances are divided into a 'training' and 'testing' sets at each level, as depicted in Figure 3.1. Recall that a testing set of data separate from the data the classifier has been trained on is used to evaluate the classifier, to demonstrate the model's generalizability to other data, and to ensure that overfitting has not occurred. As mentioned earlier, because we aim to perform 'online prediction' in order to simulate real, practical use of vulnerability prediction, traditional classifier evaluation techniques such as ten-fold cross validation are not applicable to our evaluation.

Instead, we split our dataset into training and testing sets based on dates, dedicating a set of past versions (and their files) or commits as a training set, and predict on a later version or set of commits designated as the testing set. In Figure 3.1, we see that we align corresponding file-level and commit-level predictions based on a common 'training date'.

At commit level, similar to [73, 50], we train our classifier on a set of commits from before the training date (the 'Training set', as labelled in Figure 3.1, then test on the commits after the training date (the 'Test set' as labelled in Figure 3.1). At commit level, following previous work [31, 73], we also exclude a period of early commits from the training set, as early project commits and young projects may have distributions of

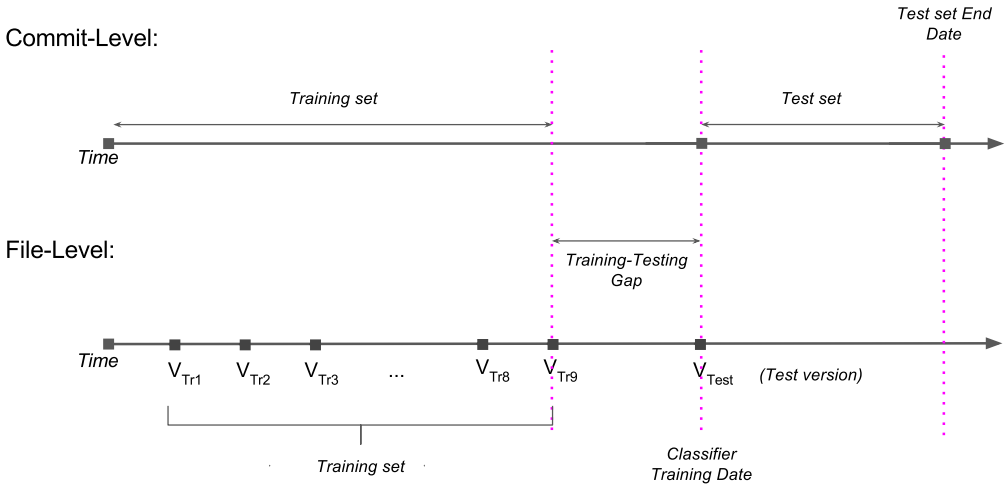

Figure 3.1: Training and testing sets for file- and commit-level prediction. The upper timeline depicts the set of commits selected for the training and testing set and used to build and evaluate the commit-level prediction models for the given 'training date'. The lower timeline depicts a set of versions chosen for the training set (V_{Tr1} to V_{Tr9}), and the version used as the test version (V_{Test}) relative to the 'training date'. Note that in both cases, we leave an identical 'training-testing gap before the training date, in order to facilitate a fair comparison.

vulnerabilities and features much different than current commit patterns, and negatively impact prediction performance. Similar to previous work [73], we exclude the first 3 years of a project's commits for projects older than 6 years, and the first 6 months of commits for projects less than 6 years old. We also exclude the commits made less than 8 months before the testing set from the training set, as these commits are likely to have a greater number of undiscovered vulnerable instances. This period is labelled as the 'Training-Testing Gap' in Figure 3.1, and the same gap is used at both commit level and file level. Finally, for the testing set, we exclude 6 months of commit data prior to our last known vulnerability (we have data up until the end of 2015), in order to exclude testing data that has a higher level of undiscovered vulnerable instances.

At file level, we use a similar technique to the *next release validation* used by Shin et al.[63], which uses several previous releases as a training set, in order to predict on an 'upcoming' release. Some key differences are that we use project 'snapshots' as opposed to releases, and that we do not use 'future vulnerabilities' (vulnerabilities discovered after the training date) in the labelling of our training versions, as discussed in Section 2.5 and Figure 2.1. We take 9 project snapshots (V_{Tr1} to V_{Tr9} in Figure 3.1), with the final project snapshot falling at the start of the 'Training-Testing Gap', in order to correspond with data collected at commit level. For our testing set, we use a snapshot of the project at the 'training date', which corresponds to time of the start of the commit-level Test set, as can be seen in Figure 3.1.

The 'Training-Testing Gap' in Figure 3.1 is used in order to omit data that has a greater number of undiscovered vulnerable instances. For instances close in time to the training date, fewer vulnerabilities have been discovered as of the training date, because it takes time to discover a vulnerability after it has been introduced. In particular, close to the training date, no 'real' vulnerabilities have yet been discovered, so all vulnerabilities in that period are undiscovered vulnerable instances. This issue is further demonstrated in Section 6.3.

In summary, training and test sets for both commit and file-level prediction are aligned. Having matching training and test sets for both file and commit level is important to ensure that (1) we use equivalent data to train both commit and file level models and (2) our evaluation of these models is done on the same time period.

Finally, while the technique of withholding data for a 'Test Set' is used to evaluate the generalizability of a given predictive model, we also want to test the generalizability of the classifier or prediction algorithm's performance to other data. In order to do this, we repeat our experiments across 8 different 'training dates', keeping file- and commit-level data aligned for each repetition of our experiment. Note that at each date, we build a new prediction model using that date's training set, and evaluate it on that date's testing set (at both commit and file level). Since the goal of our study is to systematically compare commit- and file-level vulnerability prediction, we conduct our experiments under various settings. Using different training dates affects the size of training and test sets, feature values, and labels of training sets. If prediction results under various situations show similar trend, it would be more helpful to generalize the conclusion of our experiments.

3.3 Research Question

Using the above described fair comparison, we set out to answer the following research question, with the goal of informing the development of vulnerability prediction towards practical use:

- How do commit-level and file-level 'online' vulnerability prediction compare in terms of 'predictive performance' (F_1 and AUROC) and 'cost effectiveness' (AUCEC and TPR10k)?

Recall that the aforementioned metrics are described in detail in Section 2.7. We divide the metrics into two usage scenarios, a 'predictive performance' or 'classification performance' scenario, and a 'cost effectiveness' scenario. The 'predictive performance' metrics measure the quality of prediction given that all predicted positive instances are inspected for vulnerabilities. These metrics may be useful in a security review setting, where all predicted instances are of interest, and resources exist to examine them further.

The 'cost effectiveness' metrics allow us to compare classification performance based on the number of vulnerabilities found after looking at a given amount of code. In particular, TPR10k was chosen as an evaluation metric in order to make a comparison that is independent of the total number of lines of code, which may vary between file and commit level. These 'cost effectiveness' metrics may be useful in a managerial context, when a limited or set amount of review resources are available to inspect code for vulnerabilities, and only the top n predicted lines of code can be examined. Taken together, these metrics provide different insights on the effectiveness of the vulnerability prediction techniques we are evaluating.

In this section, we have discussed in detail the design of our approach for comparing commit-level and file-level vulnerability prediction, highlighting how we design for a *fair* comparison between the two levels of granularity. We first detailed the set of features used to describe each instance for prediction, and how we choose an analogous set of features at each level of granularity. We then described in detail our technique for selecting analogous training and testing sets at commit and file levels. Finally, we have discussed the research question that we set out to answer in our experiments: how do commit- and file-level vulnerability prediction compare under several different metrics, and different usage scenarios?

Chapter 4

Experimental Setup

We train and evaluate classifiers for 7 different open-source C/C++ projects. For each of those projects, we evaluate on up to 8 different train-test splits (limited in some cases by project age and vulnerability report availability). In the following sections we discuss in detail the experimental setup, parameters, and data we used for our evaluation. We first describe the different projects on which we performed our evaluation, and provide some information on their size and buggy rate. We then detail the classification algorithms used in our evaluation, as well as the tuned parameters for each. We discuss the data preprocessing steps we applied to our training data in order to improve classification performance. Finally, we discuss the implementation of the tool used to collect data and perform our evaluation, as well as the runtime of several key steps in data collection and classifier training.

4.1 Projects under study

The 7 projects we examine in this study are listed in table 4.1. Columns 2 and 3 indicate the number of files and lines of code as of Jan 1, 2013 in each project. Columns 5 and 6 indicate the number of commits to the project between Jan 1, 2013 and Jul 1, 2015, and the LOC in those commits. Columns 4 and 7 indicate the buggy (vulnerable instance) rate of the project at file and commit level in the aforementioned version and commit period, labelled using data up until the end of 2015. Projects were selected from the set of open-source, C/C++ projects known to contain vulnerabilities in the NVD. eters

Table 4.1: Projects under study.

Project	File LOC	File Count	File Vuln. Rate	Commit LOC	Commit Count	Commit Vuln. Rate
FFmpeg	467,990	1,958	7.2%	1,273,963	31,394	0.5%
Apache httpd	70,013	274	5.5%	888,852	6,659	3.2%
Kerberos	269,782	1,502	2.2%	171,795	1,493	1.5%
OpenSSL	136,782	662	6.4%	3,585,079	5,789	3.8%
PHP	213,188	647	4.3%	6,020,445	18,729	2.0%
Wireshark	783,287	1,098	5.2%	4,977,254	16,879	2.3%
Xen	326,410	1,550	7.1%	704,390	6,315	3.0%

Table 4.2: Classifiers under study.

Classifier	**Tuned parameters**
Naive Bayes	None
Logistic Regression	Ridge parameter
Multilayer Perceptron	Learning rate, momentum, number of layers/nodes per layer
ADTree	Number of boosting iterations
J48 Decision Tree	Pruning confidence
Random Forest	Number of attributes to randomly investigate

4.2 Classification algorithms and tuning

We evaluate and compare the performance of several commonly used machine learning algorithms on each of our 7 projects. We rely mainly of the implementations of these algorithms as provided by Weka [24], an open-source java implementation of many common machine learning algorithms and related tools. Weka classifier implementations have been used in previous defect prediction work [31]. In order to tune the parameters of each classifier, we randomly select one project and use Weka's MultiSearch tool, which extends the grid search technique to more than 2 parameters. MultiSearch takes a list of parameters to be tuned and a set of possible values for those parameters (e.g., for numerical values: min, max, step), and evaluates the classifier for each possible combination of those parameters, selecting the set of values that provided the best performance in terms of F_1 score. The list of algorithms used, as well as the parameters tuned, are listed in Table 4.2. We tune a separate set of parameters for both file and commit levels because we want the classifiers to perform optimally in both cases. Compared to previous work [50], we do not show results for Support Vector Machine (SVM) because, for all projects, all metrics, and both levels of granularity, this classifier performed poorly in our experimental setting.

4.3 Preprocessing steps

Before training our classifiers, we apply a number of preprocessing steps to the training data in order to improve classification performance. Vulnerability prediction suffers to a significant degree from imbalanced data, as can be seen in Table 4.1, where positive instance rates are at 7.2%, and lower at file level, and 3.8% or lower at commit level. In addition, imbalanced data in the training set is made worse when building and evaluating a model using online prediction [73]. Because of this, we use a combination of undersampling of the negative (majority) class, and SMOTE (synthetic minority oversampling) on the positive class, as performed in [10]. In this way, the training data is resampled to contain an artificially higher number of positive instances. We tune the extent undersampling for file and commit levels individually by taking the highest F_1 score on a randomly selected project, averaged over all classifiers using their default parameters. Parameters for each of the classifiers were tuned after choosing the optimal sampling parameters.

Finally, before training each of the classifiers, we perform attribute selection on the training data, using forward feature selection in order to select a subset of features that provides a high information gain on the instances in the training data. We perform attribute selection separately for commit and file levels, giving us a separate set of optimized attributes at each level. This subset of features is then used to train and evaluate the classifiers.

4.4 Implementation and Runtime

In order to implement our evaluation, we developed a series of scripts, built primarily in python, to collect feature and vulnerability fixing data, produce feature vectors and labelling for given training dates, and preprocess, train, test, and evaluate the classification models. In this section, we discuss the implementation details and high-level functionality of the tool we built to perform and evaluate vulnerability prediction at commit and file levels. We first *collect data* on commits, files, and vulnerabilities from the project source repositories and NVD for calculating feature vectors and labellings. We then *generate training and testing sets*, calculating feature vectors and correct labellings for a set of training date parameters. Finally, we *train and test* our classification models, also preprocessing data and calculating evaluation metrics in these steps.

Our scripts first *collect data* from NVD and project source repositories for labelling instances and calculating features. NVD provides the contents of its database in a series

of XML documents, which we parse in order to obtain vulnerability information. We then traverse each of our projects' commit histories using the projects' version control system, identifying fixing commits, finding vulnerability introducing commits (as described in Section 2.5), and recording data for the calculation of features. The relevant data is stored in an SQLite database for more convenient access for calculating training and testing sets. The data includes the commit identifiers, commit date, files changed, vulnerability fixing information, meta information used in calculating features, and blame information for identifying vulnerability introducing commits and vulnerable files. We separately generate complexity metrics using the Understand C++ tool [3], which are incorporated when calculating feature vectors for our training and testing sets.

Once the data has been collected and stored to our SQLite database, we then *generate training and testing sets*. In order to do this, we query the database for a given a training date and granularity level to calculate the feature vectors and labels for the training and testing sets. As necessary, we aggregate commit-related metrics to their matching file-level metrics, and vice versa when calculating the feature vectors. When aggregating commit-level features for a file, we examine the set of commits that have previously modified that file. When aggregating file-level features to a commit, we examine the set of files modified by that commit as of the date the commit was made. We also have scripts for automatically generating configuration parameters (start and end for training and testing sets) for a given training date. We also incorporate features calculated separately using the Understand C++ tool here.

Finally we use a series of scripts, written in bash and python, in order to *train and test* our classification models, using classification algorithms implemented in Weka [24]. We first perform preprocessing on the training data (as described above in Section 4.3). We then train each of our classifiers using each of our classification algorithms, and test the models by using them to predict on the data in the testing sets Finally, we calculate the performance metrics based on the predicted results.

We ran our experiments on a 3.3GHz E5-1660 shared server with 12 logical cores, 6 physical cores, and 32GB memory. Complexity metrics from Understand C++ were run on a 2.93GHz Quad Core Intel i7 machine with and 12GB memory, dedicated for Understand C++ for the duration of the data collection process. The runtime for the entire process for all projects was on the order of weeks. The majority of that runtime cost was the collection of complexity features using Understand C++. We break down the total runtime cost into the costs for *collecting data*, *generating training and testing sets*, and *training and testing* each below. For each stage, we first report the total runtime cost, then break down the runtime costs of the steps that make up each stage.

Collecting data took on the order of weeks. Again, the majority of this cost was caused by the collection of complexity metrics with Understand C++, which took on the order of days to weeks for each project. We note that we calculate metrics for changed files after *each commit in a project's history* in order to generate complexity metrics with Understand C++. To the best of our knowledge, Understand C++ does not support the incremental analysis of changes below the level of a file. We ran multiple projects in parallel, with the longest project taking an estimated 2 weeks to complete. Collecting other features for each commit from the project repositories took approximately 4 hours for all projects, based on timing information recorded in log files. Identifying vulnerability fixing commits, including parsing NVD data and dumped history of commit log messages, and identifying vulnerability introducing commits took under an hour for all projects. Outputting the commit log messages and commit data from the project's version control system for further parsing took on the order of a day for the largest projects.

Generating training and testing sets for each project-prediction-date combination took under a day. Calculating feature vectors from previously collected data for all 8 training dates and 7 projects took approximately 8 hours for commit-level features, and approximately 2 hours for file-level features. Calculating labels from previously collected data for all 8 training dates and 7 projects took just under 2 hours each for commit and file levels.

Training and testing each of the 6 classifiers for 8 training dates and 7 projects took on the order of days. Training each individual classification model took on the order of minutes for most classifiers, except for the Multilayer Perceptron models, which took on the order of hours to train. Testing and evaluating each classifier takes on the order of minutes to test. Again, the whole process of our evaluation, from collecting data to training and testing took on the order of weeks.

In this section, we have discussed our experimental setup, projects, parameters, implementation and runtime. We have described the projects and classification algorithms used, and the preprocessing steps applied in our evaluation. Finally, we have discussed the implementation and runtime of the tool used to perform the evaluation.

Chapter 5

Findings

In this project, we seek to compare vulnerability prediction at commit level with prediction at file level under different usage scenarios. We set out to compare commit-level and file-level prediction under two usage scenarios, 'prediction performance', and 'cost effectiveness'. In order to do so, we have performed online vulnerability prediction across 7 different projects, using 6 different classifiers, and using 8 different training dates (for a total of 336 runs per level). Here we present a comparison of the series of runs using each project and classifier. We find that generally file-level prediction outperforms commit-level prediction for most projects, especially as measured by F_1, and TPR10k metrics. There are some notable exceptions, however. In particular, commit-level prediction often outperforms file-level prediction in AUROC and AUCEC metrics for the PHP project. Finally, we show the ROC and CEC curves for specific prediction runs, and demonstrate that below certain thresholds, commit level can occasionally outperform file level in both the ROC ('prediction performance') curve, and the CEC ('cost effectiveness') curve.

5.1 Comparison of prediction performance and cost effectiveness

Figure 5.1 shows the box plots for each classification performance metric, comparing commit level and file level across each of the 7 projects, for the Naive Bayes classifier. We select the Naive Bayes Classifier to show here, as it achieves the highest average F_1 score for all projects in the Jan 2013 test date at file level, but similar plots of other classifiers can be seen in Appendix A. The y-axes of the box plots measure the scores for each metric. The

plot is split up into 7 sections, depicting the performance within each project, and each project has four pairs of box plots, each comparing commit-level and file-level performance for the same metric. The four metrics presented on the plot are F_1 score, AUROC, AUCEC, and TPR10k, in that order. For all measures, a higher score indicates better performance. Each box represents the 8 runs for each test date at that repo/classifier setup. On each box plot, the minimum and maximum results (excluding outliers) are represented by the extremities of the tails. Outlier results are represented by a dot. The top and bottom lines in a box of each plot show performance results at first and third quartiles respectively, while the solid line in the box represents the median. Similar box plots for each of the other 6 classifiers can be found in Appendix A.

In each of the projects, comparing the median performance of each metric at commit level and file level, we find that, except in certain cases discussed below (for example the AUROC score for the PHP project in Figure 5.1, file-level prediction outperforms commit-level prediction. As such, we report the following finding:

File-level prediction tends to outperform commit-level prediction in both 'predictive performance' and 'cost effectiveness' metrics, across most projects and classifiers.

Our results indicate that despite being a finer granularity, commit-level prediction is not more cost effective than file-level prediction for most projects and classifiers. Cf. Posnett et al. [54], who find that when comparing the hierarchically-related granularity levels of file and package level, the finer-grained file-level vulnerability classification can be shown to be more cost effective. However, commit- and file-level prediction are not hierarchically related in the same way; as discussed in Section 3, there are differences in the meanings and distributions of corresponding prediction features.

However, we also find that for F_1 and TPR10k, consistently have median scores that are quite low. For no classifier/project pair does the median F_1 score reach above 0.5, similarly most median TPR10k scores are around 0.1 or lower. These results correspond with Morrison et al.'s [43] discussion that the current state of the art in vulnerability prediction is not effective enough that it is widely used in industry. However, based on our results, for most projects, focusing on the development of features and techniques for file-level prediction would be more promising.

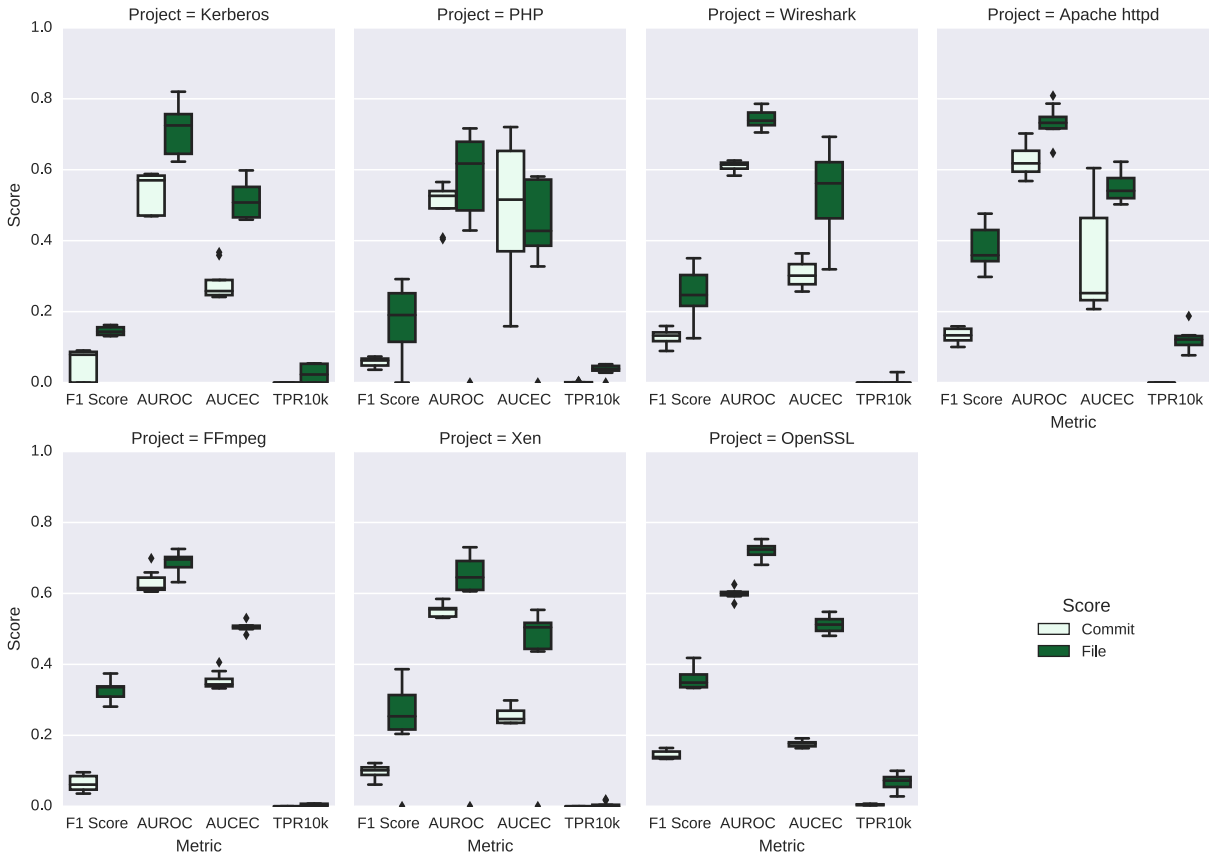

Figure 5.1: Naive Bayes performances by project, level, and metric.

5.2 Cases with higher commit-level performance

As discussed above, while file-level prediction tends to outperform commit-level prediction for most projects/classifiers and metrics. However, in this section, we highlight cases where *commit-level prediction* outperforms file-level prediction. Again, we compare median values in the box plots in Figure 5.1 and Appendix A. Most notably, the PHP project consistently has stronger median AUROC scores at commit level across 5 of the 6 classifiers, and has stronger median AUCEC scores at commit level for 4 of the 6 classifiers, Random Forest, J48, and ADTree. PHP also has a higher F_1 score at commit level for the J48 classifier. Additionally, Apache httpd and OpenSSL achieve higher median AUCEC scores at commit level with 2 classifiers (Multilayer Perceptron and Logistic Regression), while Xen has

a higher median AUROC score with 3 classifiers (ADTree, Multilayer Perceptron, and Logistic Regression).

Note also the J48 classifier, which at file level had the lowest average F_1 score across all projects for the Jan 2013 test date, while at commit level had the second highest average F_1 score across projects for the same date. This classifier also had the highest number of projects which had higher scores at *commit level* based on the AUCEC, AUROC or F_1 metrics. As mentioned briefly above, the J48 classifier had higher median commit-level scores for the following project/metric combinations: PHP and Apache httpd for the AUCEC metric; PHP, Apache httpd, OpenSSL, Kerberos, and Wireshark for the AUROC metric, and PHP for the F_1 score.

Based on these cases, in which commit-level prediction sometimes outperforms file-level prediction under certain metrics and for certain projects, we report the following findings:

> *For certain projects, particularly PHP, and for certain classifiers, particularly J48, AUROC and AUCEC performance at commit level may be comparable to, or better than, performance at file level.*

In these cases, for these projects, and using these classifiers, it may be beneficial to perform vulnerability prediction at commit level rather than file level, depending on the usage scenario, and more specifically, metric, that the user is interested in optimizing.

5.3 AUROC vs. AUCEC

Figures 5.2 and 5.3 show the ROC and CEC curves at commit and file levels for the Naive Bayes classifier using the Jan 2013 test date, and for the PHP and FFmpeg projects respectively. There are 4 curves in total on each plot, two curves plotting the ROC and CEC curves at commit level, and two curves plotting the same at file level. For all curves, the y-axis value represents the true positive rate, or recall. The recall is measured as the number of true positive predictions over the total number of vulnerable instances. For ROC curves, represented on these plots by solid lines, the x-axis of the plot indicates the false positive rate (number of false positives over total number of non-vulnerable instances). The dotted black line represents the ROC curve of a classifier which randomly predicts instances as buggy or clean. On the other hand, for the CEC curves, represented on these plots by dashed lines, the x-axis represents the total percentage of lines of code of

all instances predicted as vulnerable. For both of these values, the curve is calculated as the prediction threshold is brought from 1 (accepting instances only with 100% predicted certainty) to 0 (accepting all instances). At each threshold level, the x and y-axis values are calculated and plotted based on the predicted probability of being vulnerable as assigned by the classifier.

Note that in some curves, sections of the curve appear perfectly linear. In these cases, all of the instances (or lines of code) in the linear section of the curve were predicted with the same probability. The x-value at the end of the linear section has increased by the cumulative x-value of those instances, while the y-value has increased by the cumulative y-value. Recall that AUROC measures the area under the Receiver Operating Characteristic curve, which plots the true positive rate (recall) against the false positive rate, while CEC plots recall against the percentage lines of code inspected.

For the PHP project, with the Naive Bayes classifier, using the Jan 2013 test date (the run which Figure 5.2 plots the results for), we note that *file-level* AUROC has a score 0.09 higher than commit-level prediction, while AUCEC has a score 0.39 higher at *commit level*. For FFmpeg, for the same classifier and test date, AUROC has a score 0.07 higher at *file level* while AUCEC has a score 0.16 higher, also at *file level*. The curves for the remaining 5 projects for the Naive Bayes classifier for the Jan 2013 training date can be found in Appendix B

To understand why the results of PHP contradict those of other projects, we investigate the characteristics of PHP. We found that the training set used for PHP at commit level is 1.7 times larger than for other projects (Table 4.1). This is due to the large number of commits in the long history of PHP. It is possible that this large number of commits in the training set significantly helps commit-level prediction. However, further studies need to be done to confirm this observation.

In the two curves in Figures 5.2 and 5.3, we note that below a certain small threshold, the ROC or CEC curve shows a higher performance for one level of granularity, but overall has a higher AUC for the other. In particular, for the PHP CEC curves, below a %LOC of just under 20%, *file-level* cost effectiveness outperforms commit-level, while overall, AUCEC is higher for *commit level*. On the other hand, for the FFmpeg ROC curves, below a false positive rate of less than 5%, *commit-level* ROC outperforms file-level ROC, while overall, AUROC for *file level* is better. In light of this, we report the following finding:

For certain projects and classifiers, the ROC or CEC curves may show a higher performance for one level of prediction below a given threshold, but overall have a higher AUC for another.

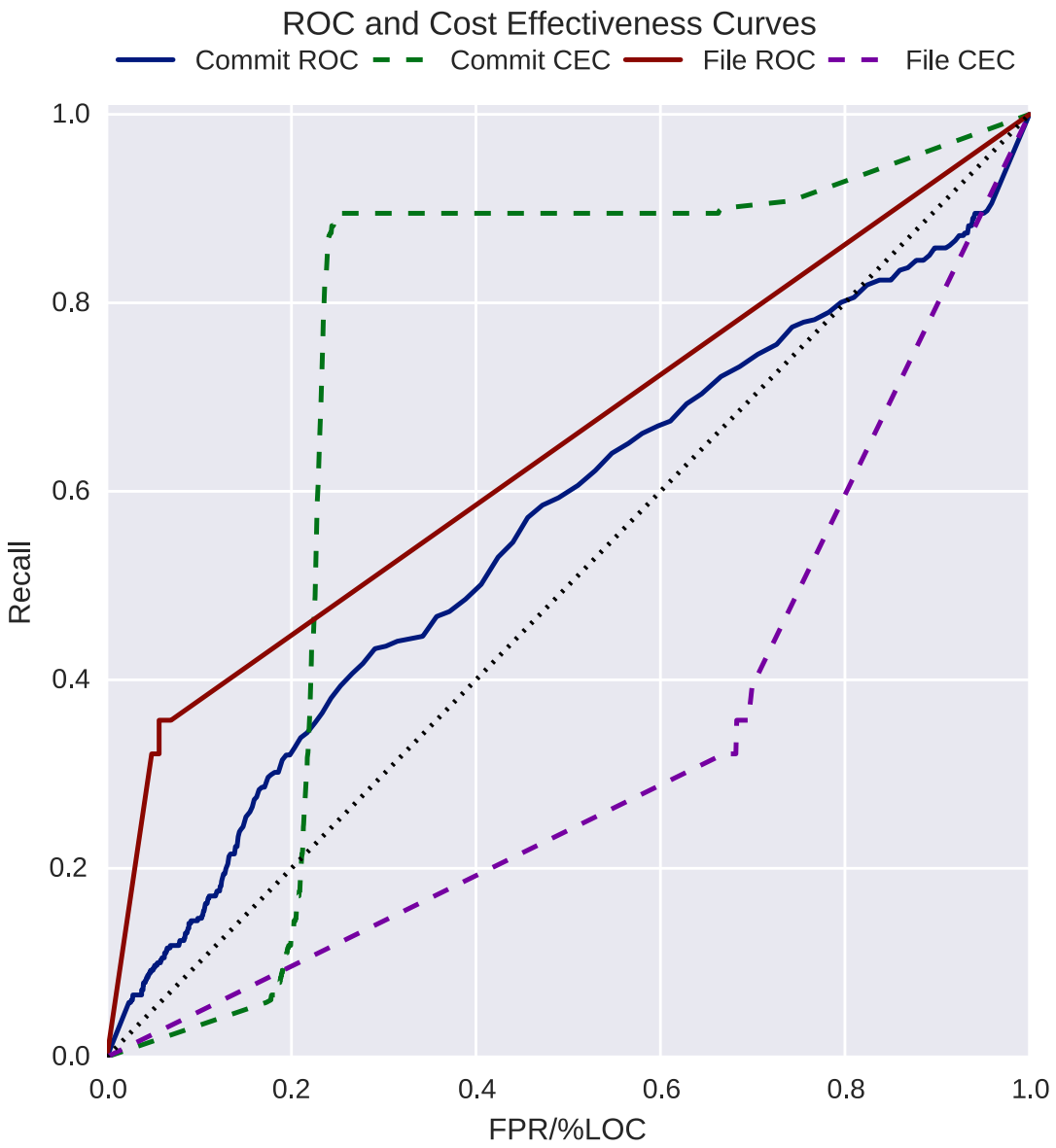

Figure 5.2: ROC and CE curves, commit and file level, for PHP, Naive Bayes classifier, and Jan 2013 test date.

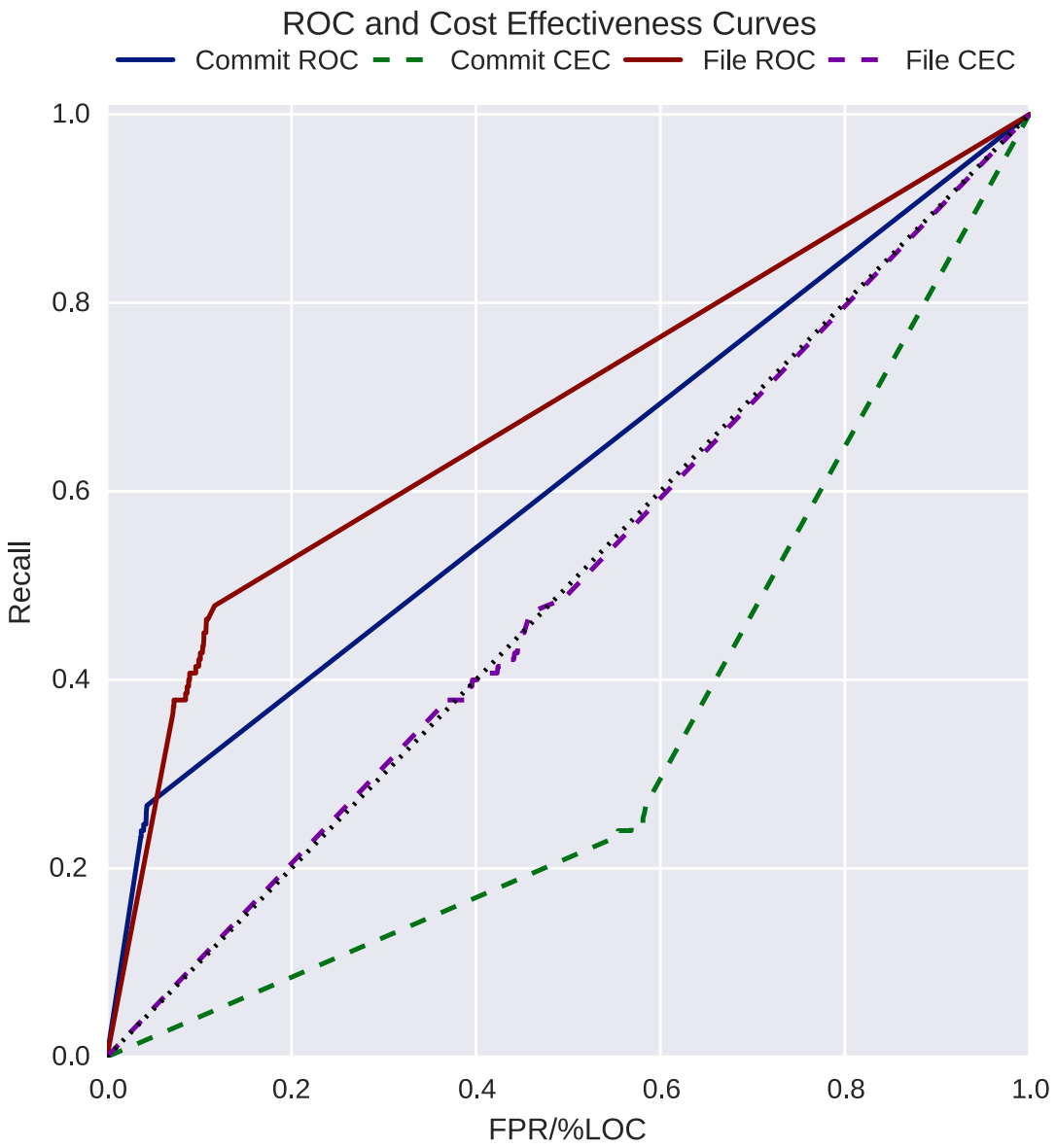

Figure 5.3: ROC and CE curves, commit and file level, for FFmpeg, Naive Bayes classifier, and Jan 2013 test date.

Chapter 6

Prediction Data Analysis

In this section, we discuss the data used for vulnerability prediction in order to allow for further insight into our methods and findings. First, we discuss in detail our technique for identifying vulnerability fixing commits, by matching fixing commits with the vulnerabilities they were written to fix, as recorded in the National Vulnerability Database (NVD). Next, we present the distribution of Common Weakness Enumeration (CWE) types within our data, which describe the kinds of vulnerabilities available for use in our evaluation. We also present data on the distribution of vulnerability fixing times in our data. Finally, we present a pairwise scatterplot showing the distribution of instances in two-item subsets of the feature space, allowing us to visualize the discriminative power of pairs of features.

6.1 Finding vulnerability fixing commits

Table 6.1 details the number of vulnerabilities for each project, the number of vulnerability fixing commits identified using each of the three techniques, as well as the number of matched NVD vulnerabilities, using data from NVD up until the end of 2015. Column 2 shows the number of vulnerabilities identified as affecting the project by NVD. Columns 3-5 specify the number of vulnerabilities matched with one or more fixing commits using the following three techniques, respectively:

(a) Some of the vulnerability fixing commits can be matched by identifying commit SHA identifiers in NVD external reference URLs. For each project, a set of project-specific regular expressions was used to identify NVD references to the project repository's

Table 6.1: Labelling Vulnerable Instances.

Project	NVD Vulns	Method (a)	Method (b)	Method (c)	Total Vuln Fix Commits	Total Matched Vulns
FFmpeg	165	157	338	118	556	163
Apache httpd	176	0	197	20	213	63
Kerberos	96	8	192	18	196	73
OpenSSL	84	7	375	21	397	95
PHP	355	18	135	69	212	61
Wireshark	261	0	1	484	485	150
Xen	89	3	456	0	456	112

web interface (if present), and extract the SHA identifier of the commit referenced. Previous work [50] uses this technique, as well as the next to identify vulnerable commits in their data.

(b) Vulnerability fixing commits can also be identified by searching references to the CVE identifiers (Common Vulnerabilities and Exposures, the identifiers used by NVD to uniquely identify a vulnerability) in commit messages.

(c) Finally, we identify vulnerability fixing commits by matching bug IDs referenced in an NVD external reference URL to a bug ID mentioned in a commit message, similar to the techniques used in previous work [31, 47, 43, 63]. We again use project-specific regular expressions to identify bug IDs from NVD references to a project's bug reporting system. We then use project-specific regular expressions, built based on manual examination of a project's contributor documentation and past commit messages, to identify references to bug IDs in the commit messages. We identify references to bug IDs that indicate that the given commit was written to fix the referenced bug. We then consider that commit to be a fixing commit for that vulnerability.

Finally, column 6 shows the total number of vulnerability fixing commits identified using the three combined methods (note that some methods may identify the same commit), and column 7 shows the total number of vulnerabilities used for those matches. The remaining vulnerabilities we were not able to match with a fixing commit using our automated technique. Note that in some cases, the number of matched vulnerabilities in column 7 may exceed the number of vulnerabilities listed as affecting the project in column 2. This is caused by the fact that we collect data from commit messages after the end of 2015, so the vulnerabilities identified in these commits are not in the NVD data we include.

Note that for this process, merge commits matched with vulnerabilities are ignored. Note also that in each case, multiple bugs or commits may be identified for each vulnerability.

We find that different matching methods tend to work better for different projects. For example, looking at Table 6.1, we find that Method (a) is able to identify a significant number of commits as vulnerability fixing commits for FFmpeg only, while methods (b) and (c) are able to identify the majority of vulnerability fixing commits for the Xen and Wireshark projects, respectively. Because our labelling heuristics rely on commit log message and bug reporting conventions followed by developers, or on the availability and completeness of external resource links provided by NVD, effectively identifying vulnerabilities requires different techniques for different projects. In this case, NVD is relatively consistent in linking to patch information for FFmpeg. FFmpeg and Xen developers are extremely consistent at referencing CVE identifiers in NVD when committing fixes for these vulnerabilities. On the other hand, for Wireshark, consistency from both developers and NVD allows us to identify the majority of the vulnerability fixing commits available for our study. Wireshark developers are consistent at reporting the bug id in fixing commit messages, and NVD is relatively consistent at including a link to the bug report with its vulnerability entry. To the best of our knowledge, we are the only project that has developed project-specific heuristics and used multiple matching techniques for matching NVD vulnerabilities to their fixing commits.

Improvement in consistency in providing links to patches or bug reports in NVD, or in consistency in reporting CVE identifiers in commit message logs by developers would reduce the need for manually tuned per-project heuristics, and facilitate a more accurate evaluation and development of vulnerability prediction methods such as ours. However, note that the ability to link a vulnerability to its bug report and patch can reveal sensitive information to would-be attackers if it is done so prior to a patch being released. Development of improved labelling techniques in order to facilitate the development of vulnerability prediction methods is discussed as a possible direction for future work in Section 9.1.

Finally, we recognize that our technique, despite using a combination of methods in order to automatically identify fixing commits, is likely to produce both false positives (label fixing commits that do not fix vulnerabilities), and false negatives (miss commits that do fix vulnerabilities). In the process of manually developing heuristics to identify bug id numbers and CVE vulnerabilities, care was taken to reduce the number of false positives (i.e. looking for 'bug', 'issue' strings before id numbers based on project-specific conventions). In addition, undiscovered vulnerabilities are not able to be matched. However, our method combines multiple vulnerability matching techniques used by existing work in vulnerability prediction [47, 63, 50, 71].

6.2 Types of vulnerabilities found in projects

Table 6.2 presents a list of the types and number of vulnerabilities present in our data. CWE labels are used by NVD to specify the type of a given vulnerability (e.g., buffer overflow, SQL injection). The first column specifies the CWE id, and the type description as specified by NVD [49]. The second column specifies the number of distinct vulnerabilities of that type in each project, and the third column specifies the number of vulnerability fixing commits matched with that type of vulnerability. Note that due to the qualitative and subjective nature of CWE type labelling, as well as the hierarchical nature of the CWE taxonomy, CWE types may overlap, meaning vulnerabilities may fall into multiple categories. For example, CWE-18 (Source Code), CWE-19 (Data Handling), CWE-20 (Input Validation), and CWE-94 (Code Injection) have a hierarchical relationship, with CWE-94 being the most specific classification in that set. NVD lists at most 1 CWE type per vulnerability.

Appendix C breaks down the CWE type distribution by project. We have selected a variety of projects with a variety of different applications in order to get a sense of the effectiveness of prediction of general vulnerabilities, rather than vulnerabilities of any one specific type. Shar et al. [61] achieve a quite high predictive performance by developing features specific to particular vulnerability types (SQL injections and cross site scripting attacks). Directions for future work on the development of vulnerability-specific features and vulnerability-specific prediction is discussed in Section 9.1.

While we focus on open-source C/C++ projects, our technique is not restricted to such projects, and should be generalizable to any project and type of vulnerability available in NVD. However, certain vulnerabilities may be less common in projects in different languages (e.g., buffer overflow vulnerabilities may be less common in Java projects), as discussed in 7

6.3 Vulnerability time to fix

Figure 6.1 shows the distribution of the time to fix for the vulnerabilities encountered in our data. The histogram shows the probability density of the time to fix a vulnerability in years. The estimated probability density function is represented by the dashed line, while the cumulative density estimate is represented by the solid line. We measure the time to fix as the time from the date of the *vulnerability introducing commit* (any commit blamed by the changes in a vulnerability fixing commit, as described in section2.5) to the vulnerability

Table 6.2: Vulnerability CWE types

CWE type	Vuln. count	Fix commit count
No Type Listed	206	638
CWE-16: Configuration	7	24
CWE-17: Code	17	57
CWE-18: Source Code	2	18
CWE-19: Data Handling	8	32
CWE-20: Input Validation	234	766
CWE-22: Path Traversal	3	3
CWE-59: Link Following	2	3
CWE-79: Cross-Site Scripting (XSS)	4	13
CWE-94: Code Injection	10	36
CWE-119: Buffer Errors	242	821
CWE-134: Format String Vulnerability	3	7
CWE-189: Numeric Errors	125	350
CWE-200: Information Leak / Disclosure	31	80
CWE-254: Security Features	3	19
CWE-255: Credentials Management	2	3
CWE-284: Improper Access Control	1	4
CWE-287: Authentication Issues	7	6
CWE-310: Cryptographic Issues	34	140
CWE-352: Cross-Site Request Forgery (CSRF)	1	3
CWE-362: Race Conditions	12	44
CWE-399: Resource Management Errors	140	426

fixing commit (as identified by our technique described above in Section 6.1). Following the techniques of previous work [31, 50, 71], we consider commits blamed by a fixing commit (that were made before the fixing commit) as vulnerable, so we may have multiple fixing times for a given vulnerability fixing commit. We can consider the vulnerability time to fix a lower bound on the vulnerability time to discovery, since vulnerabilities must first be discovered before they are fixed, however recall that we require a vulnerability fixing commit in order to label vulnerabilities using our technique. Note that it is not until the 10 year mark that about 90% of the vulnerabilities in our dataset are likely to have been fixed. However several of our projects do not have a significant number of discovered vulnerabilities listed in NVD until 2013, meaning that if we attempted to perform online prediction before 2013, the project's training set would have little to no positive instances,

making prediction impossible. Our choice of training dates attempts to account for the availability of vulnerabilities for the training set (making prediction possible), while also including later fixed vulnerabilities in the testing set. A further breakdown of vulnerability times to fix by project can be found in Appendix D.

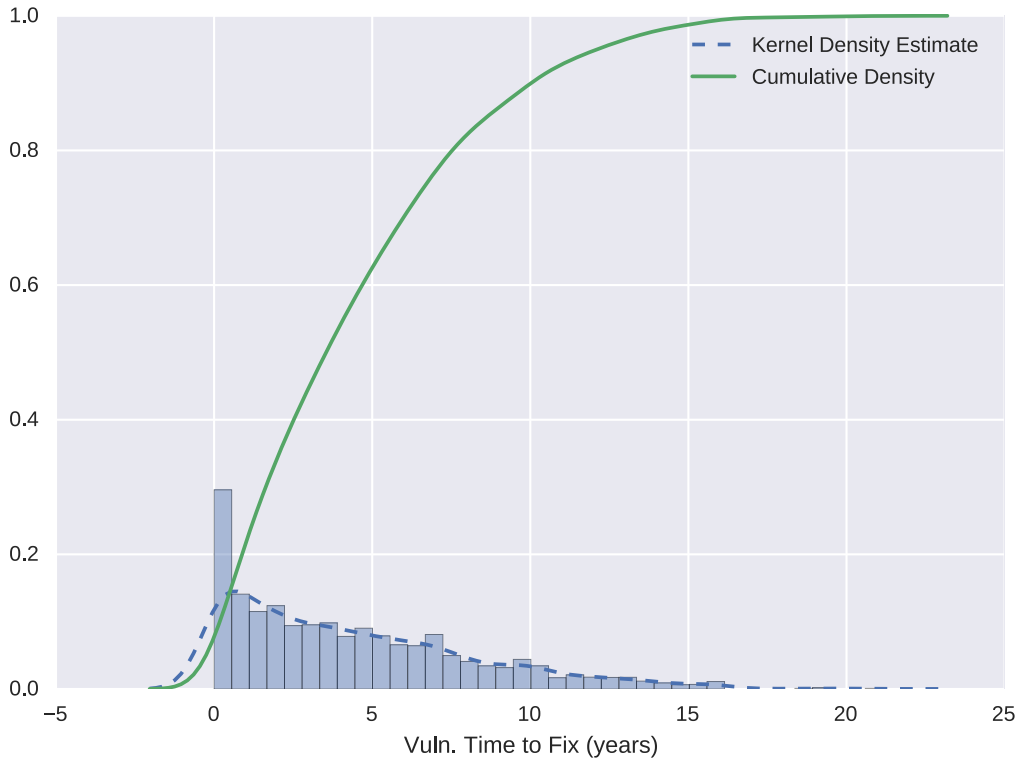

Figure 6.1: Vulnerability times to fix in years, showing both density and cumulative density estimation.

6.4 Sample feature distributions

Figures 6.2 and 6.3 show the paired distribution of a handful of selected attributes at commit and file level. The attributes shown are analogous attributes that were chosen in the attribute selection preprocessing step at commit and file level, respectively. Each of the

plots in the upper right and lower left corners of the grid is a scatterplot of two attributes (or an attribute and the 'vulnerable' label), showing the distribution of both vulnerable and non-vulnerable instances in those attributes. Note that the same 4 attributes are listed along the x and y axes of the plot, allowing a pairwise comparison of the distributions. The 'label' attribute is the vulnerability label used as the predicted variable for the classifier, that is, it indicates whether the instance contains or does not contain a known vulnerability. The label of each instance is also represented on each of the scatterplots as either a red triangle for 'vulnerable' instances, or a blue 'x' for instances not known to contain a vulnerability. The diagonal row of histograms show the distribution of values for the attribute represented by that row and column of plots.

For example, consider the top row of plots in the commit-level paired plots, showing the `hunk_count` attribute relative to other attributes. The first plot shows a histogram depicting the distribution of instances over values of `hunk_count`. Each subsequent plot is a scatterplot of instances, with the y-axis indicating the value of `hunk_count`, and the x-axis indicating the value of the other paired attribute for that instance. The final plot in this row shows a row of vulnerable instances, and a row of unlabelled instances as they are distributed in the values of the `hunk_count` attribute.

For the commit-level plot, the attributes included are: `hunk_count`, the number of 'hunks' output by the diff of the commit under default diff parameters [1]; `past_changes`, the number of changes to the project made prior to the given commit; and the `author_contributions_percent`, the percentage of commits made to the project by the author of that commit, calculated as of the 'training date'. Note that the outlying points at an author contribution percent of 30% are likely all commits made by the same author. Finally, the rightmost row and bottom column depict the distributions of the vulnerability label. For the file-level plot, the attributes included are: `sum_hunk_count`, the sum of the number of 'hunks' over all past commits made to the file; `age`, the age of the file, measured in seconds; and the `author_contributions_percent`, the cumulative percentage of commits made to the project by the set of authors who modified that file, calculated as of the 'training date'. Again, the rightmost row and bottom column depict the distributions of the vulnerability label.

The plots allow the visualization of the discriminative power of these select attributes and attribute pairs. Note that for these attributes, there is no distribution that appears to be particularly discriminative for vulnerable files or commits, perhaps pointing to the need for better, more security-specific features. The development of security-specific attributes for the improvement of vulnerability prediction is left as future work.

In this section, we have discussed the data colleced in the process of our evaluation. We first discussed in detail our technique for identifying vulnerability fixing commits, by matching fixing commits vulnerabilities in NVD. We then presented the distribution of different types of vulnerabilities within our data. We also presented information on the distribution of vulnerability fixing times in our data. Finally, we presented pairwise scatterplots showing the distribution of instances in two-item subsets of our feature space, for the visualization the discriminative power of pairs of features.

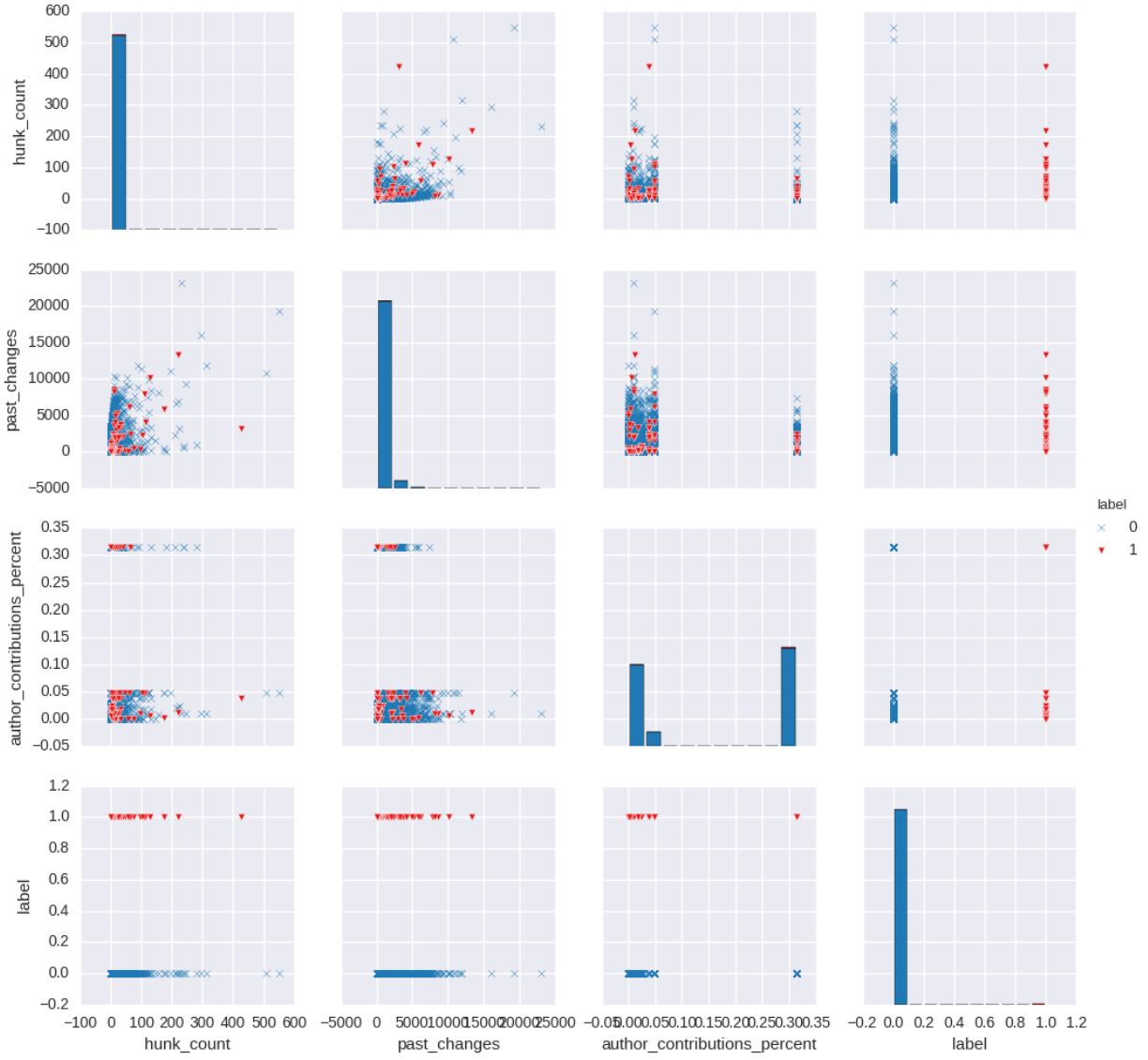

Figure 6.2: Paired plots showing the paired distribution of clean and vulnerable commits. A label value of 1 indicates an instance with a vulnerability. This is plotted along the bottom row and rightmost column of plots, and is represented in each scatterplot as a red triangle. The diagonal row of histograms show the distribution of values of a single attribute, the attribute represented by that row and column of plots. This plot was generated using the Jan 2013 test set for FFmpeg at commit level.

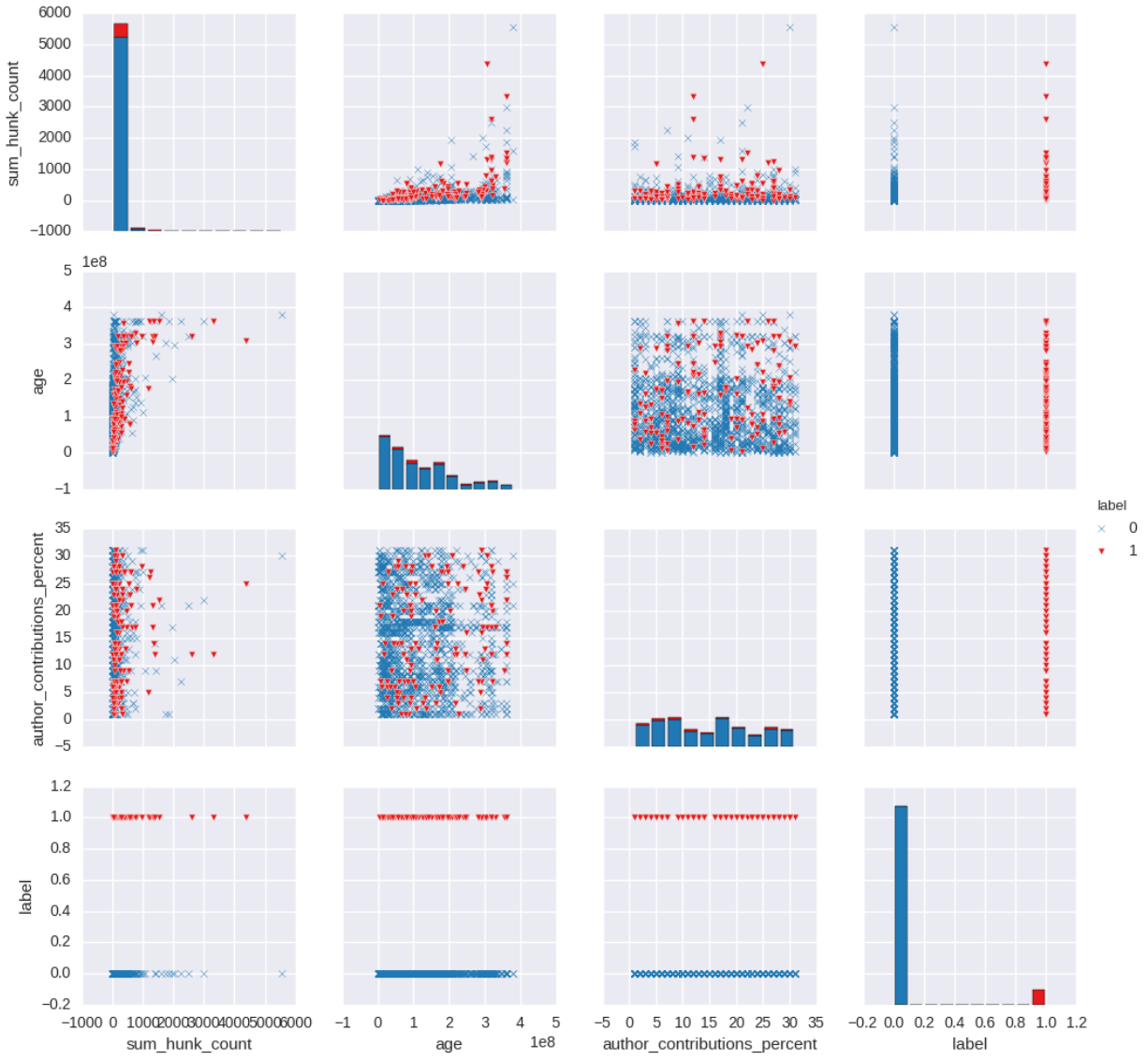

Figure 6.3: Paired plots showing the paired distribution of clean and vulnerable commits. A label value of 1 indicates an instance with a vulnerability. This is plotted along the bottom row and rightmost column of plots, and is represented in each scatterplot as a red triangle. The diagonal row of histograms show the distribution of values of a single attribute, the attribute represented by that row and column of plots. This plot was generated using the Jan 2013 test set for FFmpeg at file level.

Chapter 7

Threats to Validity

Following the work of Perry et al. [51], as referenced by Posnett et al. [54], we examine the construct, internal and external validity of our work.

Construct validity examines how accurate our methods measure the questions and hypotheses we set out to measure. One issue that could impact our construct validity is our choice of performance measures that represent only a single data point. Specifically, F_1 score and TPR10k measure single data points, i.e. F_1 measures performance using a single threshold, and TPR10k finds the percentage of vulnerabilities for a single number of LOC examined. However, we rely on multiple metrics, including AUROC and AUCEC to evaluate the performance of the classifier, which do not rely on single threshold values. Additionally, TPR10k allows us to compare the cost effectiveness of vulnerability prediction at different levels independent of the total LOC in the test set, which differs at commit and file level.

Internal validity ensures that changes to the dependent variables are indeed the cause of changes seen in independent variables. To this end, we go to lengths to ensure that our experimental design 'fairly' compares commit-level prediction with file-level prediction, as described in Section 3. It can be noted that due to limitations based on cost of implementation and collection, certain features used in previous work for commit- and file-level prediction were not used in our evaluation. However, in order to mitigate the effect of the use of certain features (or the omission of others) on prediction results, we collect a wide variety of types of features, and attempt to collect analogous features at both commit and file level. Additionally, perform attribute selection on our set of features, which selects a subset of features in order to optimize the information gain provided by that subset of features. Therefore the features we use for building and evaluating our classifiers is a

feature selected subset of our more complete list of features, tuned to provide optimized information gain for the given level of prediction. Additionally, inaccuracies in our labelling of vulnerability fixing commits (as discussed in Section 6.1), and identification of vulnerable commits (as discussed in Section 2.5), may impact the evaluated performance of our classifiers. However, we use a labelling and blaming technique comparable to previous works in vulnerability prediction [47, 31, 71].

External validity examines the extent to which the results are generalizable. We perform our evaluation on 7 different projects, 6 different classifiers, for 8 different training dates. Our projects come from a variety of applications, from a virtual machine hypervisor, to a network analysis tool, to a scripting language. However, all the projects in our study are open source projects and projects written primarily in C/C++. While our evaluation technique is theoretically applicable to closed source projects and projects written in other languages, provided there exists a reliable method for linking fixing commits to known vulnerabilities, the distribution in the number and types of vulnerabilities may change such that the generalizability of our study is impacted.

Chapter 8

Related Work

8.1 Defect and Vulnerability Prediction

In the last couple of decades, researchers have used machine learning (ML) techniques to predict defects in software [34, 25, 38, 42, 31, 73]. While defect prediction is now accepted and used in the industry, many challenges make vulnerability prediction less applicable to industry. [43]. ML techniques have shown promising results to automatically predict software vulnerability bugs [4, 63, 68, 59, 27, 60, 77], In this section, we discuss related work, highlighting vulnerability or defect prediction that was evaluated at binary level, file level, or commit and change level. We also discuss works that similarly make comparisons between different levels of granularity, though these are generally performed between levels that are hierarchically related, unlike file and commit levels.

8.2 Vulnerabilty Prediction in binaries

Various metrics, e.g., code churn and code complexity, have been used to train ML models [63, 11, 75] to predict vulnerabilities in source code and binaries. Nagappan et al. [45] study the validity of the relative code churn measures as early indicators of system defect density, and their results conclude that relative values of code churn are better predictors than using absolute values. A study by Zimmermann et al. [79] used various metrics like code churn, code complexities, dependencies, code coverage and organizational measures to predict software vulnerabilities in Windows Vista binaries.

8.3 File-Level Prediction

Many defect prediction models have been proposed at file level [70, 42, 16]. Those studies can generally achieve very good results on different benchmarks. Similarly, file-level vulnerability prediction has been thoroughly studied [4, 68, 59, 60, 77]. Shin et al. [63] compare code churn, complexity, and developer activity metrics for vulnerability prediction. They find that all three kinds of metrics have discriminative power for vulnerability prediction, and achieve a high recall, but low precision, using a classifier built using a combination of all metrics. Hovsepyan et al. [27] rely on the textual analysis of the source code and treat every word (also called a monogram in text processing) in that source file as a feature. Their results yield high accuracy (average of 0.87), precision (average of 0.85) and recall (average of 0.88). However, they label vulnerabilities using warnings detected by a tool, and as a result have a much higher vulnerability rate. It is not clear whether their results generalize to other types of vulnerabilities not detected by the tool. For these works, results are generally less accurate than for general defect prediction level. Despite the wide range of features available, several studies show that commit-related features (i.e. code churn, number of past authors) are useful indicators for finding defects or vulnerabilities at file level [63, 44, 28, 45].

8.4 Commit-Level Prediction

Empirical studies have shown that buggy commits share some common features [6, 8, 34]. For example, Bavota et al. demonstrate that commits involving hierarchy modification are more likely to be buggy than others [6]. Kim et al. also found that particular keywords and the number of changes in the commits are also significant predictors [34]. Despite commit-related features being widely used for both general defect and vulnerability prediction, commit or change-level prediction have not yet been extensively studied. For general defect prediction, existing studies [34, 62, 31] show that predicting bugs at the commit and change levels is possible. CommitGuru, a tool predicting risks associated to specific commits has also been introduced recently [58]. However, the performances of this tool have not yet been thoroughly analyzed. For vulnerability prediction, despite some empirical studies analysing the characteristics of particular vulnerabilities [22] or vulnerable changes [8, 37], the only work evaluating vulnerability prediction at the commit level is VCCFinder [50], who report precision and recall values comparable to that of some works in file-level vulnerability prediction [63]. Different from these studies, our work focuses

on identifying the challenges and parameters impacting vulnerability prediction at commit level. In addition, we compare commit and file-level vulnerability prediction.

8.5 Impact of Prediction Granularity on Results and Performance

Additional previous work evaluates the impact of granularity on both general defect and vulnerability prediction. Morrison et al. focus on file and binary levels of granularity for vulnerability prediction [43]. They obtained better results when predicting at *binary level* than for file level. Posnett et al. study the differences between file, package and module levels for general defect prediction. [54] They also found that *binary-level* prediction was better than predicting on de-aggregated elements, when measured using AUROC. However, they discovered that predicting at a *finer granularity level* is better when considering cost-effectiveness metrics. While doing a comparison between prediction at different levels of granularity, these two studies differ from our study in several ways. First, commit level is a more fine-grained level of granularity than file level. None of these studies considers the possibility of doing prediction at commit level, and we do not know whether the conclusions obtained in previous studies still hold for comparison between commit and file levels. Second, we focus our investigation not only on the results, but on the different features and labelling possible in commit and file level. Previous studies do not consider these parameters in their analysis.

Chapter 9

Conclusion

In this study, we have performed a comparison of commit- and file-level prediction across 7 projects, 6 classifiers, and 8 training dates, with the intent of informing the development of practical, real-world use of vulnerability prediction. We find that in most cases, vulnerability prediction at file level outperforms prediction at commit level. However, for some projects and classifiers, commit-level prediction may in fact be comparable with, or outperform, file-level prediction. We compare each of our projects using several different metrics, representing two different possible usage scenarios for vulnerability prediction, and we find that under both scenarios, *file level* outperforms commit-level prediction. This is despite the fact that commit-level prediction is a finer granularity level of prediction, and previous work [54, 43] finds that despite being more *accurate* at coarser levels of granularity, prediction is often more *cost effective* at finer levels of granularity.

We also provide an analysis of our automated technique for building datasets of commits and files containing known vulnerabilities, and extracting features, for use in online prediction. We discuss a threats to our internal validity that arise from limitations in our automated labelling process and from the availability of known vulnerabilities. These threats affect our technique, as they do the previous work on which our technique is based, and we provide insight in to the extent to which these threats may affect our evaluation. In addition, our choice of projects and dates is further limited by the availability of data in the National Vulnerability Database, which for several projects, have the majority of the vulnerabilities listed for the project in the database only in the past 3-4 years. We propose that further understanding of these threats, and evaluation that focuses on practical, real-world use will lead to further development of vulnerability prediction tools.

We also achieve vulnerability prediction performance scores comparable with past work— that is, low [43, 63, 50]. Our F1 scores at file level achieve a max score of 0.65, but an average of 0.24, while our F1 scores at commit level achieve a max score of 0.30, and an average of 0.10. We propose, like previous work [43], that security-specific metrics may help to further improve vulnerability prediction. In our analysis, we examine the types of vulnerabilities (as identified by NVD) that are found in our data, and propose that further insights into security-specific metrics may be found by focusing on vulnerabilities by their type.

9.1 Future Work

There are a number of directions for future work, based on insights and experiences encountered in the course of this work.

First, while we learn that file-level prediction performs better than commit-level prediction for both 'classifier performance' and 'cost effectiveness' metrics, we note that the performance of vulnerability prediction itself is quite poor. In order to improve vulnerability prediction, we propose the development of security-specific features, specific to vulnerabilities as a subset of general defects. For example, Shar et al. [61] uses features specific to particular types of vulnerabilities in order to achieve high classifier performance. However, additional security-specific features have yet to be developed that improve prediction of vulnerabilities generally.

Similarly, building an ensemble of classifiers that focus on specific vulnerability types may allow us to improve general prediction performance while still taking advantage of vulnerability type information provided by databases like NVD. However, the challenge here is finding enough instances of a given vulnerability type to build an effective classifier. Because there is already a very low positive rate for vulnerabilities generally, splitting our set of vulnerabilities into smaller groups based on type decreases the positive rate and negatively impacts the quality of the classification model.

Another possible direction for future work is to improve the linking of vulnerabilities to vulnerability fixing commits. As seen is Table 6.1, a number of vulnerabilities listed in NVD were not able to be matched to their fixing commits, impacting the internal validity of our study. Finding ways to further improve upon our technique for identifying vulnerability fixing commits would facilitate an improved evaluation of existing vulnerability prediction techniques, as well as allow higher quality vulnerability prediction models to be built.

Other directions for future work relate to possibilities for improving the cost effectiveness of commit-level prediction. Based on the results reported in [54], one might expect commit-level results, the lower granularity of the two prediction levels, to be more cost effective, however this is not the case. One option for improving cost effectiveness would be to ignore large commits during prediction, as done by Perl et al. [50], which means that unusually large commits are ignored, even though they may contain a vulnerability. In our study, based on manual inspection, we find that outliers in commit size do exist (commits generated by automated processes, e.g., the refactoring of a variable name), and impact the cost effectiveness of commit-level prediction, however we do not set a threshold, as we believe this would unfairly bias commit-level results without implementing a similar threshold for file-level prediction. Recall that in our evaluation, as in previous work [50], merge commits are ignored.

An alternative to ignoring large commits would be to look at change-level prediction, given that changes provide an even smaller granularity than commit level, and will be no larger than the largest file. We look at commit-level vulnerability prediction in order to compare with previous work [50].

Finally, while we perform our comparison on vulnerability prediction in order to discover which prediction level is more promising to pursue given poor state-of-the-art performance results, performing a comparison of commit and file-level prediction with general defect prediction is a possible direction for future work. Since our process and tool already identifies fixing commits based on bug ids, this comparison is easily performed on the same dataset using existing data and methods, with minor modification to labelling methods.

In conclusion, our project set out to compare file-level and commit-level vulnerability prediction with an evaluation that measured practical, real-world utility. While the absolute results of the prediction models we build are not promising for practical use, the insights gained by our evaluation and comparison may be useful in informing further work in vulnerability prediction. Several directions for future work exist based on the insights gained from this study, from the improvement of vulnerability prediction, to extending the evaluation to general defect prediction.

References

[1] Comparing and Merging Files. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html, 2013. [Online; accessed 5-July-2016].

[2] Git. https://git-scm.com, 2013. [Online; accessed 5-July-2016].

[3] SciTools, Inc., Understand for C/C++, www.scitools.com. www.scitools.com, 2016.

[4] Omar H Alhazmi, Yashwant K Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[5] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel JG Van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *The economics of information security and privacy*, pages 265–300. Springer, 2013.

[6] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE, 2012.

[7] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Does measuring code change improve fault prediction? In Tim Menzies, editor, *PROMISE*, page 2. ACM, 2011.

[8] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: an empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 257–268, 2014.

[9] Gargi Bougie, Christoph Treude, Daniel M. Germn, and Margaret-Anne D. Storey. A comparative exploration of freebsd bug lifetimes. In Jim Whitehead and Thomas Zimmermann, editors, *MSR*, pages 106–109. IEEE, 2010.

[10] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.

[11] 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.

[12] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J. Syst. Archit.*, 57(3):294–313, March 2011.

[13] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[14] H. Cooper, L.V. Hedges, and J.C. Valentine. *The Handbook of Research Synthesis and Meta-Analysis*, chapter 12, pages 227–228. UPCC book collections on Project MUSE. Russell Sage Foundation, 2009.

[15] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[16] Marco DAmbros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[17] P.D. Ellis. Thresholds for interpreting effect sizes. http://www.polyu.edu.hk/mm/effectsizefaqs/thresholds_for_interpreting_effect_sizes2.html, 2016. [Online; accessed 5-July-2016].

[18] Jon Eyolfson, Lin Tan, and Patrick Lam. Correlations between bugginess and time-based commit characteristics. *Springer Empirical Software Engineering*, 2013.

[19] Michael Gegick, Pete Rotella, and Laurie Williams. Predicting attack-prone components. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 181–190. IEEE, 2009.

[20] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

[21] Munawar Hafiz. Security oriented program transformations (or how to add security on demand). In *OOPSLA Companion*, pages 927–928, 2008.

[22] Munawar Hafiz. *Security on demand*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[23] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. Towards an organization of security patterns. *Volume*, 24:52–60, 2007.

[24] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[25] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Christian Holler. Predicting Security Vulnerabilities from Function Calls. In *Bachelor's Thesis*, 2007.

[27] 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*, MetriSec '12, pages 7–10, New York, NY, USA, 2012. ACM.

[28] Aram Hovsepyan, Riccardo Scandariato, Maximilian Steff, and Wouter Joosen. Design churn as predictor of vulnerabilities? *International Journal of Secure Software Engineering*, 2014.

[29] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[30] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 2013.

[32] Yasutaka Kamei, Hiroki Sato, Akito Monden, Shinji Kawaguchi, Hidetake Uwano, Masataka Nagura, Ken ichi Matsumoto, and Naoyasu Ubayashi. An empirical study of fault prediction with code clone metrics. In Koichi Matsuda, Ken ichi Matsumoto, and Akito Monden, editors, *IWSM/Mensura*, pages 55–61. IEEE, 2011.

[33] Chris F Kemerer and Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering*, 35(4):534–550, 2009.

[34] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.

[35] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[36] LLVM Linux project, http://llvm.linuxfoundation.org/index.php/Main_Page/.

[37] Andrew Meneely, Harshavardhan Srinivasan, Afiqah Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 65–74. IEEE, 2013.

[38] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[39] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan 2007.

[40] Tim Menzies, Justin S. Di Stefano, Mike Chapman, and Ken McGill. Metrics that matter. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 51–, Washington, DC, USA, 2002. IEEE Computer Society.

[41] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, December 2010.

[42] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[43] 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*. ACM, 2015.

[44] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE, 2008.

[45] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[46] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA, 2010. IEEE Computer Society.

[47] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, New York, NY, USA, 2007. ACM.

[48] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop*

on Security Measurements and Metrics, MetriSec '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM.

[49] National Vulnerability Database, http://nvd.nist.gov/.

[50] 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.

[51] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM, 2000.

[52] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[53] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. ACM.

[54] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, 2011.

[55] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[56] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press.

[57] Martin C Rinard. Living in the comfort zone. *ACM SIGPLAN Notices*, 42(10):611–622, 2007.

[58] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM, 2015.

[59] 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. ACM, 2012.

[60] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *Software Engineering, IEEE Transactions on*, 40(10):993–1006, 2014.

[61] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 642–651, Piscataway, NJ, USA, 2013. IEEE Press.

[62] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[63] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772–787, 2011.

[64] 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.

[65] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 315–317, New York, NY, USA, 2008. ACM.

[66] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7. ACM, 2011.

[67] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS '11, pages 1–7, New York, NY, USA, 2011. ACM.

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

[69] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 600–604, Washington, DC, USA, 2009. IEEE Computer Society.

[70] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on*, 39(4):552–569, 2013.

[71] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[72] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[73] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 2015 International Conference on Software Engineering (SEIP track)*, ICSE '15, 2015. in press.

[74] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.

[75] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Companion Proceedings of the 37th International Conference on Software Engineering*. IEEE.

[76] John Viega and Gary McGraw. *Building secure software: how to avoid security problems the right way*. Pearson Education, 2001.

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

[78] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European*

Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[79] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, April 2010.

[80] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

APPENDICES

Appendix A

Classifier Performances by Project, Level and Metric

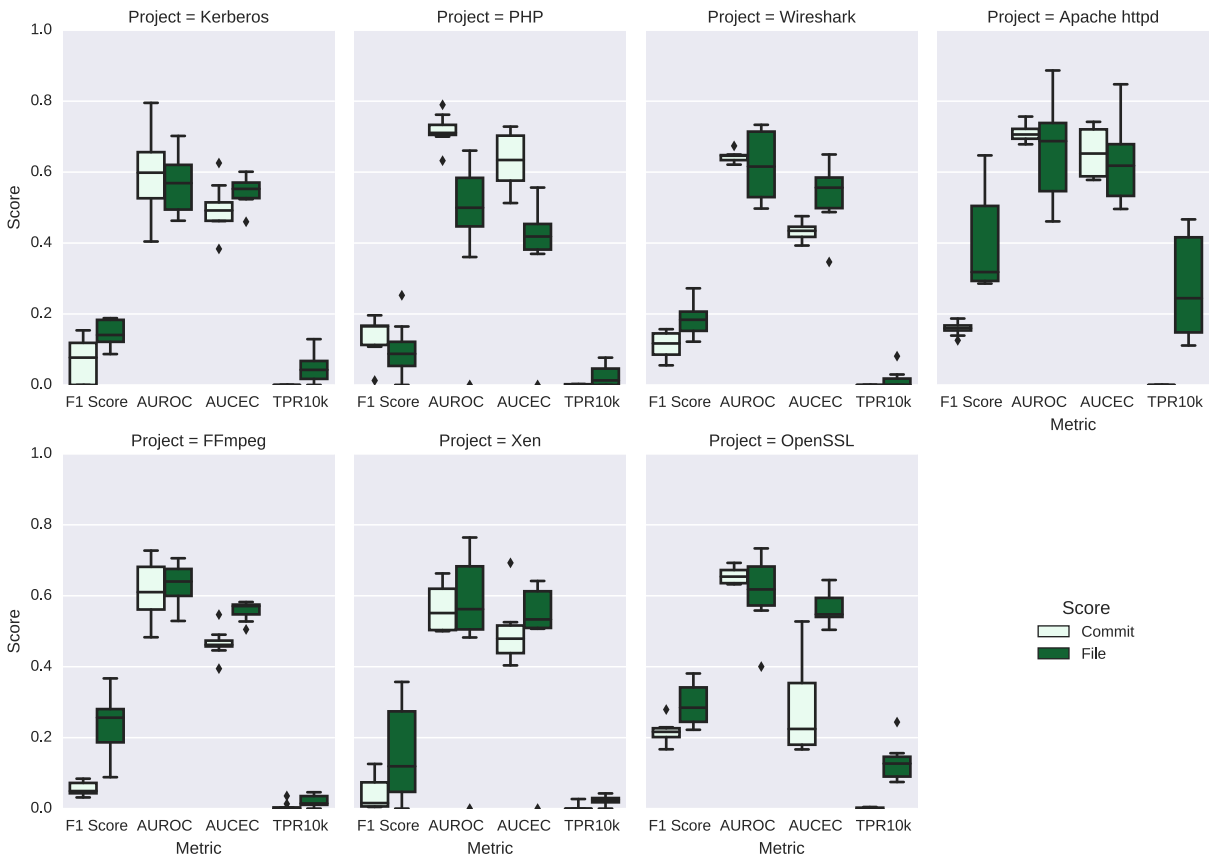

Figure A.1: J48 performances by project, level, and metric.

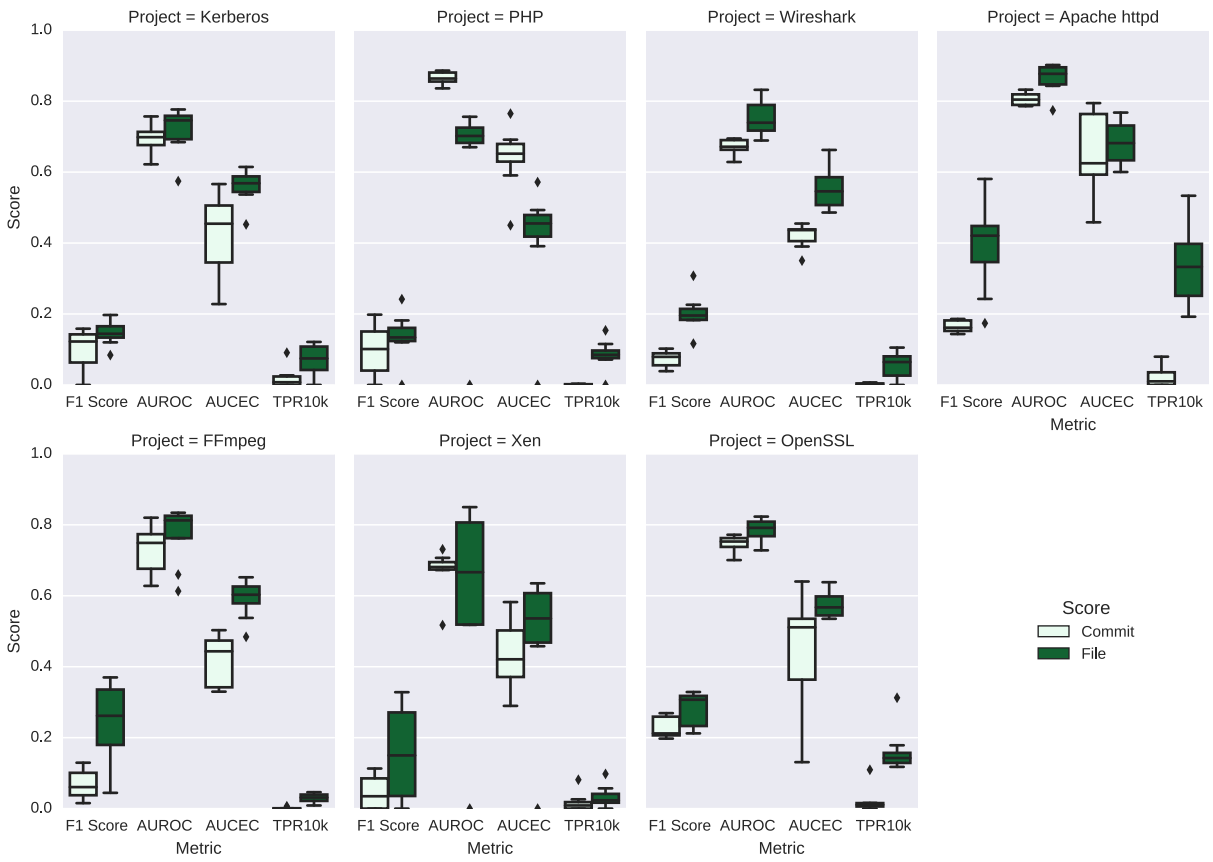

Figure A.2: ADTree performances by project, level, and metric.

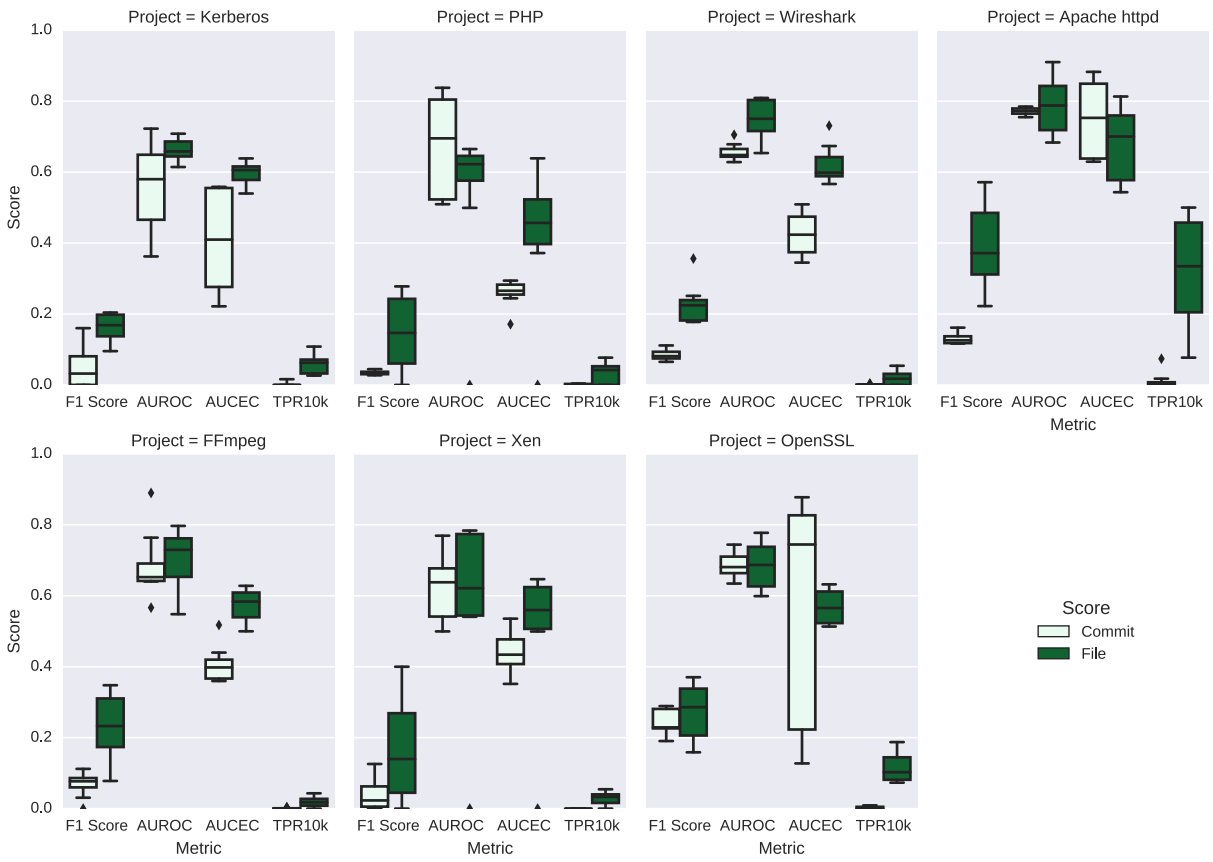

Figure A.3: Multilayer Perceptron performances by project, level, and metric.

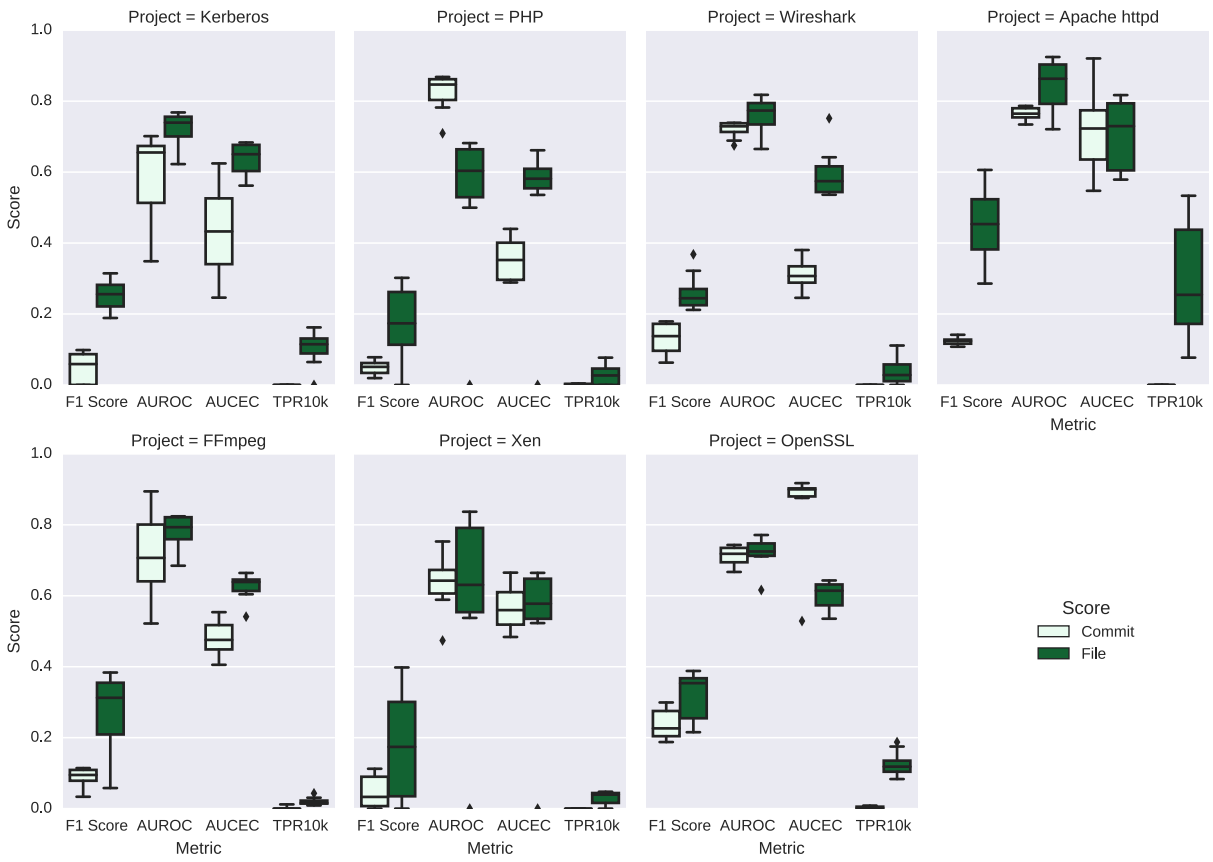

Figure A.4: Logistic regression performances by project, level, and metric.

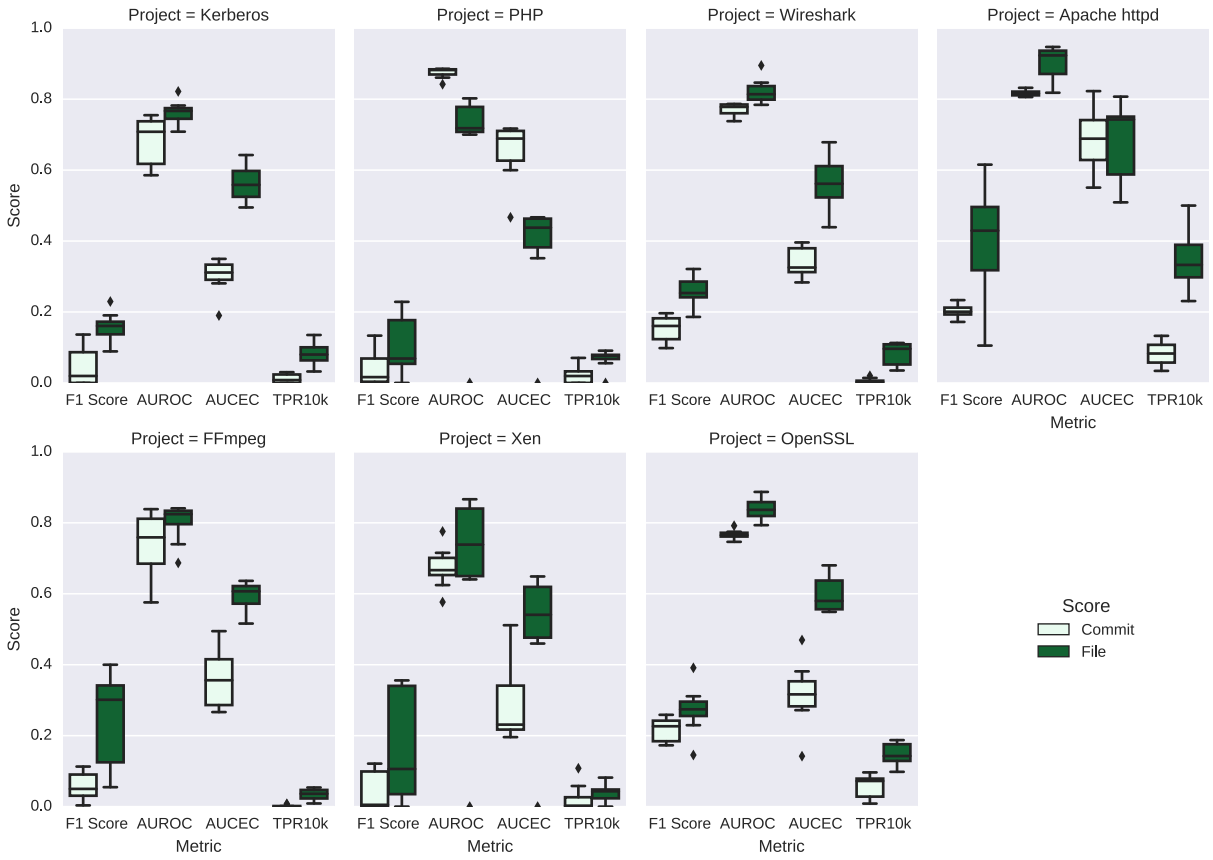

Figure A.5: Random Forest performances by project, level, and metric.

Appendix B

ROC and CE curve comparisons for Naive Bayes classifier, Jan 2013

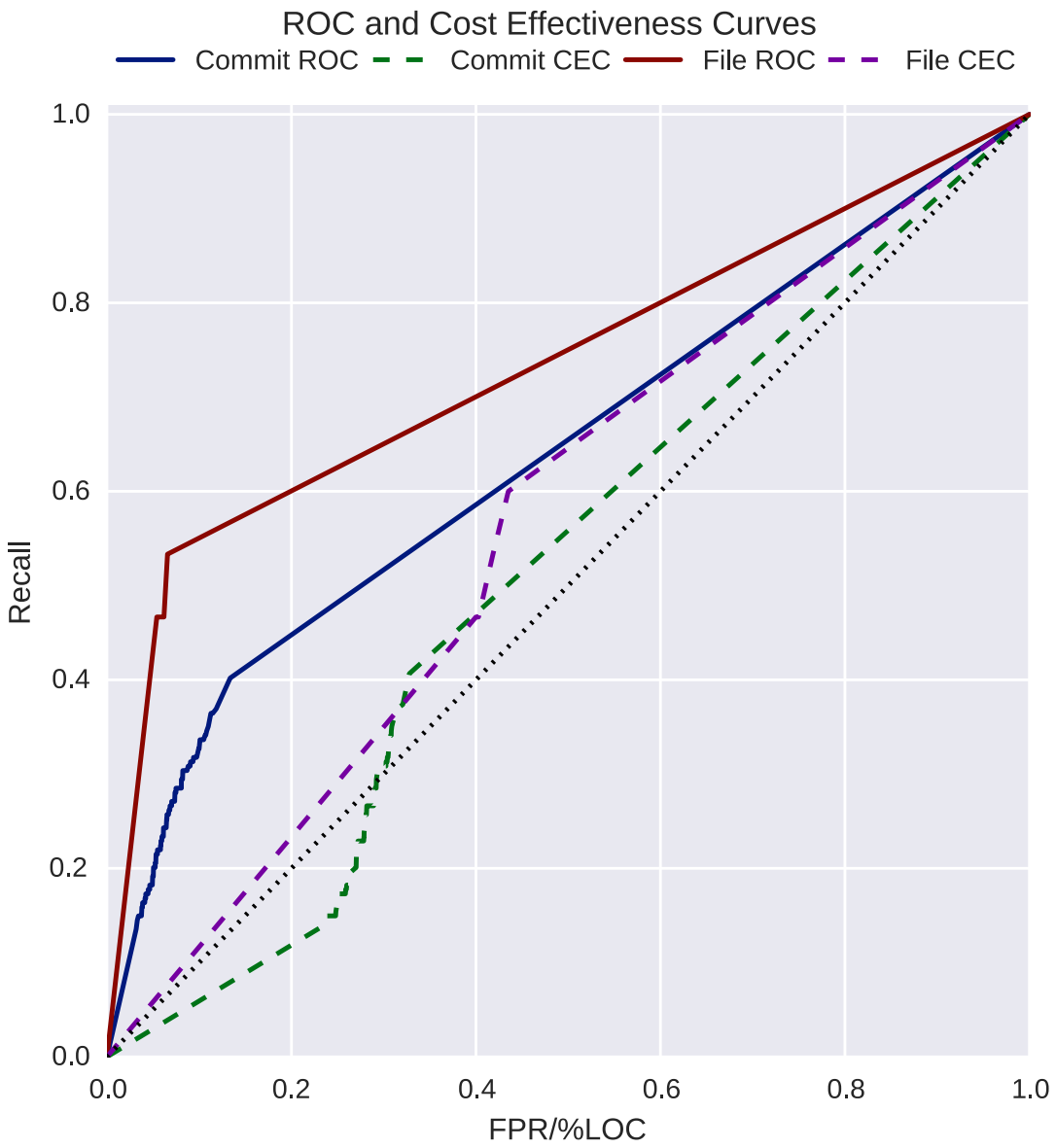

Figure B.1: ROC and CE curves, commit and file level, for httpd, Naive Bayes classifier, and Jan 2013 test date.

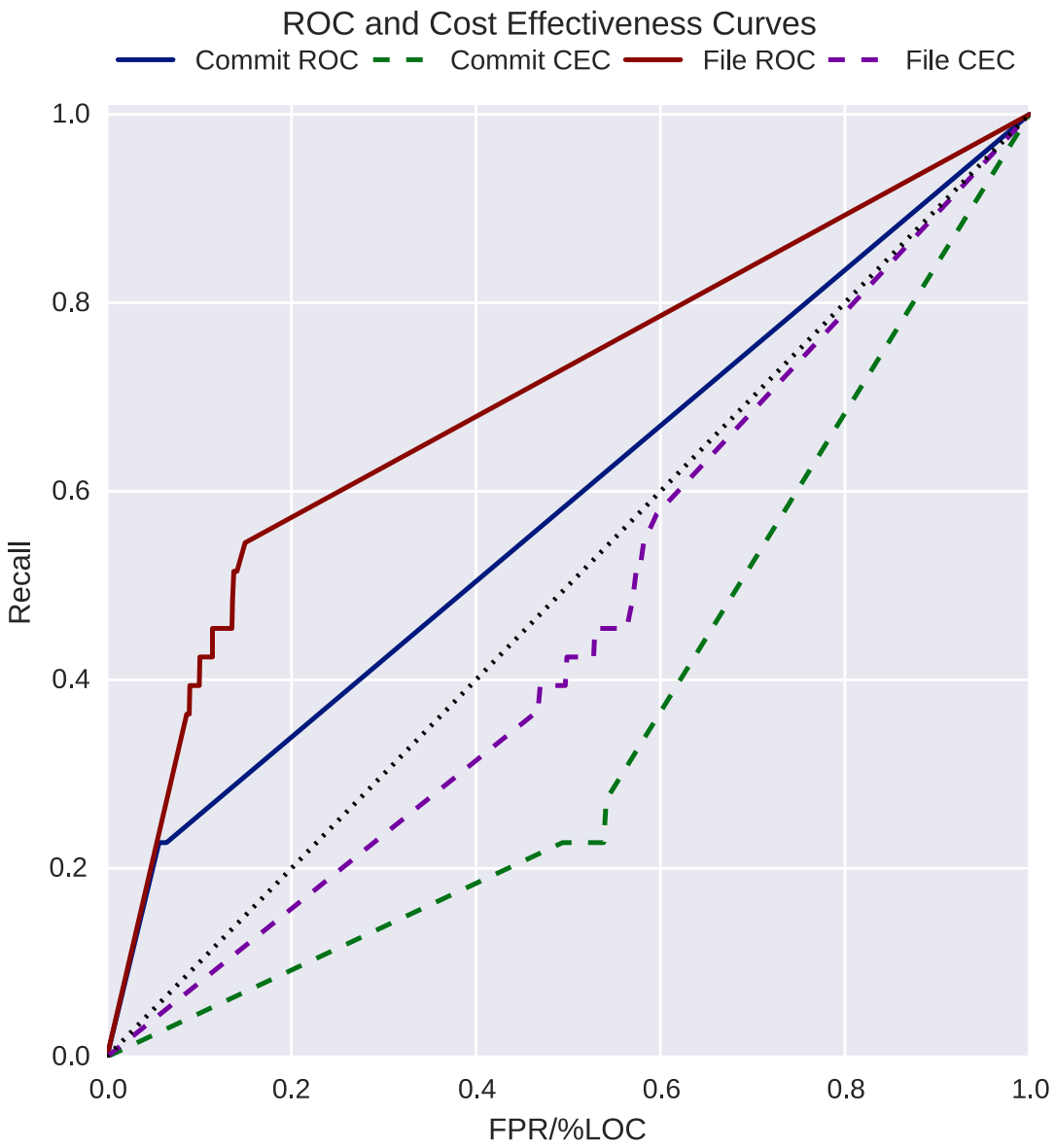

Figure B.2: ROC and CE curves, commit and file level, for Kerberos, Naive Bayes classifier, and Jan 2013 test date.

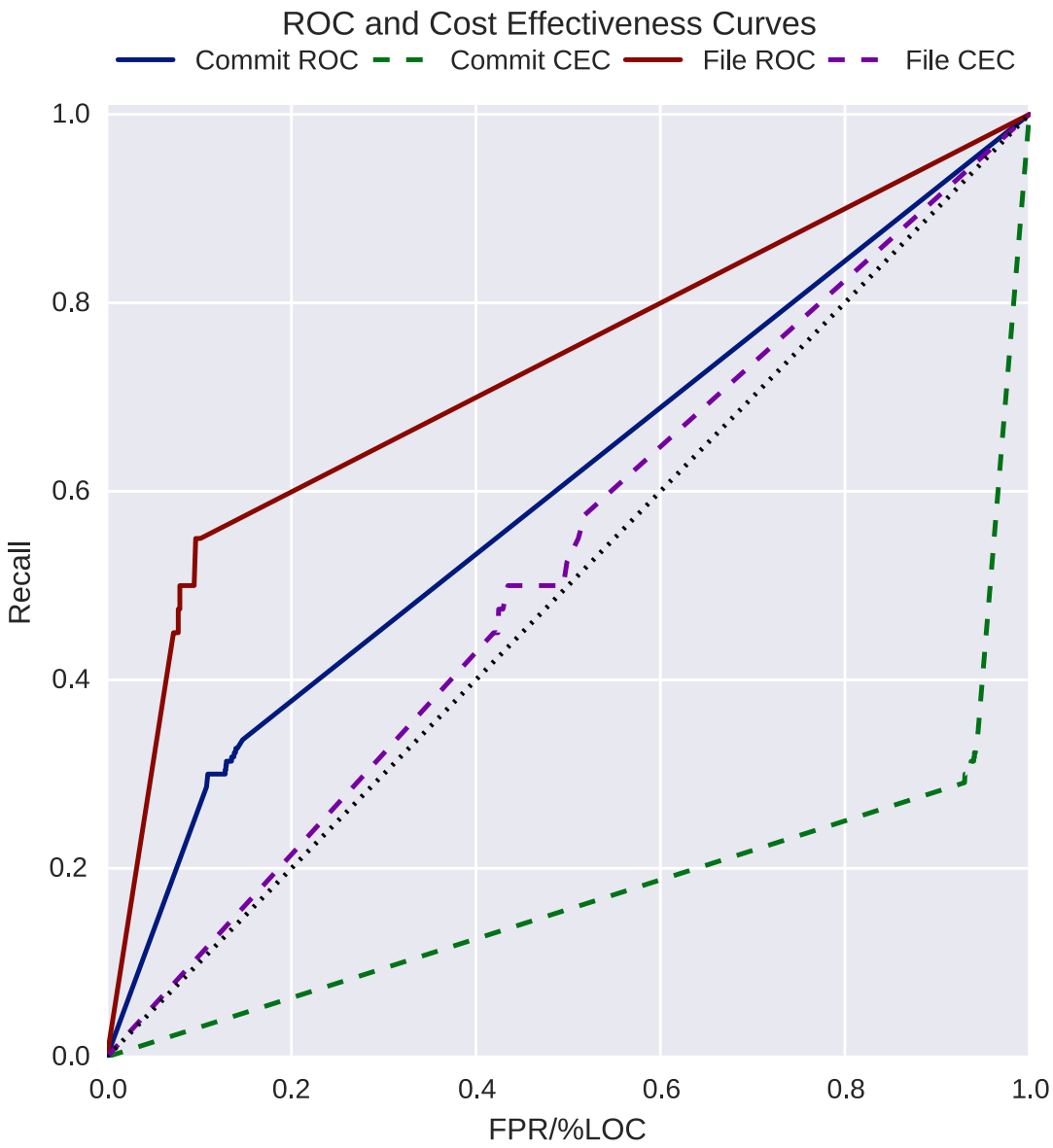

Figure B.3: ROC and CE curves, commit and file level, for OpenSSL, Naive Bayes classifier, and Jan 2013 test date.

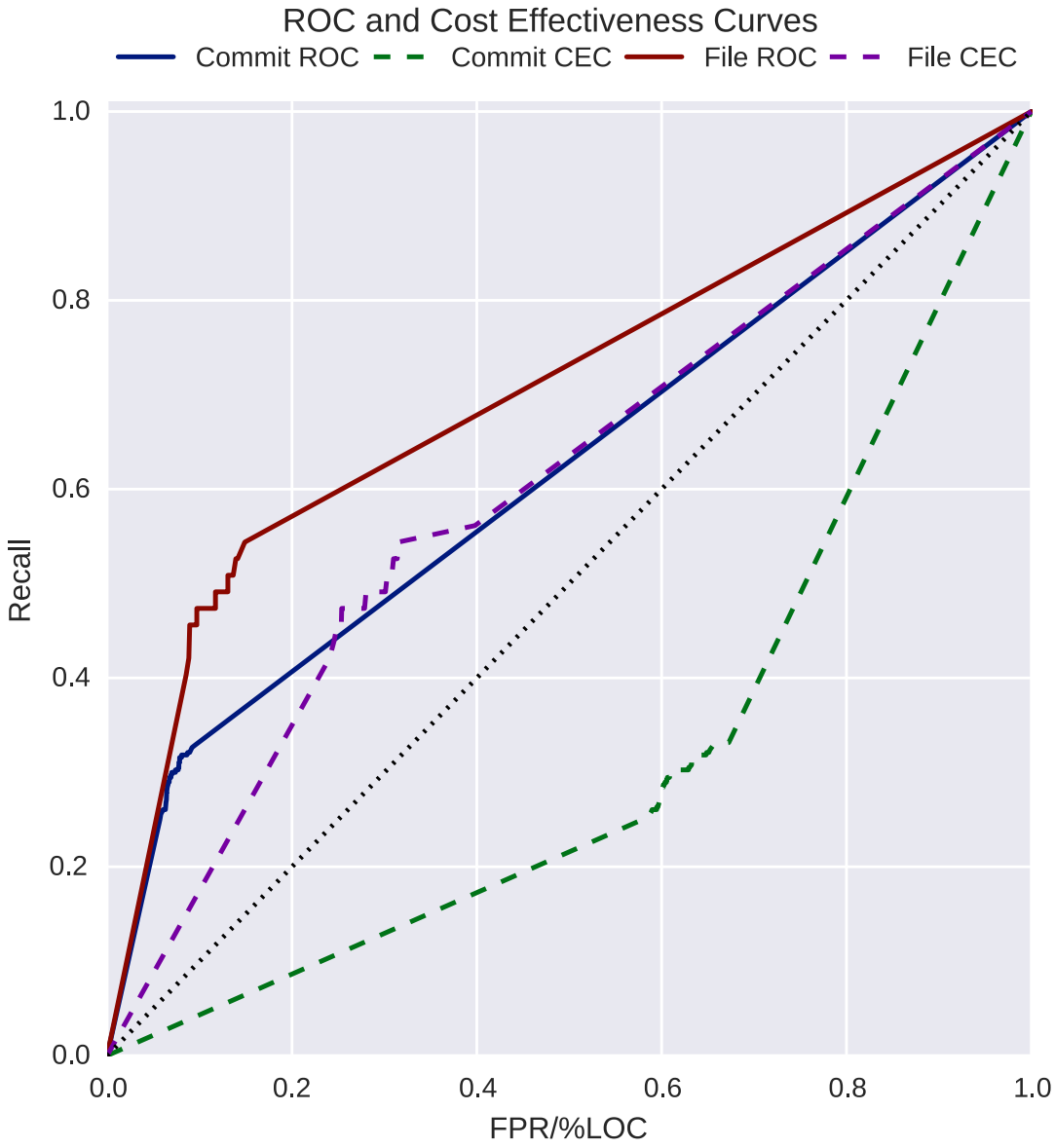

Figure B.4: ROC and CE curves, commit and file level, for Wireshark, Naive Bayes classifier, and Jan 2013 test date.

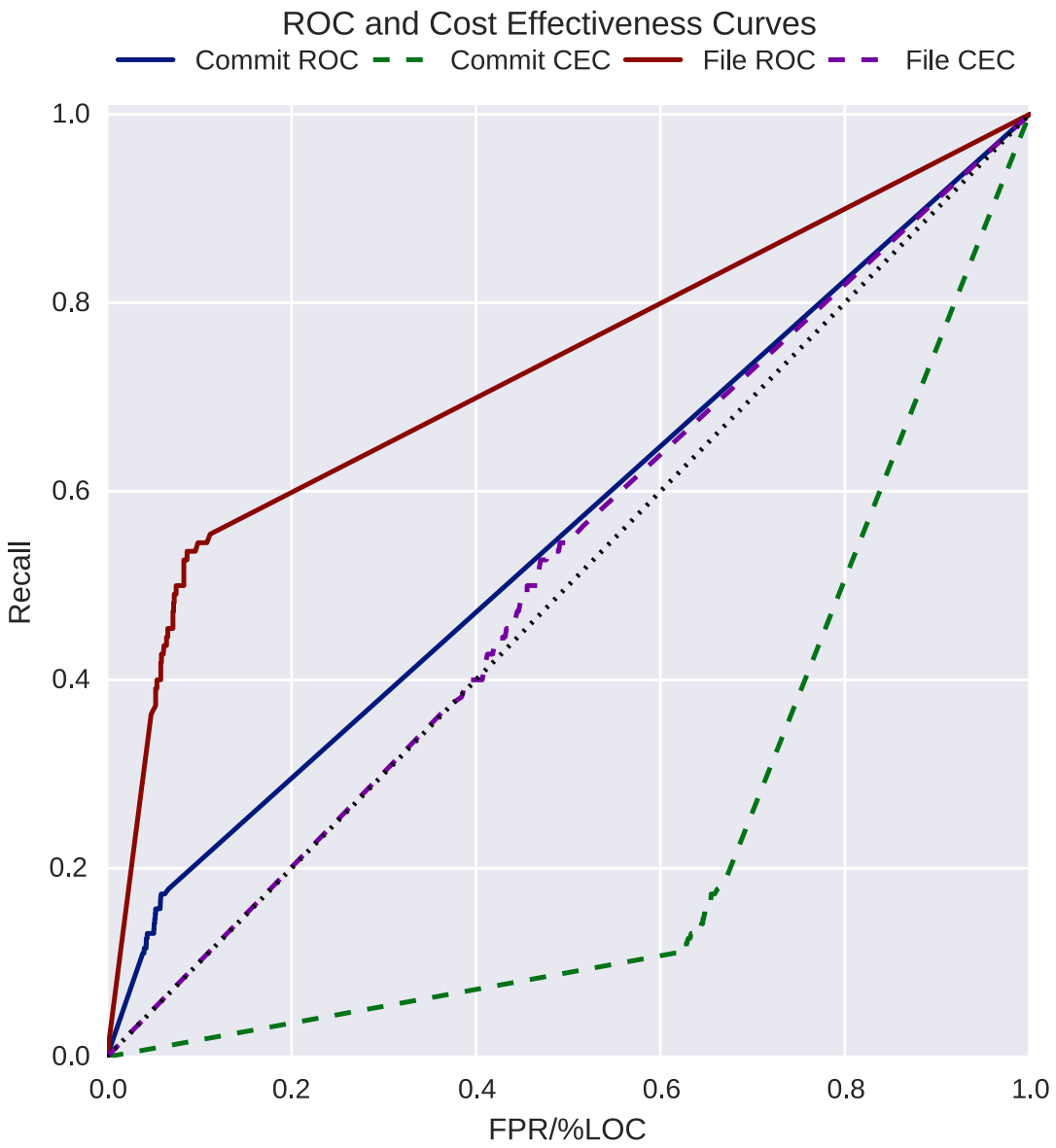

Figure B.5: ROC and CE curves, commit and file level, for Xen, Naive Bayes classifier, and Jan 2013 test date.

Appendix C

Vulnerability CWE types by project

Table C.1: Vulnerability CWE types by project

Project	CWE type	Vuln. count	Fix commit count
krb5	No Type Listed	35	91
krb5	CWE-16: Configuration	1	4
krb5	CWE-18: Source Code	2	18
krb5	CWE-20: Input Validation	29	78
krb5	CWE-94: Code Injection	2	5
krb5	CWE-119: Buffer Errors	28	63
krb5	CWE-189: Numeric Errors	17	41
krb5	CWE-200: Information Leak / Disclosure	3	10
krb5	CWE-255: Credentials Management	1	2
krb5	CWE-284: Improper Access Control	1	4
krb5	CWE-287: Authentication Issues	2	2
krb5	CWE-310: Cryptographic Issues	3	7
krb5	CWE-362: Race Conditions	1	1
krb5	CWE-399: Resource Management Errors	22	47
php-src	No Type Listed	36	115
php-src	CWE-17: Code	2	6
php-src	CWE-19: Data Handling	3	11
php-src	CWE-20: Input Validation	33	107
php-src	CWE-22: Path Traversal	3	3
php-src	CWE-59: Link Following	1	2
php-src	CWE-94: Code Injection	2	9
php-src	CWE-119: Buffer Errors	38	129
php-src	CWE-134: Format String Vulnerability	1	3
php-src	CWE-189: Numeric Errors	22	56
php-src	CWE-200: Information Leak / Disclosure	6	12
php-src	CWE-287: Authentication Issues	2	1
php-src	CWE-310: Cryptographic Issues	6	22
php-src	CWE-362: Race Conditions	2	6
php-src	CWE-399: Resource Management Errors	12	26

Table C.2: Vulnerability CWE types by project - continued

Project	CWE type	Vuln. count	Fix commit count
wireshark	No Type Listed	38	105
wireshark	CWE-17: Code	1	5
wireshark	CWE-19: Data Handling	3	11
wireshark	CWE-20: Input Validation	88	315
wireshark	CWE-94: Code Injection	3	11
wireshark	CWE-119: Buffer Errors	57	183
wireshark	CWE-134: Format String Vulnerability	1	3
wireshark	CWE-189: Numeric Errors	42	152
wireshark	CWE-200: Information Leak / Disclosure	6	4
wireshark	CWE-255: Credentials Management	1	1
wireshark	CWE-287: Authentication Issues	2	1
wireshark	CWE-310: Cryptographic Issues	1	2
wireshark	CWE-362: Race Conditions	1	1
wireshark	CWE-399: Resource Management Errors	39	136
httpd	No Type Listed	19	49
httpd	CWE-16: Configuration	1	7
httpd	CWE-17: Code	1	5
httpd	CWE-20: Input Validation	15	49
httpd	CWE-79: Cross-Site Scripting (XSS)	4	13
httpd	CWE-94: Code Injection	1	3
httpd	CWE-119: Buffer Errors	8	19
httpd	CWE-189: Numeric Errors	8	19
httpd	CWE-200: Information Leak / Disclosure	4	7
httpd	CWE-310: Cryptographic Issues	3	11
httpd	CWE-352: Cross-Site Request Forgery (CSRF)	1	3
httpd	CWE-362: Race Conditions	1	4
httpd	CWE-399: Resource Management Errors	17	43
FFmpeg	No Type Listed	47	152
FFmpeg	CWE-16: Configuration	1	2
FFmpeg	CWE-17: Code	4	4
FFmpeg	CWE-20: Input Validation	36	110
FFmpeg	CWE-94: Code Injection	2	8
FFmpeg	CWE-119: Buffer Errors	84	306

Table C.3: Vulnerability CWE types by project - continued (2)

Project	CWE type	Vuln. count	Fix commit count
FFmpeg	CWE-134: Format String Vulnerability	1	1
FFmpeg	CWE-189: Numeric Errors	32	66
FFmpeg	CWE-310: Cryptographic Issues	2	4
FFmpeg	CWE-362: Race Conditions	2	4
FFmpeg	CWE-399: Resource Management Errors	15	42
xen	No Type Listed	11	38
xen	CWE-16: Configuration	4	11
xen	CWE-17: Code	6	25
xen	CWE-19: Data Handling	2	10
xen	CWE-20: Input Validation	20	67
xen	CWE-59: Link Following	1	1
xen	CWE-119: Buffer Errors	11	66
xen	CWE-189: Numeric Errors	1	3
xen	CWE-200: Information Leak / Disclosure	8	34
xen	CWE-254: Security Features	2	9
xen	CWE-310: Cryptographic Issues	1	4
xen	CWE-362: Race Conditions	1	5
xen	CWE-399: Resource Management Errors	15	65
openssl	No Type Listed	20	88
openssl	CWE-17: Code	3	12
openssl	CWE-20: Input Validation	13	40
openssl	CWE-119: Buffer Errors	16	55
openssl	CWE-189: Numeric Errors	3	13
openssl	CWE-200: Information Leak / Disclosure	4	13
openssl	CWE-254: Security Features	1	10
openssl	CWE-287: Authentication Issues	1	2
openssl	CWE-310: Cryptographic Issues	18	90
openssl	CWE-362: Race Conditions	4	23
openssl	CWE-399: Resource Management Errors	20	67

Appendix D

Vulnerability time to fix histogram by project

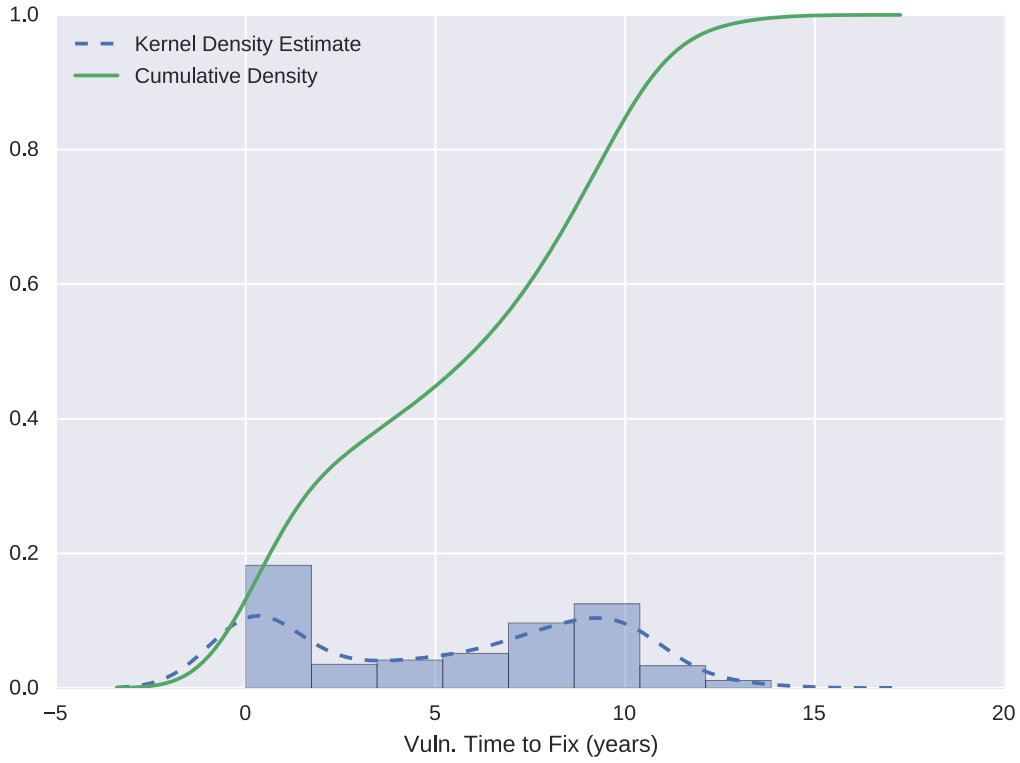

Figure D.1: Vulnerability times to fix in years for Apache httpd, showing both density and cumulative density estimation.

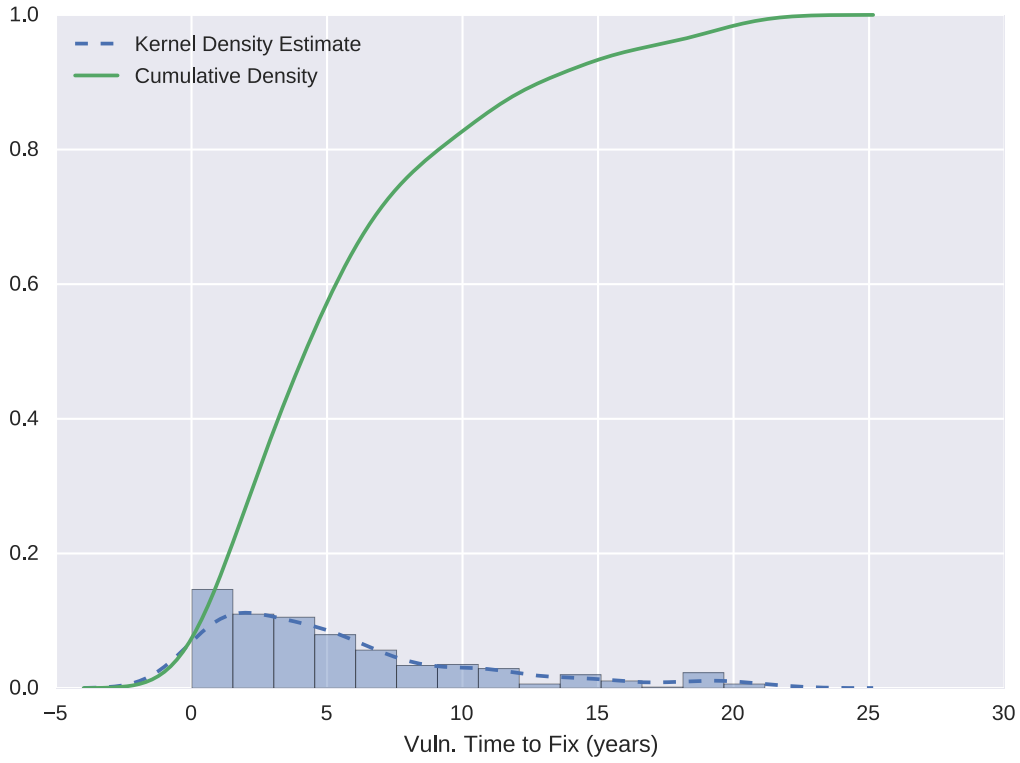

Figure D.2: Vulnerability times to fix in years for Kerberos, showing both density and cumulative density estimation.

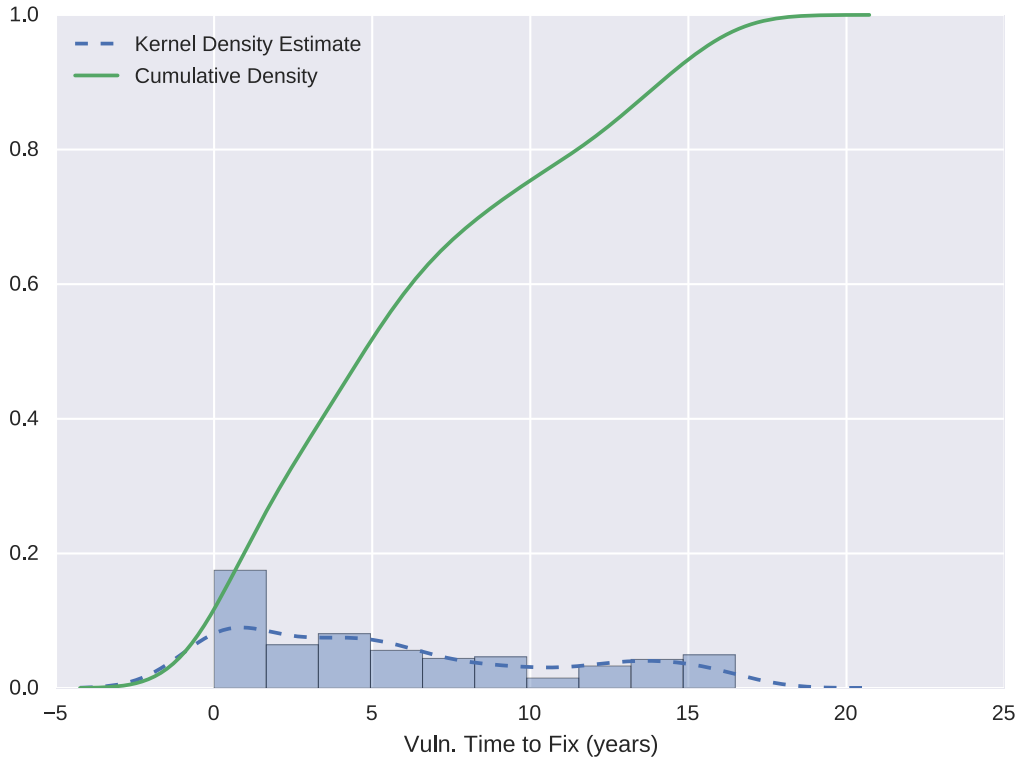

Figure D.3: Vulnerability times to fix in years for OpenSSL, showing both density and cumulative density estimation.

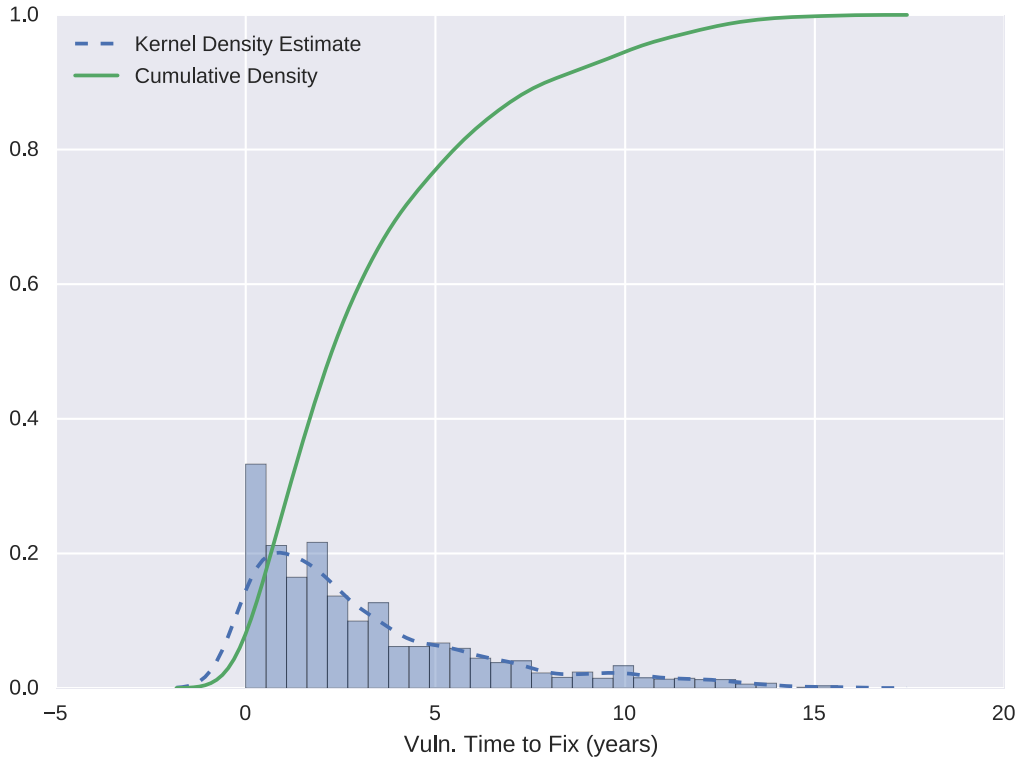

Figure D.4: Vulnerability times to fix in years for Wireshark, showing both density and cumulative density estimation.

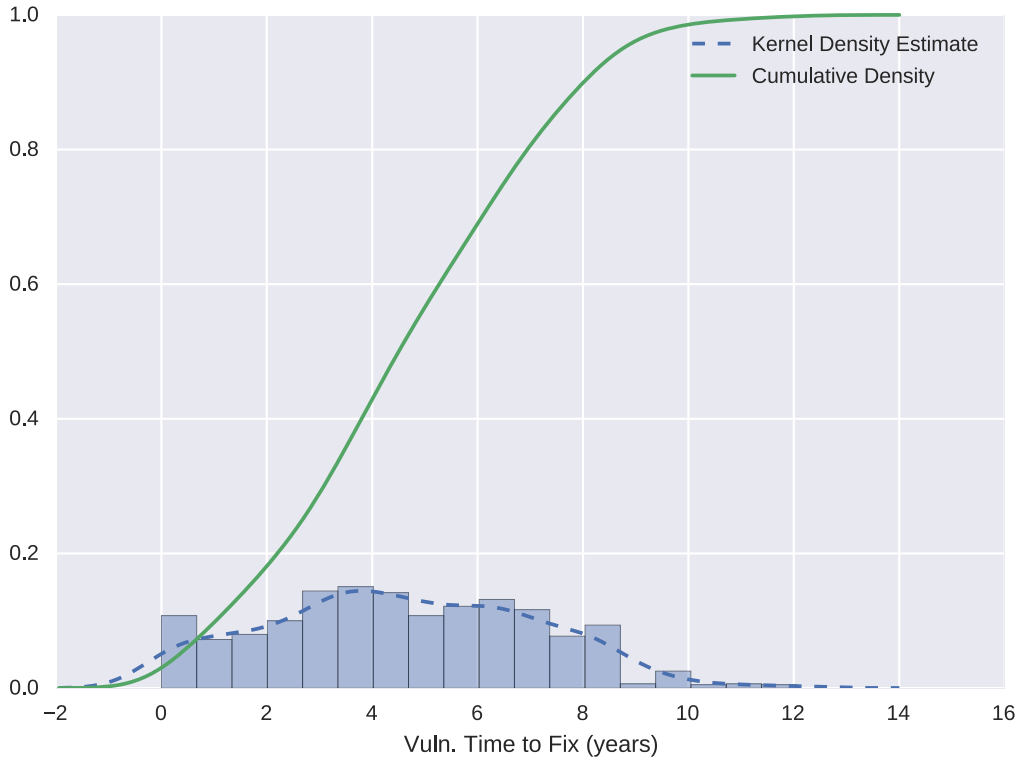

Figure D.5: Vulnerability times to fix in years for Xen, showing both density and cumulative density estimation.

Appendix E

Keywords used as features

Table E.1: Keyword Features

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

Table E.2: Keyword Features - continued

asm
dynamic_cast
namespace
reinterpret_cast
try
bool
explicit
new
static_cast
typeid
catch
false
operator
template
typename
class
friend
private
this
using
const_cast
inline
public
throw
virtual
delete
mutable
protected
true
wchar_t
malloc
calloc
realloc
free
alloca
alloc