Empir Software Eng
DOI 10.1007/s10664-011-9190-8

Can traditional fault prediction models be used for vulnerability prediction?

Yonghee Shin · Laurie Williams

Editor: Tim Menzies

Abstract Finding security vulnerabilities requires a different mindset than finding general faults in software—thinking like an attacker. Therefore, security engineers looking to prioritize security inspection and testing efforts may be better served by a prediction model that indicates security vulnerabilities rather than faults. At the same time, faults and vulnerabilities have commonalities that may allow development teams to use traditional fault prediction models and metrics for vulnerability prediction. The goal of our study is to determine whether fault prediction models can be used for vulnerability prediction or if specialized vulnerability prediction models should be developed when both models are built with traditional metrics of complexity, code churn, and fault history. We have performed an empirical study on a widely-used, large open source project, the Mozilla Firefox web browser, where 21% of the source code files have faults and only 3% of the files have vulnerabilities. Both the fault prediction model and the vulnerability prediction model provide similar ability in vulnerability prediction across a wide range of classification thresholds. For example, the fault prediction model provided recall of 83% and precision of 11% at classification threshold 0.6 and the vulnerability prediction model provided recall of 83% and precision of 12% at classification threshold 0.5. Our results suggest that fault prediction models based upon traditional metrics can substitute for specialized vulnerability prediction models. However, both fault prediction and vulnerability prediction models require significant improvement to reduce false positives while providing high recall.

Keywords Software metrics · Complexity metrics · Fault prediction · Vulnerability prediction · Open source project · Automated text classification

Y. Shin (✉)
DePaul University, Chicago, IL 60604, USA
e-mail: yshin@cdm.depaul.edu

L. Williams
North Carolina State University, Raleigh, NC 27695, USA
e-mail: williams@csc.ncsu.edu

Published online: 09 December 2011

1 Introduction

Fixing faults discovered after software release costs significantly more than fixing the faults during development. Additionally, inadequate software testing can waste a significant amount of money (NIST 2002). Hence, with limited time and budget in development teams, efficient allocation of inspection and testing resources are critical. The prediction of code locations that have a high probability of having faults to improve the efficiency of inspection and testing has been an active research area for decades [some example research papers: (Kim et al. 2007; Menzies et al. 2007b; Basili et al. 1996; Ostrand et al. 2005; Nagappan and Ball 2005; Zimmermann and Nagappan 2008; Khoshgoftaar et al. 1996; Hassan 2009)]. Research in the prediction of software vulnerabilities (security problems) using the information collectable during software development is emerging [example papers: (Meneely and Williams 2009; Neuhaus et al. 2007; Gegick et al. 2008; Zimmermann et al. 2010; Shin et al. 2011)]. Considering the consistent rate of vulnerability reports[1] and the immeasurable cost of insecure software (Rice 2007), the emerging research area of vulnerability prediction is important.

Vulnerabilities and faults are similar in that both vulnerabilities and faults can be caused by human mistakes in the development process. The mistakes are often related to complexity in code/design (McCabe 1976) and changes to the code (Graves et al. 2000). Hence, complexity metrics and code churn metrics have been used for fault prediction (Nagappan et al. 2006; Nagappan and Ball 2005; Graves et al. 2000). This similarity between vulnerabilities and faults allows the possibility of the use of the traditional fault prediction metrics (complexity and code churn) for vulnerability prediction. In our prior study (Shin et al. 2011), we provided the empirical evidence that the vulnerability prediction models using complexity and code churn metrics are helpful to predict vulnerable code locations with high recall but also generate many false positives.

Although vulnerabilities and faults have similar characteristics and some vulnerabilities can be found during the fault detection process, the differing intent of the user must be simulated in vulnerability and fault detection. Attackers actively seek vulnerabilities with malicious or criminal intent. As a result, vulnerability detection should be conducted by a software engineer trained in thinking like an attacker and in common vulnerability patterns. Conversely, normal users may encounter faults during their benevolent use of the software and fault detection can focus on exercising the specified functionality of a system. Therefore, the characteristics of code with discovered vulnerabilities may be different than the characteristics of code with discovered faults. Additionally, far fewer vulnerabilities than faults are typically reported in many projects (Alhazmi et al. 2007). Finding vulnerabilities is akin to finding a "needle in a haystack". Perhaps a vulnerability prediction model must be trained to find needles and not haystacks.

At the same time, development teams with fault prediction models may wonder if they need a separate vulnerability prediction model. If so, development teams should record and collect fault and vulnerability information in a way that the prediction is possible. Depending on the details of fault and vulnerability information available from the bug database, tracing code locations that have been fixed for faults or vulnerabilities is not always easy even when an organization maintains a bug database.

Therefore, with the aforementioned similarity and the differences, whether fault prediction models can be used to predict vulnerabilities with equal to or better prediction performance than vulnerability prediction models should be examined empirically. The goal

[1] http://www.cert.org

of our study is to determine whether fault prediction models can be used for vulnerability prediction or if specialized vulnerability prediction models should be developed when both models are built with traditional fault prediction metrics.

The traditional fault prediction metrics we analyze are complexity, code churn, and prior fault count metrics. We included the prior fault count metric because the metric has been effective in fault prediction (Ostrand et al. 2005; Arisholm and Briand 2006) and has not been investigated in the context of vulnerability prediction yet. To achieve our goal, we performed an empirical case study on a widely-used open source project, the Mozilla Firefox[2] web browser. We built both fault prediction models and vulnerability prediction models using the three types of traditional fault prediction metrics and measured how accurately the models predict vulnerable code locations. We also compared the prediction performance of the fault prediction models and the vulnerability prediction models to analyze the effect of the numbers of reported faults and vulnerabilities in fault and vulnerability prediction.

The contributions of this study are as follows:

- We provide empirical evidence that the code with vulnerabilities tends to be more complex, to have more frequent and larger changes, and to have more past faults than the code with faults.
- We provide empirical evidence that traditional fault prediction metrics can detect a high portion of vulnerable code locations, but should be significantly improved to reduce false positives while providing high recall.
- We provide empirical evidence that both fault and vulnerability prediction models provide similar prediction performance for vulnerability prediction and can be used interchangeably when the models are built with traditional fault prediction metrics.
- Our threshold sensitivity analysis shows that vulnerability prediction can exhibit a drastic decrease in recall with an increase of precision for certain range of classification thresholds. This result calls for special attention in determining classification thresholds for vulnerability prediction.
- We show that the performance of fault and vulnerability prediction is largely affected by the number of the reported faults and vulnerabilities in previous releases.

The rest of this paper is organized as follows: Section 2 provides background and related work. Section 3 describes study design including metrics, experimental design, modeling techniques, and evaluation methods. Section 4 provides the results of our experiments. Section 5 discusses the implication of the results. Section 6 mentions threats to validity. Section 7 summarizes our study.

2 Background and Related Work

This section provides the terms that we will use in this paper and discusses prior studies related to our study.

2.1 Terms

A software *fault* is "an accidental condition that causes a functional unit to fail to perform its required function (IEEE 1988)."

[2] http://www.mozilla.com/

A software *vulnerability* is "an instance of an error in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy" (Krsul 1998). A software vulnerability provides functionality beyond required functionality, such as enabling an elevation of privilege or providing more information about an internal implementation of a system than necessary in an error message. Attackers can exploit this additional functionality.

In our study, the unit of analysis is a file. When a file has at least one reported fault, we call the file *faulty*. When a file has at least one reported vulnerability, we call the file *vulnerable*. We call a file that is neither faulty nor vulnerable *neutral*.

A *fault prediction model* is a model that is trained to predict the probability that a file will have at least one fault. A binary value of whether a file has a fault or not is used as the value of the dependent variable to train a fault prediction model.

A *vulnerability prediction model* is a model that is trained to predict the probability that a file will have at least one vulnerability. A binary value of whether a file has a vulnerability or not is used as the value of the dependent variable to train a vulnerability prediction model.

In this paper, both the fault prediction model and the vulnerability prediction model use the traditional complexity, code churn, and fault history metrics as independent variables.

2.2 Binary Classification Evaluation Criteria

We perform vulnerability prediction by classifying a file as faulty (or vulnerable) according to the probability of a file having faults (or vulnerabilities). A file is classified as faulty (or vulnerable) if the predicted probability of having faults (or vulnerabilities) is greater than a certain threshold. Binary classification can have two kinds of errors: *False Positives (FP)* and *False Negatives (FN)*. An FP happens when a file is classified as faulty or vulnerable when it is not. FPs can lead to a waste of resources in code inspection and testing. An FN happens when a file is classified as non-faulty or non-vulnerable when it actually has one or more faults or vulnerabilities. A software system may be released with undetected faults and vulnerabilities if the files with FNs are not considered for inspection and testing. Correct classifications are called *True Positives (TP)* and *True Negatives (TN)*. Table 1 summarizes the four types of classification.

From the binary classification results, we measured the degree of correct classification using *recall*, *precision*, and *probability of false alarm*.

Recall is the percentage of actual positives that are correctly classified as positives as defined in Eq. 1:

$$recall = TP * 100/(TP + FN) \tag{1}$$

Precision is the percentage of predicted positives that are correctly classified as positives as defined in Eq. 2:

$$precision = TP * 100/(TP + FP) \tag{2}$$

Table 1 Binary classification results

	Predicted Yes	Predicted No
Actual Yes	TP	FN
Actual No	FP	TN

Probability of false alarm (PF) is the percentage of actual negatives that are incorrectly classified as positives as defined in Eq. 3:

$$PF = FP * 100/(FP + TN) \tag{3}$$

We further elaborate these evaluation criteria in Section 3.3 for the purpose of fault and vulnerability prediction.

Additionally, we measure *accuracy* to measure the performance of overall correct classification as defined in Eq. 4:

$$Accuracy = (TP + TN) *100/(TP + FP + TN + FN) \tag{4}$$

We use accuracy to measure the classification performance of bug reports as a part of feasibility study that will be detailed in Section 4. Because accuracy can provide misleadingly high values for the data sets with very small percentage of faults or vulnerabilities (Menzies et al. 2008) as in our case, we use accuracy only for the purpose of comparison with a prior study conducted for classification of bug reports.

Setting a high classification threshold results in higher precision and lower recall than setting a low classification threshold. An appropriate threshold should be determined depending on the project type and resource availability in a project team. For a risk-adverse project that requires the removal of as many vulnerabilities as possible, setting a low threshold is beneficial. For a cost-adverse project that desires to minimize the waste of inspection and testing efforts that can result from false positives, setting a high threshold is beneficial (Jiang et al. 2008b).

2.3 Related Work

Fault Prediction Various metrics have been used for fault prediction including product metrics, such as OO design and complexity metrics, and process metrics, such as code churn and past fault history metrics (Hassan 2009; Basili et al. 1996; Khoshgoftaar et al. 1996; Zimmermann and Nagappan 2008; Ostrand et al. 2005; Nagappan and Ball 2005; Nagappan et al. 2006; Menzies et al. 2007b; Graves et al. 2000; Kim et al. 2007). In this section, we summarize a few representative studies on fault and failure prediction that use complexity, code churn, and fault history metrics.

Nagappan et al. (Nagappan et al. 2006) performed a post-release failure prediction using complexity metrics on five Microsoft software systems. Although no single complexity metric was strongly correlated with post-release failures in all five projects, subsets of the complexity metrics were statistically significantly correlated with the post-release failures in all five projects.

Nagappan and Ball (Nagappan and Ball 2005) also performed a post-release failure prediction using relative code churn metrics on Windows Server 2003. Their multiple linear regression model using principal components provided a high correlation between the estimated failures and the actual failures in software modules (r=0.889 for the Pearson correlation and r=0.929 for the Spearman rank correlation). Their binary classification model correctly classified 89% of the modules.

Graves et al. observed that frequently changed modules and the modules with large changes tend to have more faults than other modules. They also observed that more recent changes induce more faults than older changes (Graves et al. 2000).

Ostrand et al. and Arisholm et al. used the number of faults in prior releases for fault prediction. In these studies, the coefficients of the fault history metrics in their prediction models were statistically significant (Ostrand et al. 2005; Arisholm and Briand 2006).

Vulnerability Prediction Neuhaus et al. performed a vulnerability prediction on the Mozilla open source project by analyzing the header file inclusion and function call relationships. Their prediction model provided 45% recall and 70% precision, and estimated 82% of the known vulnerabilities in the top 30% of components predicted as vulnerable (Neuhaus et al. 2007).

Gegick et al. predicted vulnerabilities using source lines of code, alert density from a static analysis tool, and code churn metrics. They performed a case study on a commercial telecommunications software system at component level. Their regression tree model predicted with 100% recall with 8% false positive rate at the best case (Gegick et al. 2008).

Gegick et al. also performed a study to investigate whether the failure information can be used as an indicator of vulnerabilities. Their case study on a Cisco software system found 57% of the vulnerable components in the top nine percent of the total predicted component ranking, but with a high percentage of false positives (48%) (Gegick et al. 2009).

Meneely and Williams investigated the relationships between the structure of developer collaboration and vulnerabilities. Their empirical study with the Red Hat Enterprise Linux kernel shows that files changed by nine or more developers are more likely to have a vulnerability than files changed by fewer developers (Meneely and Williams 2009).

Shin and Williams investigated the relationships between code complexity and vulnerabilities on the Mozilla JavaScript engine at function level. The correlations between code complexity and vulnerabilities were weak (Spearman $r=0.3$ at best) but statistically significant (Shin and Williams 2008). Shin et al. further investigated whether complexity, code churn, and developer activity metrics can be used as indicators of vulnerable code locations (Shin et al. 2011). In their case study using the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel, the vulnerability prediction model predicted over 80% of the known vulnerable files with less than 25% false positives for both projects.

Although some of these studies showed the usefulness of traditional metrics to predict vulnerable code locations, none of the above studies investigated whether fault prediction models can be used to predict vulnerabilities, or compared the effectiveness of fault prediction models and vulnerability prediction models in depth as in our study.

3 Study Design

The goal of our study is to determine whether fault prediction models can be used for vulnerability prediction or if specialized vulnerability prediction models should be developed when both are built with traditional metrics of complexity, code churn, and fault history. For this purpose, we perform three experiments. First, we build a fault prediction model and evaluate its ability to predict faults. Second, we evaluate the ability of the fault prediction model to predict vulnerable code locations. Third, we build a vulnerability prediction model and evaluate its ability to predict vulnerabilities. In Subsections 3.1 through 3.4, we will provide the metrics, experimental design, measurements of prediction performance, and the modeling technique.

3.1 Traditional Fault Prediction Metrics

We use three types of traditional fault prediction metrics in this study: 18 complexity metrics, five code churn metrics, and one past fault history metric. Table 2 provides the definitions of the metrics.

3.2 Experiment Design

3.2.1 Experiment 1: Fault Prediction

To provide a baseline of comparison with the performance of vulnerability prediction, we built a fault prediction model by training it with the fault status of a file as a dependent variable and the traditional fault prediction metrics as independent variables. Then, we measured the recall, precision, and PF of the predicted faulty files. In this study, we refer to fault prediction using a fault prediction model as *FF prediction*.

Although there have been many fault prediction studies, those studies have been performed in different contexts: the studies have been performed using different subject

Table 2 Definitions of metrics

Metric	Definition
Complexity metrics	
LOC	The number of lines of code in a file.
CountLineCodeDecl	The number of lines of variable declarations.
CountDeclFunction	The number of functions defined in a file.
EssentialComplexity (average, sum)	The number of branches after iteratively reducing all the programming primitives such as a for loop in a function's control f low graph into a node until the graph cannot be reduced any further. Completely well-structured code has essential complexity 1. We used the average and sum values for each file.
CyclomaticStrict (average, sum, max)	The number of conditional statements in a function. We used the average, sum, and maximum values for each file.
MaxNesting (average, sum, max)	The maximum nesting level of control constructs such as if or while statements in a function. We used the average, sum, and maximum values for each file.
CommentDensity	The ratio of lines of comments to lines of code.
FanIn (average, sum, max)	The number of inputs to a function such as parameters and global variables. We used the average, sum, and maximum values for each file.
FanOut (average, sum, max)	The number of outputs from a function such as assignments to the function parameters or global variables. We used the average, sum, and maximum values for each file.
Code churn metrics	
NumChanges	The number of check-ins for a file.
LinesChanged	The number of code lines changed.
LinesInserted	The number of code lines inserted.
LinesDeleted	The number of code lines deleted.
LinesNew	The number of new code lines.
Past fault history metric	
NumPriorFaults	The number of faults in the prior release.

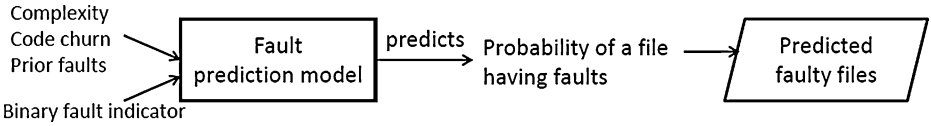

Fig. 1 Fault prediction using a fault prediction model (FF prediction)

projects, have used different metrics or modeling techniques, or have been performed on different sizes of entities such as component or file. Therefore, interpretation and comparison of the prediction results from different studies require caution, which is why we need a baseline for our own study.

Figure 1 shows the relationship between metrics, the model, and the purpose of prediction in FF prediction. In Figs. 1, 2, and 3, a rectangular represents a model and a parallelogram represents a list of files classified as faulty or vulnerable.

3.2.2 Experiment 2: Vulnerability Prediction using a Fault Prediction Model

If fault prediction models can be used for vulnerability prediction, organizations do not need to spend extra time and resources to create separate models for vulnerability prediction. Therefore, we built the same fault prediction model as we built for Experiment 1 and counted the number of correctly predicted vulnerable files contained in the list of predicted faulty files. Then, the result of the prediction is measured in terms of the recall, precision, and PF of those predicted vulnerable files. In this paper, we refer to vulnerability prediction using a fault prediction model as *VF prediction*.

Figure 2 shows the relationship between metrics, the model, and the purpose of prediction in VF prediction.

3.2.3 Experiment 3: Vulnerability Prediction using a Vulnerability Prediction Model

We built a vulnerability prediction model by training it with the vulnerability status of a file as a dependent variable and the traditional fault prediction metrics as independent variables. Then, we measured the recall, precision, and PF of the predicted vulnerable files. In this paper, we refer to the vulnerability prediction using a vulnerability prediction model as *VV prediction*.

Figure 3 shows the relationship between metrics, the model, and the purpose of prediction in VV prediction.

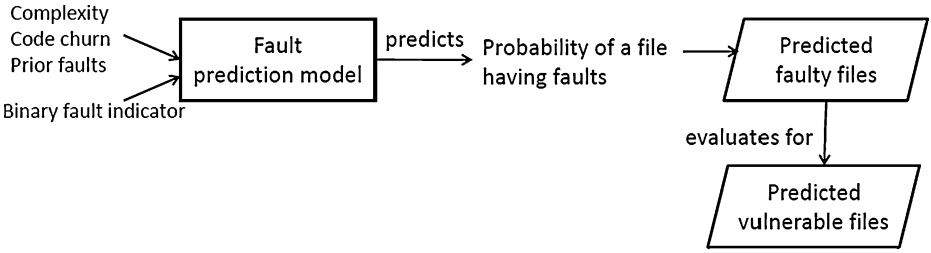

Fig. 2 Vulnerability prediction using a fault prediction model (VF prediction)

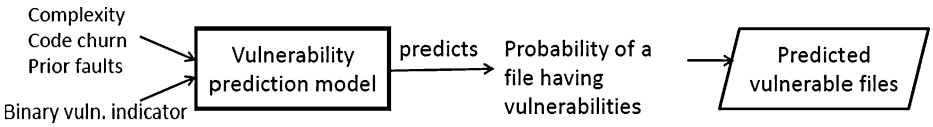

Fig. 3 Vulnerability prediction using a vulnerability prediction model (VV prediction)

3.3 Evaluation Criteria for Prediction Performance

To clarify recall and precision from the three types of predictions (FF, VF, and VV), we customize the definitions of recall,precision, and PF that we defined in Section 2.2 as follows:

- $Recall_{FF}$ is the percentage of actual faulty files that are correctly classified as faulty using a fault prediction model.
- $Recall_{VF}$ and $recall_{VV}$ are the percentages of actual vulnerable files that are correctly classified as vulnerable using a fault prediction model and a vulnerability prediction model, respectively.
- $Precision_{FF}$ is the percentage of predicted faulty files that are correctly classified as faulty using a fault prediction model.
- $Precision_{VF}$ and $precision_{VV}$ are the percentages of predicted vulnerable files that are correctly classified as vulnerable using a fault prediction model and a vulnerability prediction model, respectively.
- PF_{FF} is the percentage of actual non-faulty files that are incorrectly classified as faulty using a fault prediction model.
- PF_{VF} and PF_{VV} are the percentages of actual non-vulnerable files that are incorrectly classified as vulnerable using a fault prediction model and a vulnerability prediction model, respectively

If organizations have to inspect too many files only to find a small percentage of faults and vulnerabilities based upon prediction results, the prediction model is not efficient. Therefore, we additionally measured the number of files and lines of code to be inspected as a result of fault and vulnerability prediction. We define *File Inspection ratio (FI)* as the percentage of files to be inspected as a result of prediction in Eq. 5:

$$FI = (TP + FP) * 100/(TP + FP + TN + FN) \tag{5}$$

FI_F and FI_V are the percentages of files to be inspected as a result of fault prediction and vulnerability prediction, respectively. For example, FI_V=0.2 and $recall_{VV}$=0.8 indicate that 80% of the vulnerable files can be found in the 20% of the files predicted to be vulnerable. If we randomly choose files to be inspected, a security engineer may have to inspect 80% of the total files to detect 80% of the vulnerable files. Therefore, FI_V=0.2 and $recall_V$=0.8 indicate that the model is effective in reducing file inspection and testing efforts. Note that FF prediction and VF prediction provide the same FI_F since both of them use the same fault prediction model.

Although a file can be a logical unit for code inspection, a large file may require more effort to inspect than a small file. Therefore, we additionally measure *Line Inspection ratio (LI)*. LI is the percentage of lines of source code to be inspected as a result of prediction. First, we define the lines of code in the files that were true positives as TP_{LOC}, similarly with TN_{LOC}, FP_{LOC}, and FN_{LOC}. Then, LI is defined in Eq. 6:

$$LI = (TP_{LOC} + FP_{LOC}) * 100/(TP_{LOC} + FP_{LOC} + TN_{LOC} + FN_{LOC}) \tag{6}$$

LI_F and LI_V are the percentages of lines of code to be inspected as a result of fault prediction and vulnerability prediction, respectively.

Note that a small number of large files may contain many faults and vulnerabilities. Then, LI can be large when FI is small. In this case, FI may provide an overly optimistic result, while LI may provide an overly pessimistic result. If we predict faults and vulnerabilities at a finer granularity, such as a method or procedure level rather than a file level, the model may result in smaller LI. In fact, whether the file-based inspection effort estimation is better than the LOC-based inspection effort estimation is an open issue in the literature. Some researchers (Mende and Koschke 2009; Arisholm et al. 2007; Menzies et al. 2010) argue that a line of code is a better unit for effort estimation than a module, because the costs of testing and inspection are not equally distributed across modules and actually depend on the size of a module. On the other hand, Ostrand et al. (Ostrand et al. 2007) argues that while LOC-based effort estimation is appropriate for unit-testing, a coarse level of effort estimation such as the interface or functionality level is more appropriate for integration and system testing. In fact, either file-based inspection effort estimation or LOC-based inspection effort estimation may not precisely represent the effort for code analysis and debugging depending on the complexity of the problem. Therefore, we present both FI and LI in this study as a basic measurable approximation of effort, acknowledging that further research on the appropriate inspection effort estimation is required.

Note that one could use a trivial predictor that simply classifies all files as vulnerable, resulting in perfect recall and low precision. However, we would like to raise precision than that from the trivial predictor with sufficiently high recall using the information that can be obtained from software metrics. Therefore, we need to set an appropriate threshold to classify files based on the predicted probabilities. However, determining a precise threshold requires a cost analysis for misclassification, which is project-specific, as we discussed Section 2.2. Therefore, we perform classification at 91 classification thresholds between 0.01 and 0.91 at an interval of 0.01 instead of choosing a single classification threshold. After threshold 0.91, precision was not computable because no file was classified as faulty or vulnerable. If a project is risk-adverse and has enough resources for inspection and testing, a project team can select a model that performs well at a low threshold. If a project is cost-adverse, a project team can select a model that performs well at a high threshold. Our study compares the prediction performance in terms of basic evaluation metrics such as recall, precision, and PF separately because each evaluation metric provides an intuitive absolute measure of benefit in using a model and help development teams make an informed decision in selecting a prediction model. However, other approaches that compare models based on a combination of these basic metrics are also possible for other purposes (Jiang et al. 2008b; Menzies et al. 2007b; Witten and Frank 2005).

3.4 Prediction Models

We used a logistic regression technique to predict faulty files and vulnerable files. Logistic regression techniques have been frequently used and effective in fault prediction (Basili et al. 1996; Nagappan et al. 2006; Zimmermann and Nagappan 2008; Shin et al. 2011).

Logistic regression techniques calculate the probability of an event occurring by relating the linear combination of independent variables to the log of odds of an event (logit) (Ott and Longnecker 2001). This technique results in an outcome with a value in

the range of 0 to 1. Equation 7 defines the logit and the probability of an event resolved from the logit:

$$logit = \ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n = z \tag{7}$$

$$p = \frac{1}{1+e^{-z}}$$

where p is the probability that an event will occur, z is the linear combination of independent variables, and β_i is the change in logit per unit change of values of independent variables.

We additionally tried other classification techniques including Bayesian network, J48, and RandomForest. However, the results from those techniques were similar to the results from the logistic regression. Lessmann et al. (Lessmann et al. 2008) also observed no significant difference in 17 classification techniques that they tested for fault prediction. Therefore, we provide only the results from the logistic regression.

Prior studies have shown that the prediction performance of using only a small set of variables is as good as using many variables (Menzies et al. 2007b). By finding the small set of the most effective variables, we can reduce the time for data collection, model training, and prediction. We used the InfoGain variable selection method to selects variables. The InfoGain method selects variables that provide the highest information gain (Witten and Frank 2005). Using the InfoGain method, our models provided the best prediction performance when seven variables were selected. Except for the variable selection option, we used the default options for the logistic regression implemented in Weka 3.7 tool.[3]

After we trained a model on a training data set, we measured the prediction performance of the model on a test data set. For this purpose, we used 10×10 cross-validation (Witten and Frank 2005). In 10×10 cross-validation, a data set is divided randomly into ten folds. Then, nine folds of the data are used as a training data set and one fold of the data is used as a test data set. Each fold is used once as a test set. Because the prediction results can have a large variance depending on the distribution of faulty or vulnerable files as a result of random split of folds, it is common to repeat ten-fold cross-validation multiple times (Mark A. Hall and Holmes 2003; Witten and Frank 2005). We repeated ten-fold cross-validation ten times for each prediction.

Our data set is highly unbalanced in that the number of files with faults or vulnerabilities is much smaller than the number of files without faults or vulnerabilities in our study. This smaller set of data is called a minor class and the larger set of data is called a major class. Since the classification techniques try to reduce the error of the major class (Witten and Frank 2005), the minor class often has high false negatives. In fact, our initial vulnerability prediction model predicted all files as non-vulnerable in 64 out of 100 predictions, preventing any meaningful analysis. To deal with this issue, we adjusted the data set to make it balanced by applying data sampling. Data sampling can be performed by increasing the data instances in the minor class by duplication of data or by removing data instances from the major class. (Kamei et al. 2007; Menzies et al. 2008). We used an under-sampling technique to train a model by randomly removing the data instances in the major class from the training set until the number of data instances in the major class and the number of data instances in the minor class became equal. We performed the under-sampling for each run

[3] http://www.cs.waikato.ac.nz/ml/weka/

of 10×10 cross-validation. We under-sampled only training data, not test data. If we under-sample test data, the test data do not reflect the real distribution of vulnerabilities anymore. If we do not under-sample test data, the model is trained for a different distribution than the test data. However, the purpose of under-sampling is to train a model with a somewhat different distribution than the test set so that a classification model is not overly biased to the major class. Although under-sampling as well as any sampling is not a perfect approach to represent real data and should be used carefully, they are known to be helpful in practice (Menzies et al. 2008).

Although logistic regression does not assume a normal distribution of data and therefore does not require data transformation, recall was improved by log transformation in our initial investigation as was similarly observed by Jiang et al. (Jiang et al. 2008a). We provide the results of log transformation in this study.

In each iteration of 100 cross-validations, the data sampling and variable selection were performed following the algorithm described by (Shin et al. 2011). Figure 4 provides the algorithm generalized for r x k cross validation.

4 Case Study: Mozilla Firefox Browser

Mozilla Firefox is a widely-used open source web browser written in C/C++ and consists of over 10,000 files and over two million lines of source code. We used Firefox 2.0 in this study.

4.1 Data Collection

This subsection describes the method we used to collect faults and vulnerabilities in a file and to collect complexity, code churn, and fault history metrics. We also provide descriptive statistics of the metrics. We collected faults reported since the release of Firefox 2.0 and before the release of Firefox 3.0 from the Mozilla Bugzilla[4] bug database. These faults include the faults for the minor releases of Firefox 2.0. Each bug report includes the details of a bug including bug description, bug resolution method, bug status, and bug patches. Bug resolution method indicates the method by which a bug has been resolved such as FIXED, WONTFIX, and DUPLICATE. Bug status indicates the life cycle of a bug such as NEW, ASSIGNED, VERIFIED, RESOLVED, and CLOSED. From a bug patch, we can identify the files that have been changed to remove faults. We only considered the bug patches that were approved or reviewed by the module owner.

The bug reports include the details for both faults and non-faults. Non-faults include functional enhancements and refactoring as well as performance optimization. In this study, we are interested only in faults. However, the bug reports do not have explicit information that we can distinguish between faults and non-faults other than natural language description. To facilitate the classification of faults and non-faults, we used automated text classification. The details of the automated text classification are described in Section 4.2. In our prediction models, we considered only the bug reports that were classified as faults by the automated text classification and whose resolution method and status were “FIXED and RESOLVED” or “FIXED and VERIFIED”, and that had patches written in C/C++. We considered a file faulty if a bug report that had bug patches for the file was classified as a fault. Over 80,000 bug reports have been reported between the

[4] https://bugzilla.mozilla.org/

```
cross_validation (Set Data, int k, int r, int v) { // cross-validation with k folds and r repetition
    Data ← Perform data transformation on Data
    performance ← 0
        repeat r times {
            Randomly split Data into k stratified bins, B = {b1, b2, ..., bk}
            for each bin bi {
                Strain ←B – { bi }
                Stest ← bi
                Strain_vulnerable← vulnerable files in Strain
                N ← |Strain_vulnerable|
                Strain_neutral ← N neutral files remaining after random removal from Strain
                                 for under-sampling
                Strain ←Strain_vulnerable ∪ Strain_neutral
                V ← select v variables using the InfoGain selection method from Strain
                Train the model M on Strain with variables in V
                performance ← performance + prediction performance of M tested on Stest

            }
    }
    return performance ← performance / (k*r)
}
```

Fig. 4 Pseudo code for k x r cross-validation (Shin et al. 2011)

releases of Firefox 2.0 and Firefox 3.0. Among these, 27,016 bug reports were flagged as "FIXED and RESOLVED" or "FIXED and VERIFIED" and 15,571 of those bug reports had attachments. Among these, 6,967 bug reports had patches written in C/C++.

We collected vulnerabilities reported for Firefox 2.0 and its minor releases from the Mozilla Foundation Security Advisories (MFSAs).[5] Each MFSA includes bug IDs for the bug reports in the bug database. From these bug reports, we identified the files that had been changed to mitigate vulnerabilities. We considered a file vulnerable if there existed a bug report that had bug patches for the file among those bug reports linked from MFSAs for Firefox 2.0.

Firefox 2.0 consists of 11,051 files. Among them, 2,328 files were classified as faulty (21% of the total files), 363 files were vulnerable (3% of the total files), and 298 files were classified as both faulty and vulnerable (13% of the faulty files and 82% of the vulnerable files) as shown in Fig. 5. Among the 363 vulnerable files, 65 files were classified as non-faulty.

We collected complexity metrics from the initial release of Firefox 2.0. To collect the complexity metrics, we used a commercial source code analysis tool, Understand C++.[6]

To collect the code churn metrics, we searched the modified files from the CVS source code repository. We collected the changes that had been made for the 12 months before the release of Firefox 2.0. For the measurement of faults, vulnerabilities, and code churn, we only considered the changes made to C/C++ and their header files, excluding other files such as scripts since the complexity metrics were available only for C/C++ files.

To collect the fault history metric from the prior releases of Firefox 2.0, we collected faults from Firefox 1.0 and its minor releases using the same procedure we used to collect faults in Firefox 2.0. To count the number of past faults in a file, we counted the number of bug reports with patches written in C/C++ and made for the purpose of fixing faults for the file.

Although a large portion of vulnerabilities also have faults (82% of the vulnerable files), those vulnerable files corresponds to only 13% of the total faulty files. Therefore, the

[5] http://www.mozilla.org/security/known-vulnerabilities/
[6] http://www.scitools.com/

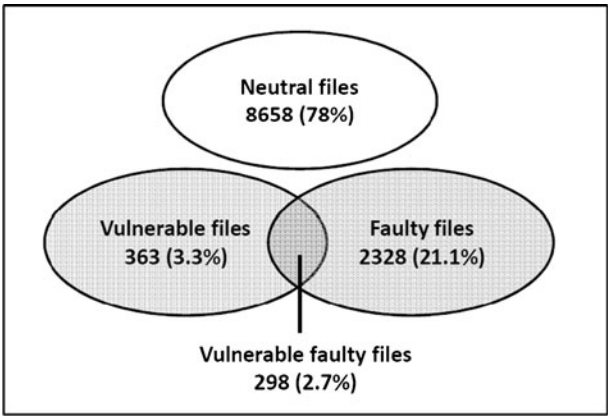

Fig. 5 Distribution of faulty and vulnerable files in Firefox 2.0

distribution of measurements for the metrics of faulty files and vulnerable files may differ. Figure 6 shows the box plots for the comparison of log scaled measurements between neutral files, faulty files, and vulnerable files for four representative metrics: LOC, NumChanges, LinesChanged, and NumPriorFaults. In Fig. 6, a metric value of 0 was substituted with 0.1 to perform log transformation. This is the reason why Fig. 6 shows negative values for some data points. The details of descriptive statistics including min, max, median, mean, quartiles, and standard deviation for all the metrics are provided in Table 11 in Appendix A. In Fig. 6, both faulty files and vulnerable files have higher complexity, more frequent and larger changes, and more past faults than the neutral files. More interestingly, vulnerable files tend to have higher complexity, more frequent and larger changes, and more past faults than faulty files in Firefox 2.0. All other metrics showed a similar tendency except for FanIn, FanOut, and CommentDensity. Considering

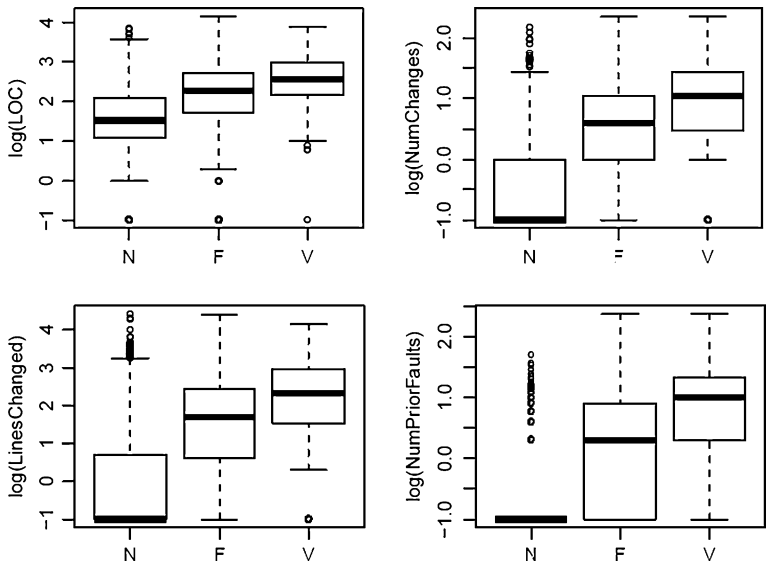

Fig. 6 Comparison of metric values for neutral, faulty, and vulnerable files

that both faulty and vulnerable files tend to have higher metric values than neutral files, and 82% of vulnerable files are also faulty, we can hypothesize that the vulnerability prediction performance from the fault prediction model and the vulnerability prediction model will be similar. Note that even though 82% of vulnerable files are faulty, if faulty files have positive correlations with the metrics while vulnerable files do not have the positive correlations, we cannot expect that both the fault prediction model and the vulnerable prediction model will provide the similar prediction performance.

4.2 Automated Text Classification of Faults and Non-faults

We performed automated text classification to classify bug reports as faults and non-faults because classifying the bug reports manually is labor intensive and can impede future repeated studies. In automated text classification, words in documents are used as features to compute the similarity between documents. Automated text classification has been used for various purposes including spam email filtering. Antoniol et al. (Antoniol et al. 2008) has also performed automated text classification of bug reports to classify faults and non-faults of the Mozilla project. Their logistic regression model provided 77% accuracy, 76% recall, and 82% precision in the best case. Because the result was promising, we decided to use an approach similar to theirs. However, we performed the classification only for the bug reports that had patches written in C/C++ for Firefox 1.0 and Firefox 2.0. The bug classification for Firefox 1.0 was performed to obtain the past faults of Firefox 2.0.

For classification of faults and non-faults, we first performed a feasibility study using a sample data set and then performed actual classification using the entire data set. For the feasibility study, we manually classified a set of randomly chosen bug reports and performed automated classification on the manually classified bug reports using logistic regression. The method of text classification is similar to the method for vulnerability prediction described in Section 3.4 only with different dependent and independent variables. The dependent variable is the probability of a bug report describing a fault. The independent variables are a set of words chosen from bug reports. That is, each term in a bug report such as “crash” or “error” is an independent variable and the weighted frequency of the term is the values of the independent variable. In the feasibility study, accuracy was 81%, recall was 88%, and precision was 86%. We detail the procedure of the feasibility study (Step 1) and actual classification (Step 2).

Step 1. Test the feasibility of automated bug report classification with sample bug reports.

Step 1.1 Determine the sample size.

A popular approach to determine the required sample size is to estimate classification accuracy by setting the range of confidence interval around the expected accuracy (Fitzpatrick-Linz 1981; Crews-Meyer and Hudson 2004). Here, the accuracy is defined as the fraction of correctly classified bug reports from the total bug reports. Equation 8 shows the formula to determine the sample size.

$$N = \frac{z_{\alpha/2}^2 pq}{E^2} \tag{8}$$

where p is the expected accuracy percentage, $q = 100 - p$, E is the allowable error, and $z_{\alpha/2}$ is the critical value of normal distribution for two-tailed significance level α. $z_{\alpha/2}$ is used to compute the confidence interval.

For our data set, we used 77% as the expected accuracy referring to the results for the Mozilla project provided by Antoniol et al (Antoniol et al. 2008). We used $z_{\alpha/2}$=1.96 for the 95% two-sided confidence level following conventional practice. There is no well-known rule to choose the value for E and we used 5% as is often used in the literature (Crews-Meyer and Hudson 2004) . Therefore, the required sample size to provide 77% accuracy with the 95% confidence level allowing 5% error is 272 as computed below:

$$N = \frac{1.96^2 * 77 * 23}{5^2} = 272$$

Note that 272 is the size of test data to be classified using the classification model. To build a classification model, we need training data. For this purpose, we manually classified 272*3=816 bug reports to perform three-fold cross-validation. Note also that we used three-fold cross-validation instead of ten-fold cross-validation because three-fold cross-validation is commonly used in practice (Witten and Frank 2005), and the main reason we performed the automated classification was to reduce manual effort of reading and classifying bug reports. Ten-fold cross-validation would require 2,720 bug reports to be manually classified.

Step 1.2 Manually classify the sample bug reports.

Using the sample size computed at Step 1.1, we manually classified 816 sample bug reports of Firefox 2.0. We used the titles and bug descriptions of bug reports for the classification. A few examples of the terms that frequently appear in the bug reports for faults are *crash*, *failure*, *error*, and *memory corruption*. A few examples of the terms frequently appear in the bug reports for non-faults include *refactor*, *add*, *implement*, *tidy up*, and *improve*. As a result, 585 bug reports were classified as faults and 231 bug reports were classified as non-faults.

Step 1.3 Preprocess the texts.

In this step, we normalized the texts in the bug reports by performing the standard preprocessing. That is, we first removed punctuations and converted all words to lower case. Then, we applied the standard Porter stemmer (Porter 1980) that removes suffixes from various forms of verbs (-ed, -ing, etc.) or plural forms of words. Finally we created a term frequency vector that included the frequency of words in each bug report.

Step 1.4 Build a classification model and evaluate the model.

Because not all terms are effective in classifying bug reports, the number of terms used as independent variables was chosen using the InfoGain variable selection method. Then, we built a logistic regression model using the terms selected by the InfoGain variable selection method. The number of words to be used was empirically determined by building the classification model with various numbers of independent variables. The model was evaluated using three-fold cross-validation and the whole process repeated ten times to reduce the bias caused by the random split of folds. Table 3 shows the mean results from 10×3 cross-validation when 10, 20, 50, and 100 words were used as independent variables. Because the model provided the best results when the number of chosen words was 50, we chose 50 variables for the final classification model used in Step 2.

Table 3 Mean of classification results

# of Words	Accuracy	Recall	Precision
10	73	87	79
20	76	87	82
50	81	88	86
100	80	88	85

Note that two of three projects studied by Antoniol et al. (Antoniol et al. 2008) also provided the best accuracy with 50 variables and the remaining one project also provided close to the best accuracy with 50 variables, indicating that the terms that distinguish between faults and non-faults are limited to around 50 terms. With 50 variables, the accuracy of automated classification was 81%, recall was 88%, and precision was 86%. Figure 7 shows the distribution of accuracy, recall, and precision from 10×3 cross-validation. The minimum performance for all three metrics was over 77%, and outliers were not observed for recall and precision. The two outliers for accuracy were 85.66% and 84.56%.

Step 2. Perform automated bug report classification for all bug reports.

Because the classification performance was reasonably high, we trained the classification model using the manually classified bug reports and used the model to classify all bug reports for Firefox 1.0 and Firefox 2.0 as detailed in the following sub-steps:

Step 2.1 Preprocess the texts.

This step is the same as Step 1.3.

Step 2.2 Build the classification model.

We selected 50 terms as independent variables using the InfoGain variable selection method and built a logistic regression model using the 816 manually classified bug reports. For Firefox 1.0, all 7,950 bug reports were classified using the logistic regression model. For Firefox 2.0, only the bug reports that were not classified manually were classified using the classification model.

Step 2.3 Perform automated bug report classification.

Table 4 summarizes the results of manual and automated classification. Around two thirds of the bug reports were classified as faults, and one thirds of the bug reports were classified as non-faults with a slight difference (3%) from manual classification.

4.3 Prediction Results

In this subsection, we present the results from the three experiments described in Section 3.

4.3.1 Experiment 1: Fault Prediction

Figure 8 shows the FF prediction results at classification thresholds 0.01 through 0.91 at an interval of 0.01. At each threshold, the performance metrics were computed by averaging

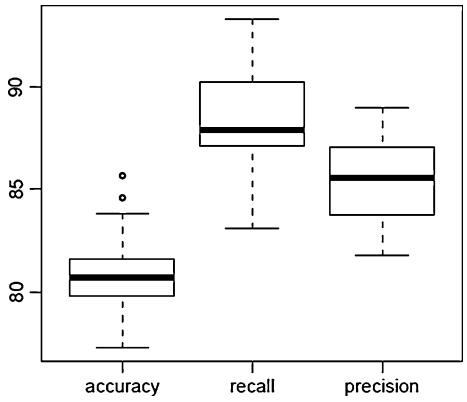

Fig. 7 Classification results with 50 terms as independent variables

the results from 100 runs of cross-validation. In all FF, VF, and VV predictions, precision was not computable after threshold 0.92 because no file was predicted as faulty in some of the test data sets that were used in the cross-validation. Therefore, we present results only up to the threshold of 0.91. The recall_{FF} slowly decreases from 100% to 52% up to threshold 0.8 and rapidly drops to 3% after threshold 0.8. Because precision and recall tend to trade off each other, precision increases from 21% to 90%. PF_{FF} is between approximately 100% to 0%. FI_F decreases from 100% to 1% and LI_F decreases from 100% to 9%.

Since there is no standard on prediction performance that indicates software quality is good enough, we consider 70% recall and 70% precision reasonable, as has been reported in other fault and vulnerability prediction studies (Menzies et al. 2007b; Zimmermann and Nagappan 2008; Neuhaus et al. 2007; Guo et al. 2004). Table 5 shows the prediction results at the highest threshold that the FF prediction provides reasonable recall_{FF} and at the lowest threshold that the FF prediction provides reasonable precision_{FF}. At recall_{FF} of 70% at threshold 0.52, around half of the predicted files in the FF prediction are false positives with FI_F of 30% and LI_F of 66%. At precision_{FF} of 73% at threshold 0.84, 36% of the files are correctly predicted as faulty. FI_F is low with 10%, but LI_F is fairly high with 43%.

As we discussed in Section 2.2, choosing the right thresholds depends on the type of software product and the availability of resources in a project team. A risk-adverse project requires high recall and a cost-adverse project requires high precision. A threshold can be determined based on previous prediction results. However, when the prediction model is built on the initial release of a software product that has no previous prediction results, a project team is likely to determine a threshold based on the amount of code to be inspected

Table 4 Summary of bug report classification

Release	# of Bug Reports Classified	# of Bugs Reports for Faults	# of Bug Reports for Non-Faults
Manual classification	816	585 (72%)	231 (28%)
Firefox 1.0 (model)	7,950	5,501 (69%)	2,449 (31%)
Firefox 2.0 (model)	6,151	4,262 (69%)	1,889 (31%)
Firefox 2.0 (model+manual)	6,967	4,847 (70%)	2,120 (30%)

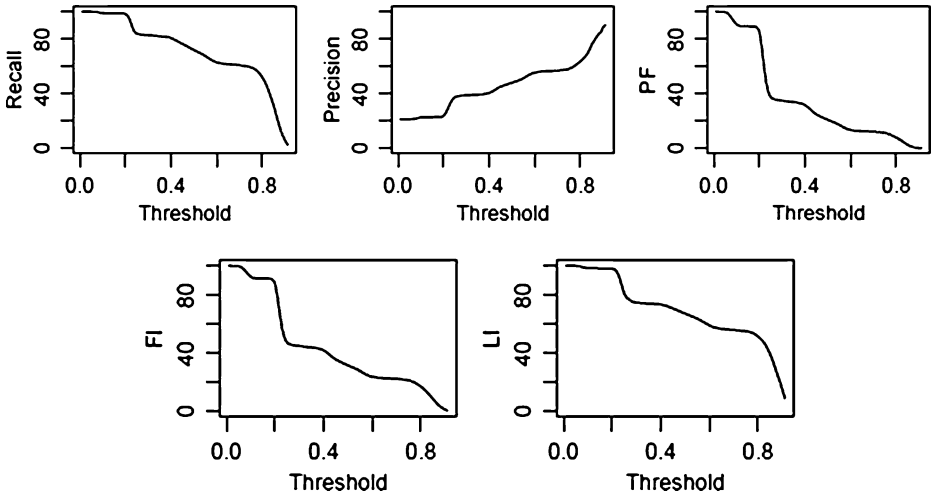

Fig. 8 Results of FF prediction at various thresholds

as a result of prediction or to use a middle point threshold 0.5. Therefore, we also present the prediction results with FI that is closest to 10% and the prediction results at threshold 0.5. For the FF prediction, the results at threshold 0.84 provide FI_F of 10%. The $recall_{FF}$ and $precision_{FF}$ at threshold 0.5 are 72% and 48%, respectively.

Figure 9 shows the variance of the FF prediction results at threshold 0.5 from the 100 runs of cross-validation .

In summary, the FF prediction provided either reasonable recall or reasonable precision of over 70% depending on thresholds, but not both at the same time.

4.3.2 Experiment 2: Vulnerability Prediction using a Fault Prediction Model

Figure 10 shows the VF prediction results at classification thresholds 0.01 through 0.91 at an interval of 0.01. The $recall_{VF}$ is between 100% and 10% and $precision_{VF}$ is between 3% and 55%. Similar to the FF prediction, the $recall_{VF}$ rapidly drops and $precision_{VF}$ rapidly increases after threshold 0.8.

Table 6 shows the prediction results at the highest threshold that the VF prediction provides reasonable $recall_{VF}$ and at the threshold that the VF prediction provides the highest $precision_{VF}$. At $recall_{VF}$ of 74% at threshold 0.81, only 16% of predicted files are actually vulnerable. The highest $precision_{VF}$ is 55% at threshold 0.91 with $recall_{VF}$ of 10% and does not reach a reasonable precision level. FI_F is 10% at threshold 0.84 with $recall_{VF}$ of 59% and $precision_{VF}$ of 19%. The $recall_{VF}$ and $precision_{VF}$ at threshold 0.5 are 91% and 9, respectively.

Table 5 Results of FF prediction at the interesting thresholds

Threshold	Recall	Precision	PF	FI	LI
0.50	72%	48%	21%	32%	67%
0.52	**70%**	49%	20%	30%	66%
0.84	36%	**73%**	4%	**10%**	43%

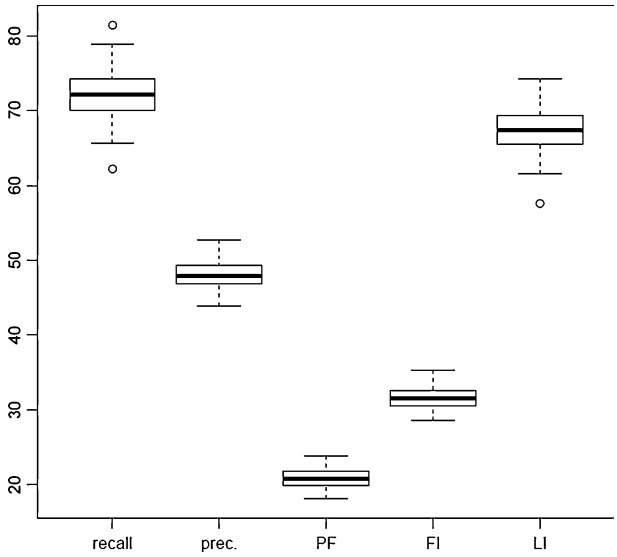

Fig. 9 Variance of FF prediction results at threshold 0.5

Figure 11 shows the variance of the VF prediction results at threshold 0.5 from the 100 runs of cross-validation.

In summary, the VF prediction provided reasonable recall with low precision. The highest precision was below 70%.

4.3.3 Experiment 3: Vulnerability Prediction using a Vulnerability Prediction Model

Figure 12 shows the VV prediction results at classification thresholds 0.01 through 0.91 at an interval of 0.01. The recall_{VV} is between 100% and 11% and precision_{VV} is between 4% and 52%. Similar to the FF prediction and the VF prediction, the recall_{VV} suddenly drops and precision_{VF} rapidly increases starting at threshold 0.8.

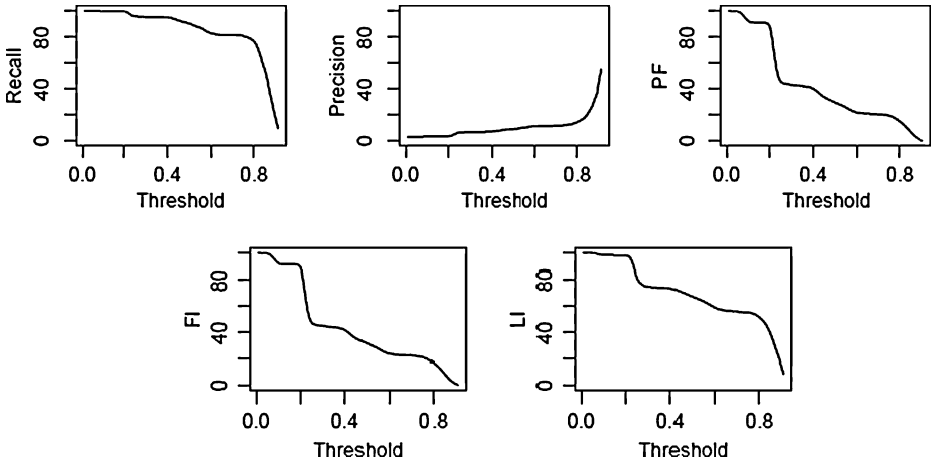

Fig. 10 Results of VF prediction at various thresholds

Table 6 Results of VF prediction at the interesting thresholds

Threshold	Recall	Precision	PF	FI	LI
0.50	91%	9%	30%	32%	67%
0.81	**74%**	16%	14%	16%	50%
0.91	10%	**55%**	0.3%	0.6%	9%
0.84	59%	19%	9%	**10%**	43%

Table 7 shows the prediction results at the highest threshold that the VV prediction provides reasonable recall_{VV} and at the threshold that the VV prediction provides the highest precision_{VV}. At recall_{VV} of 72% at threshold 0.74, only 16% of the predicted files are vulnerable. The highest precision_{VV} is 52% at threshold 0.91 with recall_{VV} of 15% and does not reach a reasonable level of precision. FI_V is 11% at threshold 0.79 with recall_{VV} of 62% and precision_{VV} of 20%.

Figure 13 shows the variance of the VV prediction results at threshold 0.5 from the 100 runs of cross-validation.

Note that in all the FF, VF, and VV predictions, the prediction models were optimized to provide high recall up to threshold around 0.8 and after that the prediction models were optimized to provide high precision. In Fig. 14, we notice that the VF prediction provides slightly higher recall and lower precision than the VV prediction across all thresholds in general by predicting more vulnerable files with more false positives. Therefore, we can expect that the recall and precision from the two predictions will become closer when we compare the results from the VV prediction at each threshold and the results from the VF prediction at a higher threshold than at the threshold for the VV prediction.

Figure 15 shows the comparison between the VV prediction and the VF prediction after shifting the curve for the VF prediction toward the left with a 0.1 threshold scale. After adjusting the compared thresholds, the results from the two predictions become closer than before adjusting the thresholds in the intersection of the recall-optimized regions from the

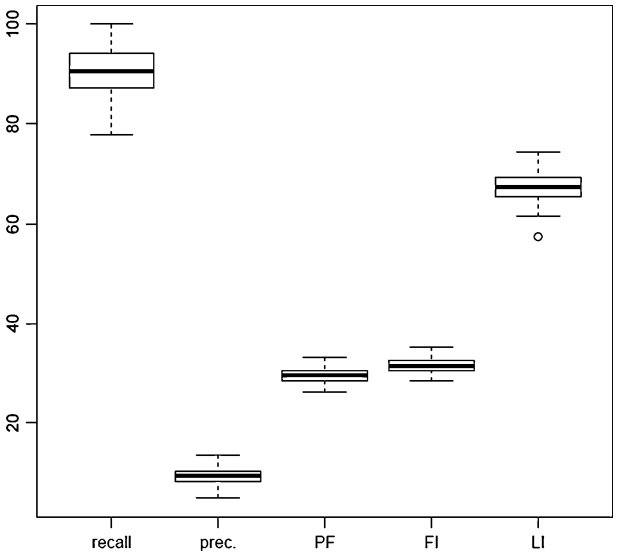

Fig. 11 Variance of VF prediction results at threshold 0.5

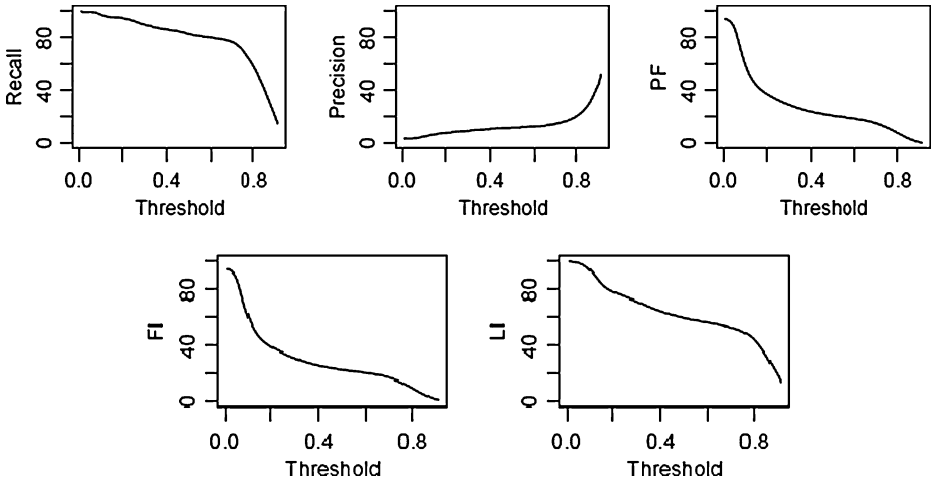

Fig. 12 Results of VV prediction at various thresholds

both predictions (0.01 to 0.70 for VV prediction and 0.11 to 0.80 for VF prediction). We further compared the prediction results using a *t*-test at each of 70 thresholds from threshold 0.01 to threshold 0.70 with Bonferroni correction to deal with erroneous hypothesis rejection from multiple hypothesis testing (70 hypothesis tests in this case). Recall, precision, and LI showed no statistically significant difference at the 0.05 significance level in over one third of the comparisons (43, 23, and 30 comparisons, respectively). However, when recall_{VF} was higher than recall_{VV}, precision_{VF} was lower than precision_{VV} with the maximum differences in recall and precision being 6% and 2%, respectively. In the precision-optimized regions, the trend was reversed after adjusting the thresholds: recall_{VV} was higher than recall_{VF}; precision_{VV} was lower than precision_{VF}. PF and FI showed a statistically significant difference at the 0.05 significance level in over 97% of the comparisons. PF_{VF} and FI_{VF} tend to be higher than PF_{VV} and FI_{VV} in the recall-optimized regions, as recall_{VF} is higher than recall_{VV}.

As an example, both the VV prediction at threshold 0.5 and the VF prediction at threshold 0.6 provided recall of 83% and LI of 60%. Precision, PF, and FI were over 11%, 21%, and 23%, respectively, with a 1% difference between the two predictions (see Table 8). If a security engineer wanted to find 83% of the vulnerable files by randomly selecting files, he/she would need to select 83% of the files for inspection. Compared with this random file selection, both vulnerability predictions reduced the number of files to be inspected by at least 71%.

Table 7 Results of VV prediction at the interesting thresholds

Threshold	Recall	Precision	PF	FI	LI
0.50	83%	12%	21%	23%	60%
0.74	**72%**	16%	13%	15%	51%
0.91	15%	**52%**	0.6%	1%	14%
0.79	62%	20%	9%	**11%**	46%

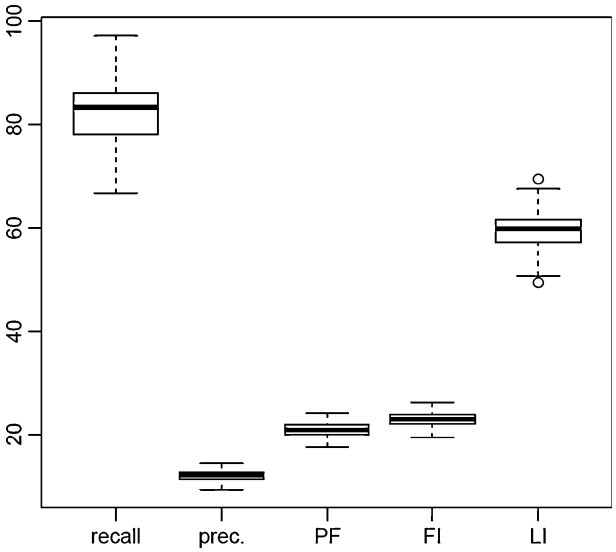

Fig. 13 Variance of VV prediction results at threshold 0.5

Figure 16 shows the variance of the VV prediction results at threshold 0.5 and the VF prediction results at threshold 0.6.

Although the prediction results from the VF prediction and the VV prediction are close, the sets of predicted vulnerable files can be different. Thus, we are interested in whether the fault prediction model can predict the vulnerable files that the vulnerability prediction model misses and vice versa. Therefore, we further examined the predicted files at thresholds 0.3, 0.5, and 0.7 for the VV prediction and thresholds 0.4, 0.6, and 0.8 for the VF prediction. Table 9 shows the numbers of correctly predicted vulnerable files from the VV prediction and the VF prediction across 100 runs of cross-validation. The smaller set from the two predictions at each threshold is in bold. In Table 9, at the three pairs of

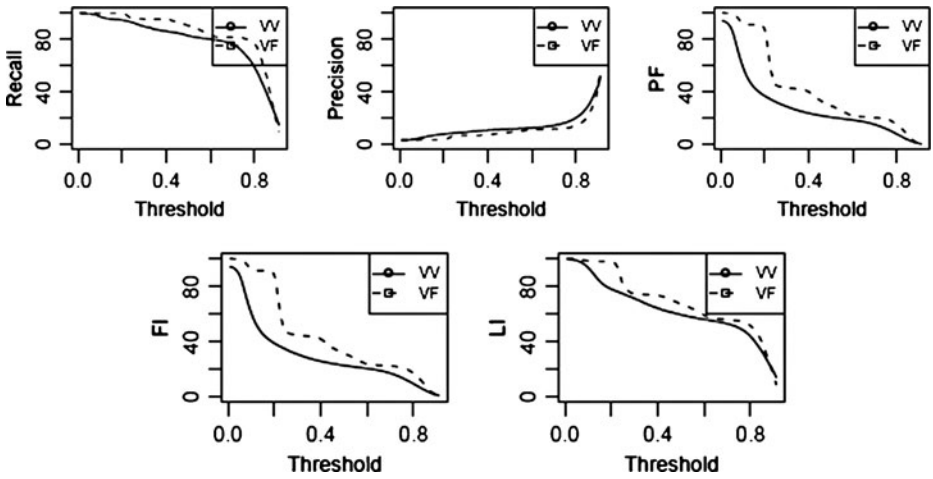

Fig. 14 Comparison of results between VF prediction and VV prediction at various thresholds

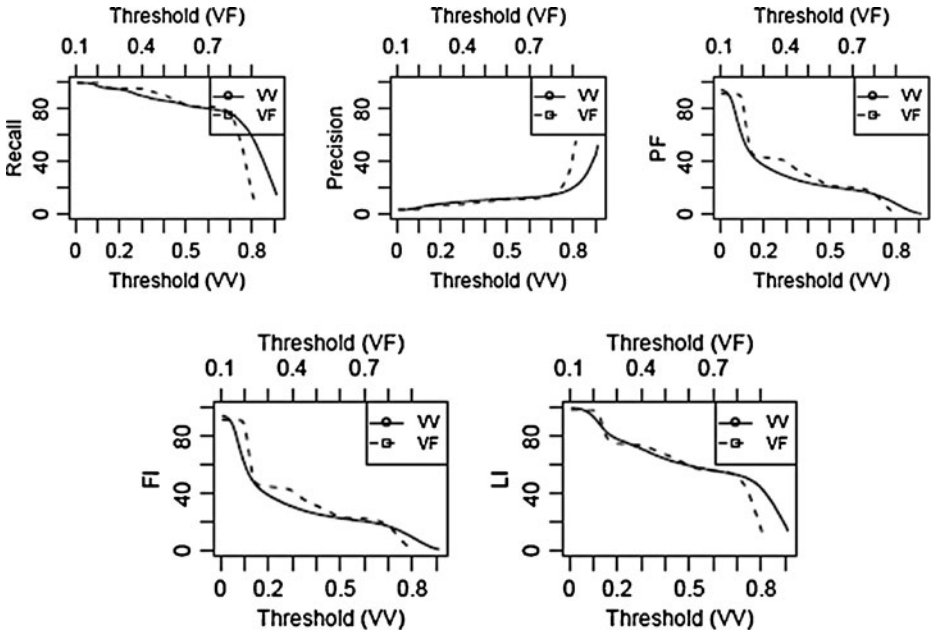

Fig. 15 Comparison of results between VF prediction and VV prediction at adjusted thresholds

thresholds, the difference in the numbers of predicted vulnerable files from the VV prediction and the VF prediction is at most 10. The differences between the smaller set and the intersection of the predicted files from the two predictions at each threshold is less than 2, indicating that the vulnerable files predicted by one model essentially subsumes the vulnerable files predicted by the other model at those thresholds.

Overall, we conclude that there is not sufficient evidence that VF prediction models and VV prediction models provide different prediction performance, indicating fault prediction models can be used as a substitute for vulnerability prediction models for Firefox 2.0 when traditional metrics are used.

4.4 Analysis of the "Needle Effect"

The fact that only a small percentage of files are vulnerable (3% of the total files and 13% of the faulty files) makes a vulnerability prediction akin to finding a needle in a haystack. Therefore, we can expect the performance of fault prediction and vulnerability prediction will be different as we have already seen in the FF prediction and the VV prediction. At the same time, the difference in the values of the three types of metrics between faulty files and

Table 8 Results of VV and VF prediction at the interesting thresholds

Threshold	Recall	Precision	PF	FI	LI
VV 0.50	83%	12%	21%	23%	60%
VF 0.60	83%	11%	22%	24%	60%

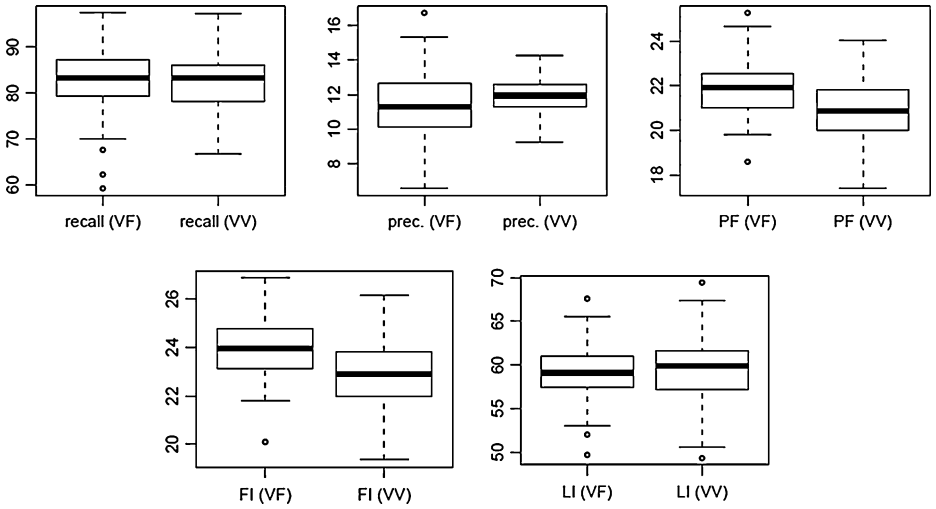

Fig. 16 Prediction results of VV prediction at threshold 0.5 and VF prediction at threshold 0.6

vulnerable files, as described in Section 4.1, can lead to the difference in prediction performance. Figure 17 compares the results from the FF prediction and the VV prediction at threshold 0.5. The VV prediction provides higher recall, but much lower precision than the FF prediction.

We investigated whether this difference occurred largely because vulnerabilities were more rare occasions compared with faults ("needle effects") by adjusting the number of the faulty files in the fault prediction. Our hypothesis was that if the difference in the prediction performance mainly came from the difference in the number of reported faults and vulnerabilities, the performance of the FF prediction and the VV prediction would be similar if we adjusted the number of faulty files. To test this hypothesis, we removed a set of randomly-selected faulty files. Figure 18 presents the prediction performance averaged over 100 runs of 10×10 cross-validation after removal of faulty files at threshold 0.5. In Fig. 18, FF and VV represent the fault prediction model and the vulnerability prediction model that we used in Experiment 1 and Experiment 3. FF*n* indicates a fault prediction model used *n* times as many faulty files as vulnerable files after removing the randomly-selected faulty files. The numbers in the parentheses in Fig. 18 represent the numbers of faulty files or vulnerable files used for the models. Table 10 shows the results of pairwise *t*-tests after Bonferroni correction to deal with multiple hypothesis testing. In Table 10, '√' indicates the difference in the prediction performance was significantly different at the 0.05 significance level.

Table 9 Comparison of the predicted files from VV and VF prediction

Thresholds	Correctly predicted vulnerable files in VV (1)	Correctly predicted vulnerable files in VF (2)	Intersection between (1) and (2)
VV 0.3–VF 0.4	**336**	346	334
VV 0.5–VF 0.6	312	**306**	305
VV 0.7–VF 0.8	291	**286**	284

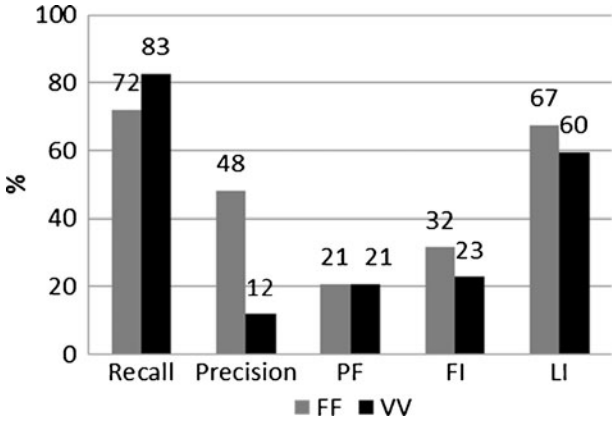

Fig. 17 Comparison of the results of FF prediction and VV prediction

For the fault prediction models, recall is stable across the models using various numbers of faulty files in Fig. 18 and was not statistically significantly different. However, precision dramatically increases when the number of faulty files increases. Although PF was statistically significantly different in around half of the pairwise comparisons, the values of PF are very close between all the fault prediction models. FI and LI from the fault prediction models gradually increase according to the increase in the number of faulty files.

Between the VV prediction model and the FF1 prediction model, where the numbers of vulnerable files and faulty files are equal, the VV prediction model provides 10% higher recall and 1% lower precision than the FF1 prediction model on average. PF and FI are not statistically significantly different between the two models. The VV prediction model provides 7% higher LI than the FF1 prediction model.

The dramatic change of precision depending on the number of faulty files indicates that precision is greatly affected by the needle effect in general. The difference in recall between the VV prediction model and the FF1 prediction model may have been caused by the difference in the distribution of metrics values between faulty and vulnerable files as we have seen in Fig. 6. The high recall from the VV prediction model compared with the FF1 prediction model and the similar precision and PF between those two models suggest that

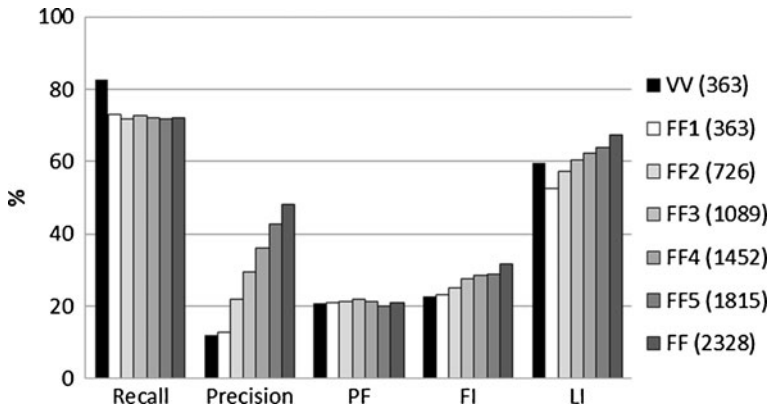

Fig. 18 Effects of the number of faulty and vulnerable files

Table 10 Pairwise comparison of prediction performance after faulty file removal

	Recall						Precision						PF						FI						LI					
	F1	F2	F3	F4	F5	F	F1	F2	F3	F4	F5	F	F1	F2	F3	F4	F5	F	F1	F2	F3	F4	F5	F	F1	F2	F3	F4	F5	F
V	√	√	√	√	√	√	√	√	√	√	√	√	–	–	√	–	√	–	–	√	√	√	√	√	√	√	–	√	√	√
F1		–	–	–	–	–		√	√	√	√	√		–	√	–	√	–		√	√	√	√	√		√	√	√	√	√
F2			–	–	–	–			√	√	√	√			–	–	√	–			√	√	√	√			√	√	√	√
F3				–	–	–				√	√	√				–	√	√				√	√	√				√	√	√
F4					–	–					√	√					√	–					–	√					√	√
F5						–						√						√						√						√

the traditional fault prediction metrics are more effective for vulnerability prediction than fault prediction when the numbers of faulty files and vulnerable files are equal.

The radical change in precision was also observed in Menzies et al.'s study. They showed that precision is an unstable performance measurement for a highly unbalanced data set (Menzies et al. 2007a). For this reason, they preferred PF to precision. Although having only a small percentage of the minor class does not always indicate low precision depending on a data set, metrics, and modeling techniques, we also observed low precision and the instability of precision in this study from both fault prediction and vulnerability prediction. However, we prefer precision to PF as a measure of prediction performance, because development teams would act on the files predicted as being faulty or vulnerable rather than on the files predicted as non-faulty or non-vulnerable, and precision is a more direct measure of unnecessary code inspection and testing effort than PF. Additionally, because PF was almost the same across all prediction models at certain thresholds in our study (e.g. PF at threshold 0.5 in Tables 5 and 7), PF is not an appropriate measure to compare models in our case.

In summary, the difference in the distribution of measures of complexity, code churn, and fault history metrics as well as the difference in the distribution of discovered faulty files and vulnerable files seem to bring the difference in prediction performance. The metrics used in our study for Firefox 2.0 are more effective for vulnerability prediction than fault prediction assuming the same numbers of reported faulty files and vulnerable files. However, low precision in vulnerability prediction compared with fault prediction in reality is largely attributable to the rarity of vulnerabilities.

5 Discussion

Overall, the models using the traditional fault prediction metrics provided high recall and low precision in vulnerability prediction from both the VV prediction and the VF prediction at threshold up to 0.8 and low recall and medium precision after threshold 0.8. This clear distinction of recall- and precision-optimized regions depending on classification thresholds indicates that security engineers must judiciously choose a classification threshold depending on their goals of model application and classification techniques.

When the numbers of faulty files and vulnerable files are the same, the performance of the VV prediction using the traditional fault prediction metrics was better than the performance of the FF prediction. In both the fault and vulnerability predictions, the most frequently selected metrics by the InfoGain variable selection method were NumPriorFaults, NumChanges, LinesChanged, LinesInserted, LinesDeleted, CountLineCode, and CountLineCodeDecl.

The high recall from the vulnerability predictions can be attributed to the fact that the files with high complexity, frequent and large changes, and many past faults tend to have more vulnerabilities than the files with low complexity, less frequent and small changes, and a small number of past faults. However, the precision of the vulnerability prediction was lower (10% on average after adjusting the classification threshold) than the precision of the fault prediction (48%) because only a small percentage of files was vulnerable. This dependence on the amount of reported vulnerabilities in vulnerability prediction has two implications. First, if the amount of the reported vulnerabilities is small just because the latent vulnerabilities have not been discovered yet, we can expect a portion of the false positives may actually be true positives that will be reported as vulnerable files as time passes or may remain latent due to the lack of malicious intent on the part of attackers. If

so, time spent for inspecting and testing the predicted vulnerable files may be worthwhile. Second, if the number of reported vulnerabilities is actually small even after considerable time has passed after the release of software, it would be difficult to expect high precision from a vulnerability prediction using the traditional fault prediction metrics in general.

Our results show that the fault prediction model provides prediction performance similar to the vulnerability prediction model across a wide range of thresholds. When those thresholds are used, reliability test engineers and security engineers would inspect the same code areas predicted by the fault prediction models. One can argue that if the two teams use exactly the same information provided by the fault prediction model and spend their resources on the same code areas, security engineers will be much less efficient than reliability test engineers, given that vulnerabilities are only a small portion of faults. This is indeed true, and security inspection is costly compared to non-security inspection, because security engineers have to find rarely-existing vulnerabilities using their own knowledge and skills. Although reliability test engineers and security engineers can inspect the same code areas, their focus should be different. Therefore, we believe prioritizing the code locations to inspect vulnerabilities first with objective measurable evidence can guide security engineers.

Alhazmi and Ray (Alhazmi et al. 2007) reported that the ratio of vulnerabilities to the total number of faults was 1% to 5% in their study with five versions of Microsoft Windows operating systems and two versions of Red Hat Linux systems. In our study, the ratio of vulnerable files to the total faulty files is 15%. Although the number of files to be inspected is reduced by over 71% compared with random file selection in the VF prediction at threshold 0.5 and the VV prediction at threshold 0.6, 24% of 11,051 files is still many files (2,652) for inspection. Using high thresholds improves inspection efficiency. However, if a project team wants to detect as many vulnerable files as possible using a vulnerability prediction model, further prioritization should be followed using an expert's judgment or from using other methods such as static analysis tools in addition to the use of vulnerability prediction. Note that static analysis tools alone cannot guide the security inspection and testing because static analysis tools are also known to have a high percentage of false positives, and the results from static analysis tools also require prioritization (Kim and Ernst 2007; Heckman and Williams 2008).

Although VF prediction and VV prediction provide similar results in many classification thresholds, a project team should also consider the ease of data collection. For Firefox, vulnerability information can be obtained more easily and more precisely than fault information.

6 Threats to Validity

Construct validity refers to whether we are measuring what we are supposed to measure. We used the fault prediction metrics that have been frequently used and effective in fault prediction in prior studies. However, other fault prediction metrics may provide different results. The number of faulty and vulnerable files can vary depending on the methods of fault and vulnerability collection. For Firefox, we considered a file faulty or vulnerable when there was at least one bug report with patches to fix faults or vulnerabilities in the file. However, patches may not have been committed to the source code repository, or vulnerabilities and faults may have been fixed, but the patches may not have been posted on the bug database. Nonetheless, considering the maturity of the development process for

Firefox, we believe the missing counts of vulnerabilities and faults are not at the level that can threaten our results.

We assumed the efficiency of inspection is proportional to the number of files and the lines of code to be inspected. However, the efficiency of inspection may vary depending on the complexity of the problem implemented in the code and the importance of the code in terms of security. Since these factors are not readily obtainable in an objective way, experts' judgment should be accompanied when the models are used in organizations.

The classification of faults and non-faults for Firefox using the automated text classification technique is not perfect. However, we believe that the 88% recall and 86% precision for the training data set is reasonably high.

Conclusion validity refers to whether our conclusion is based on sound statistical methods. To avoid making an arbitrary conclusion, we used statistical tests to compare the prediction results.

External validity concerns whether our results can be generalized beyond the data set we used in this study. Since our study was performed on a single project, further replicated studies are required to generalize our observations. However, we believe our study increases the understanding on vulnerability prediction and reveals concerns about the application of vulnerability prediction in organizations and in future research.

7 Summary

We investigated whether fault prediction models can be used for vulnerability prediction or if specialized vulnerability prediction models should be developed when both are built with the traditional fault prediction metrics of complexity, code churn, and fault history. We examined the effectiveness of those metrics for vulnerability prediction on the Mozilla Firefox 2.0 web browser. In our study, the fault prediction model and the vulnerability prediction model provided similar prediction performance for vulnerability prediction through a wide range of classification thresholds. Overall, both the VV prediction and the VF prediction models using the traditional fault prediction metrics provided high recall and low precision in vulnerability prediction at thresholds up to 0.8 and provided low recall and a medium level of precision after threshold 0.8. The low precision from vulnerability prediction can primarily be explained by the small number of reported vulnerabilities.

Another important finding is that code with vulnerabilities tends to be more complex, to have more frequent and larger changes, and to have more past faults than code with faults. Our analysis on Firefox 2.0 indicates that fault prediction models provide capability similar to vulnerability prediction models for predicting vulnerabilities based upon traditional metrics. Therefore, convenience in data collection can be an important factor to decide whether to use a fault prediction model or a vulnerability prediction model to predict vulnerabilities. However, both models require improvement to reduce false positives in vulnerability prediction while providing high recall.

Acknowledgment This work was supported in part by the National Science Foundation Grant No. 0716176 and the CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also thank the NCSU Software Engineering Realsearch group for their reviews for the initial version of this paper. We appreciate Dr. Robert Bell and Dr. Raffaella Settimi for their advice on statistics. Most of all, we thank the editors and reviewers of Empirical Software Engineering journal for their thorough reviews and helpful suggestions.

Appendix A Comparison of Metrics Values for Neutral, Faulty, and Vulnerable Files

Table 11 Descriptive statistics for metrics of neutral, faulty, and vulnerable files for Firefox 2.0

	File Type	Min	1st Q	Median	Mean	3rd Q	Max	Std.
LOC	N	0	12	33	134	124	7014	317
	F	0	53	187	476	521	14126	849
	V	0	142	363	867	960	7778	1243
CountLineCodeDecl	N	0	6	17	50	47	5152	160
	F	0	29	70	149	164	14128	365
	V	0	62	139	257	305	2740	340
CountDeclFunction	N	0	0	1	7	8	256	15
	F	0	1	11	23	31	401	36
	V	0	7	24	42	52	401	55
AvgEssential	N	0	0	1	1	2	104	3
	F	0	1	1	2	2	111	4
	V	0	1	1	2	2	43	4
SumEssential	N	0	0	3	26	13	1139	41
	F	0	1	21	64	72	1438	123
	V	0	9	49	126	135	1438	214
AvgCyclomaticStrict	N	0	0	1	3	4	390	7
	F	0	1	3	4	6	641	14
	V	0	1	4	4	6	74	6
SumCyclomaticStrict	N	0	0	4	30	27	1694	80
	F	0	1	41	129	147	2236	241
	V	0	17	105	250	279	2236	392
MaxCyclomaticStrict	N	0	0	2	8	8	595	20
	F	0	1	9	21	26	1666	52
	V	0	3	17	38	43	1666	108
AvgMaxNesting	N	0	0	0	0	1	8	1
	F	0	0	0	0	1	9	1
	V	0	0	0	1	1	4	1
SumMaxNesting	N	0	0	0	2	1	401	10
	F	0	0	0	3	1	188	12
	V	0	0	0	4	1	188	15
MaxMaxNesting	N	0	0	0	1	1	16	1
	F	0	0	0	1	1	9	1
	V	0	0	0	1	1	9	2
AvgFanIn	N	0	0	0	2	1	1769	24
	F	0	0	0	2	2	933	20
	V	0	0	0	2	3	89	6
SumFanIn	N	0	0	0	13	2	5309	90
	F	0	0	0	22	3	3881	121
	V	0	0	0	34	4	3881	224
MaxFanIn	N	0	0	0	5	2	5296	64
	F	0	0	0	5	3	933	27

Table 11 (continued)

	File Type	Min	1st Q	Median	Mean	3rd Q	Max	Std.
	V	0	0	0	6	4	307	24
AvgFanOut	N	0	0	0	3	4	63	5
	F	0	0	0	2	3	138	5
	V	0	0	0	2	3	29	4
SumFanOut	N	0	0	0	10	6	1592	41
	F	0	0	0	15	4	832	52
	V	0	0	0	17	4	832	63
MaxFanOut	N	0	0	0	4	5	177	8
	F	0	0	0	4	4	156	10
	V	0	0	0	5	4	156	12
CommentDensity	N	0	0	1	3	3	124	5
	F	0	0	0	1	1	183	5
	V	0	0	0	1	1	8	1
NumChanges	N	0	0	0	1	1	151	4
	F	0	1	4	10	11	224	18
	V	0	3	11	25	27	224	36
LinesChanged	N	0	0	0	59	5	26095	507
	F	0	4	50	450	279	25302	1444
	V	0	34	216	1081	924	14073	2185
LinesInserted	N	0	0	0	34	2	16542	298
	F	0	2	26	254	147	17849	872
	V	0	17	107	577	508	9097	1253
LinesDeleted	N	0	0	0	25	1	10307	221
	F	0	1	17	196	111	10702	636
	V	0	14	87	503	414	7226	1033
LinesNew	N	0	0	0	14	0	6989	124
	F	0	0	0	93	31	10396	438
	V	0	0	2	174	64	4832	557
NumPriorFaults	N	0	0	0	1	0	50	2
	F	0	0	2	7	8	234	15
	V	0	2	10	19	21	234	29

*. N: Neutral files, F: Faulty files, V: Vulnerable files

References

Alhazmi OH, Malaiya YK, Ray I (2007) Measuring, analyzing and predicting security vulnerabilities in software systems. Comput Secur 26(3):219–228

Antoniol G, Ayari K, Penta MD, Khomh F, Guéhéneuc Y-G (Oct. 27–30 2008) Is it a bug or an enhancement? A text-based approach to classify change requests. In: 2008 Conference of the Center for Advanced Studies on Collaborative Research, Ontario, Canada.

Arisholm E, Briand LC (Sep. 21–22 2006) Predicting fault-prone components in a Java Legacy System. In: the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil, pp. 8–17.

Arisholm E, Briand LC, Fuglerud M (5–9 Nov. 2007) Data mining techniques for building fault-proneness models in telecom Java Software. In: 18th IEEE Int'l Symposium on Software Reliability Engineering (ISSRE'07), Trollhättan, Sweden, pp. 215–224.

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Software Eng 22(10):751–761

Crews-Meyer KA, Hudson PF (2004) Landscape complexity and remote classification in Estern Costal Mexico: applications of Landsat-7 ETM^{+} Data. Geocarto International 19 (1).

Fitzpatrick-Linz K (1981) Comparison of sampling procedure and data analysis for a Land-Use and Land-Cover Map. Photogramm Eng Rem Sens 47(3):343–351

Gegick M, Rotella P, Williams L (2009) Toward non-security failures as a predictor of security faults and failures. Paper presented at the International Symposium on Engineering Secure Software and Systems, Leuven, Belgium, February 04–06.

Gegick M, Williams L, Osborne J, Vouk M (Oct. 27 2008) Prioritizing software security fortification through code-level metrics. In: 4th ACM workshop on Quality of protection, Alexandria, Virginia, pp 31–38.

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Software Eng 26(7):653–661

Guo L, Ma Y, Cukic B, Singh H (2004) Robust prediction of fault-proneness by random forests. In: the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, Bretagne, France, pp 417–428.

Hassan AE (2009) Predicting faults using the complexity of code changes. In: the 31st International Conference on Software Engineering, pp 78–88.

Heckman S, Williams L (Oct. 9–10 2008) On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: 2nd International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, pp 41–50.

IEEE (1988) IEEE Std 982.1-1988 IEEE standard dictionary of measures to produce reliable software. IEEE Computer Society.

Jiang Y, Cukic B, Menzies T (2008a) Can data transformation help in the detection of fault-prone modules? In: Proceedings of the 2008 Workshop on Defects in Large Software Systems (DEFECTS'08), Seattle, Washington, pp 16–20.

Jiang Y, Cukic B, Menzies T (10–14 Nov. 2008b) Cost curve evaluation of fault prediction models. In: 19th International Symposium on oftware Reliability Engineering (ISSRE'08), pp 197–206.

Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto K (20–21 Sept. 2007) The effects of over and under sampling on fault-prone module detection. In: 1st International Symposium on Empirical Software Engineering and Measurement, Madrid, Spain, pp 196–204.

Khoshgoftaar TM, Allen EB, Kalaichelvan KS, Goel N (1996) Early quality prediction: a case study in telecommunications. IEEE Software 13(1):65–71

Kim S, Ernst MD (Sep. 3–7 2007) Which warnings should i fix first? In: the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp 45–54.

Kim S, Zimmermann T, E. James Whitehead J, Zeller A (2007) Predicting faults from cached history. In: the 29th International Conference on Software Engineering, pp 489–498.

Krsul IV (1998) Software vulnerability analysis. PhD dissertation, Purdue University, West Lafayette.

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Software Eng 34(4):485–496

Mark A. Hall, Holmes G (2003) Benchmarking attribute selection techniques for discrete class data mining. IEEE Trans Knowl Data Eng 15 (3).

McCabe TJ (1976) A complexity measure. IEEE Trans Software Eng 2(4):308–320

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE'09), Vancouver, Canada.

Meneely A, Williams L (November 2009) Secure open source collaboration: an empirical study of Linus' Law" computer and communications security. In: Computer and Communications Security (CCS), Chicago, IL, pp 453–462.

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with precision: a response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". IEEE Trans Software Eng 33(9):637–640

Menzies T, Greenwald J, Frank A (2007b) Data mining static code attributes to learn defect predictors. IEEE Trans Software Eng 33(1):2–13

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code feature: current results, limitations, new approaches. Autom Softw Eng 17(4):doi:10.1007/s10515-010-0069-5

Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (May 2008) Implications of ceiling effects in defect predictors. In: the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08), Leipzig, Germany, pp 47–54.

Nagappan N, Ball T (May 15–21 2005) Use of relative code churn measures to predict system defect density. In: the 27th International Conference on Software Engineering, St. Louis, MO, USA, pp 284–292.

Nagappan N, Ball T, Zeller A (May 20–28 2006) Mining metrics to predict component failures. In: the 28th International Conference on Software Engineering, Shanghai, China, pp 452–461.

Neuhaus S, Zimmermann T, Zeller A (October 29–November 2 2007) Predicting vulnerable software components. In: the 14th ACM Conference on Computer and Communications Security (CCS'07), Alexandria, Virginia, USA, pp 529–540.

NIST (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards & Technology.

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Software Eng 31(4):340–355

Ostrand TJ, Weyuker EJ, Bell RM (July 9–12 2007) Automating algorithms for the identification of fault-prone files. In: the 2007 International Symposium on Software Testing and Analysis (ISSTA'07), London, UK, pp. 219–227.

Ott RL, Longnecker M (2001) An introduction to statistical methods and data analysis, 5th edn. Duxbury, Pacific Grove

Porter MF (1980) An algorithm for suffix stripping. Program 16(3):130–137

Rice D (2007) Geekonomics: The real cost of insecure software. Addison-Wesley Professional,

Shin Y, Williams L (Oct. 27 2008) Is complexity really the enemy of software security? In: the 4th ACM Workshop on Quality of Protection, Alexandria, Virginia, USA, pp. 47–50.

Shin Y, Meneely A, Williams L (2011) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Software Eng.

Witten IH, Frank E (2005) Data mining: practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann Publishers, Boston

Zimmermann T, Nagappan N (10–18 May 2008) Predicting defects using network analysis on dependency graphs. In: the 13th International Conference on Software Engineering, pp. 531–540.

Zimmermann T, Nagappan N, Williams L (Apr. 6–11 2010) Searching for a needle in a haystack: predicting security vulnerabilities for Windows Vista. In: 3rd International Conference on Software Testing, Verification and Validation, Paris, France, pp. 421–428.

Yonghee Shin received the BS degree in computer science from Sookmyung Women's University in Korea and the MS degree in computer science from Texas A&M University. She received the PhD degree in computer science from North Carolina State University (NCSU) under the advice of Dr. Laurie Williams. She is currently working as a postdoctoral researcher at DePaul University. Her research interests are in software engineering focusing on software metrics, software reliability and security, empirical software engineering, requirements traceability, and software testing. She worked for Daewoo telecommunications and Samsung SDS in Korea for eight years before returning to academia.

Laurie Williams received the BS degree in industrial engineering from Lehigh University, the MBA degree from the Duke University Fuqua School of Business, and the PhD degree in computer science from the University of Utah. She is a professor in the Computer Science Department at North Carolina State University (NCSU). Her research focuses on agile software development practices and processes, software reliability, software testing and analysis, software security, healthcare information technology, and broadening participation and increasing retention in computer science. She is the director of the NCSU Science of Security Lablet, the research director of the NCSU Institute for Next Generation IT Systems, and an area representative for the Secure Open Systems Initiative. She worked for IBM Corporation for nine years in Raleigh, North Carolina, and Research Triangle Park, North Carolina, before returning to academia. She is a member of the IEEE.