Can Fault Prediction Models and Metrics be Used for Vulnerability Prediction?

Yonghee Shin and Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
{yonghee.shin, lawilli3}@ncsu.edu

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 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 20% 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 predicted vulnerabilities with high recall (over 90%) and low precision (9%). The precision from these vulnerability predictions was much lower than the precision from fault prediction (47%). Our results suggest that fault prediction models based upon traditional metrics can be substituted for specialized vulnerability prediction models, but requires significant improvement to reduce false positives.**

Keywords- Software metrics; complexity metrics; fault prediction; vulnerability prediction; open source projects

I. INTRODUCTION

Finding faults in software systems early in the development life cycle is important to reduce software maintenance costs that are caused by later fixes [1]. With limited time and budget in development teams, efficient allocation of inspection and testing efforts are also critical. Hence, prediction of code locations that have high possibility of having faults using the software metrics that are available early in the development life cycle has been an active research area (representative publications [2-7]). Recently, the importance of predicting software vulnerabilities (security problems) in software systems early in the development life cycle is being emphasized (representative publications [8-13]). Considering the explosive growth of vulnerability reports in recent years [14], and the immeasurable cost of insecure software [15], this growing emphasis is no surprise.

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 [16] and in changes to the code [17]. Hence, complexity metrics and code churn metrics have been used for fault prediction [5, 17, 18]. Additionally, problematic code areas in one release tend to be also problematic again in later releases [4]. These similarities between vulnerabilities and faults may enable us to use the traditional fault prediction metrics – complexity, code churn, and fault history metrics for vulnerability prediction.

At the same time, vulnerabilities and faults are different in that attackers actively seek vulnerabilities with malicious or criminal intent, while faults are exposed based upon the normal use of software. This difference necessitates software engineers who conduct vulnerability detection to think like an attacker and have special security skills. As a result, a model trained for vulnerability prediction rather than fault prediction may be necessary. Additionally the reported vulnerabilities are much fewer than the reported faults in many projects [19]. Finding vulnerabilities is akin to finding a "needle in a haystack", so perhaps a vulnerability prediction model must be trained to find needles and not haystacks.

These similarities and differences motivate us to empirically investigate the effectiveness of traditional fault prediction metrics to predict vulnerabilities. *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.*

To achieve our goal, we performed an empirical case study on a widely-used open source project: the Mozilla Firefox[1] 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 files. 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.

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

The contributions of this study are as follows:

- We provided empirical evidence that traditional fault prediction metrics can detect a high portion of vulnerable files, but should be significantly improved to reduce false positives.
- We provided empirical evidence that fault prediction models and vulnerability prediction models provide similar prediction performance when they use the traditional fault prediction metrics and can be used interchangeably.
- We showed that fault and vulnerability predictions are largely affected by the number of the reported faults and vulnerabilities.
- As a by-product of our fault data gathering, we showed that automated text classification is feasible and useful to classify faulty files.

The rest of this paper is organized as follows: Section II provides background and related work. Section III describes study design including metrics, modeling techniques, and evaluation methods. Section IV provides the results of our case study. Section V discusses the implication of the results. Section VI mentions threats to validity. Section VII summarizes our study.

II. Background and Related Work

This section provides the terms we use in this study and discusses prior studies related with our study. We also provide the binary classification evaluation criteria for prediction.

A. Terms

A *software fault* is "an accidental conditional that causes a functional unit to fail to perform its required function [20]."

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" [21]. 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.

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*.

B. 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 [2-7, 17, 18]. We summarize a few representative studies on fault prediction using complexity, code churn, and fault history metrics below.

Nagappan et al. [18] performed a post-release failure prediction using complexity metrics on five Microsoft software systems. Subsets of the complexity metrics were statistically significantly correlated with the post-release failures in all five projects. However, no single set of complexity metrics was correlated with post-release failures with all five projects. The researchers were able to find a combination of complexity metrics which could significantly predict post-release failures in all the five projects.

Nagappan and Ball [5] also performed a post-release failure prediction using relative code churn metrics on Windows Server 2003. Their multiple regression model 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 modules when counting both true positives and true negatives.

Graves et al. [17] observed frequently changed modules and the modules with large changes tend to have more faults than other modules. They also observed the recent changes induce more faults than older changes.

Ostrand et al. [4] and Arisholm et al. [22] have used the number of faults in prior releases for fault prediction and found the fault history is useful for fault prediction.

Vulnerability Prediction Neuhaus et al. [11] performed a vulnerability prediction on the Mozilla open source project by analyzing the import (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% components predicted as vulnerable.

Gegick et al. [13] 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. [9] also performed a study to investigate whether the failures can be used as an indicator of vulnerabilities. Their case study on a Cisco software system found 57% of vulnerable components in the top nine percent of the total component ranking, but with a high percentage of false positives (48%).

Meneely and Williams [8] 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.

Shin and Williams [12] 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 [10] also compared the prediction performance between fault prediction models and vulnerability prediction models on the Mozilla JavaScript engine at function level. The study discussed in this paper uses a larger project, uses additional metrics, uses an improved data training technique, uses different prediction performance measurements, and

provides in-depth analysis on the use of fault prediction models and metrics for the purpose of vulnerability prediction.

None of the above studies investigated whether fault prediction models can be used to predict vulnerabilities and compared the effectiveness of fault prediction models and vulnerability prediction models in depth as in our study.

C. 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 over 0.5 in our study. 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 waste of resources in code inspection and testing. An FN happens when a file is classified as neutral when it is not. FNs can allow software systems to be released with faults and vulnerabilities. Correct classifications are called True Positives (TP) and True Negatives (TN). Table I summarizes the four types of classification.

TABLE I. BINARY CLASSIFICATION RESULTS

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

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

Recall is the ratio of the correctly classified true positives to the actual positives as defined in the following formula:

$$recall = TP / (TP + FN) \quad (1)$$

Precision is the ratio of the correctly classified true positives to the predicted positives as defined in the following formula:

$$precision = TP / (TP + FP) \quad (2)$$

III. 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. We broke the goal into three specific questions, as will be discussed. In subsections A through D, we will describe the dependent and independent variables for the prediction models, measurements of prediction performance, and the modeling techniques that will be used to answer the questions.

Q1*: Can traditional fault prediction metrics predict faults with reasonable recall and precision?*

We answer this question to provide a baseline of comparison with the performance of vulnerability prediction. We build 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. We call the fault prediction using a fault prediction model a *FF prediction*. Then, we measure the recall and precision of the fault prediction.

Reasonable levels of recall and precision might differ depending on the criticality of the projects. Since there is no standard on prediction performance that indicates software quality good enough, we consider 70% recall and 70% precision as reasonable, as has been reported in other fault and vulnerability prediction studies [2, 6, 11, 23]. Note that not many studies reported both recall and precision or provided both high recall and high precision at the same time. Ref. [2] provided only recall (71% on average), but not precision. In [11], precision was 70%, but recall was 45%. Only [6] provided 70% recall and 70% precision. In [23], among the 56 predictions using various classification techniques on five projects, only 14 predictions provided over 70% recall, but precision was not reported. Note that all these studies have been performed on different projects, or used different metrics or modeling techniques, or performed on different sizes of entities such as component or file. Therefore, interpretation and comparison of the prediction results from different studies require caution.

We present the relationships between metrics, models, and the purpose of the prediction in Fig. 1 through Fig. 3.

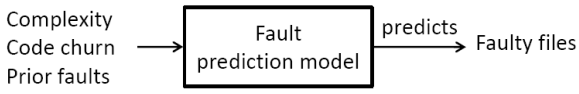

Figure 1. Fault prediction using a fault prediction model (FF prediction)

Q2: *Can fault prediction models predict vulnerabilities with reasonable recall and precision?*

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 build the same fault prediction model as we built for Q1 and count the number of correctly predicted vulnerable files contained in the list of predicted faulty files. We call the vulnerability prediction using a fault prediction model a *VF prediction.* The result of the VF prediction is measured in terms of the recall and precision of the correctly predicted vulnerable files. We consider that a fault prediction model is effective for vulnerability prediction if both recall and precision are higher than 70%.

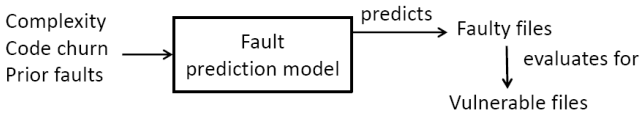

Figure 2. Vulnerability prediction using a fault prediction model (VF model)

Q3*: Can vulnerability prediction models using traditional fault prediction metrics predict vulnerabilities with reasonable recall and precision?*

To answer this question, we build 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 measure the recall and precision of the predicted vulnerable files. We call the vulnerability prediction using a vulnerability prediction model a *VV prediction.* We consider that a vulnerability prediction model is effective if both recall and precision are higher than 70%.

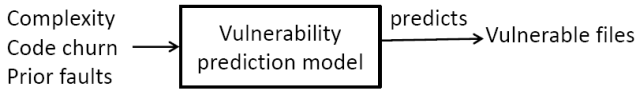

Figure 3. Vulnerability prediction using a vulnerability prediction model (VV model)

A. Dependent and Independent Variables

In our study, the dependent variable for a fault prediction model is the probability that a file will have at least one fault. The dependent variable for a vulnerability prediction model is the probability that a file will have at least one vulnerability.

Table II provides the definitions of the complexity, code churn, and fault history metrics that are used as independent variables in this study.

B. Evaluation Criteria for Prediction Performance

To clarify recall and precision from the three types of predictions (FF, FV, and VV), we redefine recall and precision that we defined in Section II.C as follows:

$Recall_{FF}$ is the ratio of the correctly classified faulty files to the actual faulty files using a fault prediction model.

$Recall_{VF}$ and $Recall_{VV}$ are the ratios of the correctly classified vulnerable files to the actual vulnerable files using a fault prediction model and a vulnerability prediction model, respectively.

We define $precision_{FF}$, $precision_{VF}$, and $precision_{VV}$ in the same way as we define $recall_{FF}$, $recall_{VF}$, and $recall_{VV}$.

If organizations have to inspect too many files only to find a small percentage of faults and vulnerabilities, the models are 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 ratio of files to be inspected to the total number of files as a result of prediction as in the following formula:

$$FI = (TP + FP) / (TP + FP + TN + FN) \quad (3)$$

FI_V and FI_F are the ratios of files to be inspected to the total files as a result of vulnerability prediction and fault prediction, respectively. FI_V=0.2 and $recall_V$=0.8 indicates that 80% of the vulnerable files can be found in the 20% of the predicted files. 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.

TABLE II. DEFINITIONS OF METRICS

Metric	Definition
CountLineCode	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 flow 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 assignment to the parameters to call a function or global variables. We used the average, sum, and maximum values for each file.
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.
NumPriorFaults	The number of faults in the prior release.

Even though a file can be a logical unit for code inspection, a large file may require more effort to maintain than small files. Therefore, we additionally measure *Line Inspection ratio (LI). LI* is the ratio of lines of source code to be inspected to the total lines of code 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 as in the following formula:

$$LI = (TP_{LOC} + FP_{LOC}) / (TP_{LOC} + FP_{LOC} + TN_{LOC} + FN_{LOC}) \quad (4)$$

LI_V and LI_F are the ratios of lines of code to be inspected to the total lines of code as a result of vulnerability prediction and fault 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, a high value of *LI* does not necessarily mean that the model is inefficient. If we predict faults and vulnerabilities at a finer granularity such as function level rather than file level, the model may result in smaller *LI*. Therefore, we value *FI* than *LI* as a better evaluation criterion.

For ease of read, we use percentage instead of ratio in the rest of this paper. For example, recall 0.8 will be presented as 80% recall.

C. Prediction Models

We used a logistic regression technique to predict faulty files and vulnerable files. Logistic regression techniques have been effective in fault prediction [3, 6, 18].

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) [24]. This way the outcome has a value range of 0 to 1. The formula for the logit and the probability of an event resolved from the logit are as follows:

$$\text{logit} = \ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n = z \quad (5)$$

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

We used the logistic regression implemented in Weka 3.7 tool[2].

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. [25] 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 are as good as using many variables [2]. By finding the small set of the most effective variables, we can focus on the most effective variables as well as reducing the time for data collection, model training, and prediction. We used a variable selection method, InfoGain, to select seven variables that provide the highest information gain [26]. Except for this variable selection option, we used the default options for the logistic regression implemented in Weka.

After we train 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 10x10 cross-validation [26]. In 10x10 cross-validation, nine folds of data are used as a training data set and one fold of data is used as a test data set. Each fold is used exactly once as a test data set. Then the ten folds cross-validation is repeated ten times. Therefore, we performed 100 times of prediction for each model and report the average of the results in this paper.

The numbers of files with faults or vulnerabilities are much smaller than the numbers 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 [26], the minor class often has high false negatives. Considering the impact of a single exploited vulnerability, lowering false negatives are relatively more important than lowering false positives in vulnerability prediction. To deal with this issue, we applied data sampling that has been effective in fault prediction with unbalance data [27, 28]. Specifically 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 become equal. We performed the under-sampling for each run of 10x10 cross-validation.

Although logistic regression does not assume normal distribution of data and therefore does not require data transformation, recall has improved by log transformation in practice. We provide the results of log transformation in this study.

IV. A CASE STUDY: MOZILLA FIREFOX BROWSER

Mozilla Firefox is a widely-used open source web browser. Firefox is written in C/C++ and consists of 11,051 files and over 2 million lines of source code. In this study, we used Firefox 2.0.

A. Data Collection

We collected faults reported since the release of Firefox 2.0 and before the release of Firefox 3.0 from the Mozilla Bugzilla[3] bug database. Each bug report includes the details of a bug including bug description, bug status, bug resolution method, and bug patches. Bug status indicates the life cycle of a bug such as *NEW*, *ASSIGNED*, *VERIFIED*, *RESOLVED*, and *CLOSED*. Bug resolution indicates the method by which a bug has been resolved such as *FIXED*, *WONTFIX*, and *DUPLICATE*. From a bug patch, we can identify the files that have been changed to remove faults. The bug reports include both enhancements and faults, when we are interested only in faults. However, the bug reports do not have explicit information that we can automatically distinguish between enhancements and faults. To facilitate the classification of enhancements and faults, we used automated text classification. The details of the automated text classification are described in Section IV. B. We considered only the bug reports that were classified as faults

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

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

by the automated text classification and whose status became *RESOLVED* or *VERIFIED* by fixing the faults. We counted the number of bug reports that have bug patches for a file as a surrogate measure of the number of faults in a file. Over 80,000 bug reports have been reported between the releases of Firefox 2.0 and Firefox 3.0. Among these, 6,965 bug reports were flagged as *RESOLVED* or *VERIFIED* status after the bugs have been fixed and had patches written in C/C++.

We collected vulnerabilities reported for Firefox 2.0 to Firefox 2.0.0.19 from the Mozilla Foundation Security Advisories (MFSAs)[4]. Each MFSA includes bug IDs for the bug reports in the bug database. From these bug reports, we identified the files that have changed to mitigate vulnerabilities. To count the number of vulnerabilities in a file, we counted the number of bug reports that are included in MFSAs and that have bug patches for the file. Among 11,051 total files, 2,261 files were classified as faulty (20% of the total files), 363 files were vulnerable (3% of the total files), and 294 files were both faulty and vulnerable as in Fig. 4. Note that a file can have both vulnerabilities and faults and 80% of the vulnerable files also have faults in Firefox 2.0.

To collect the complexity metrics, we used a commercial source code analysis tool, Understand C++[5].

To collect the code churn metrics, we searched the modified files from the CVS source code repository. We collected the changes that have 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 release of Firefox 2.0, we collected faults from Firefox 1.0 and its minor releases using the same procedure we used for Firefox 2.0. Table III shows the means and medians of five representative metrics per file. In Table III, both faulty files and vulnerable files have higher complexity, more frequent and larger changes, and more past faults than neutral files. Of note, vulnerable files tend to have higher complexity, more frequent and larger changes, and more past faults than faulty files in Firefox 2.0.

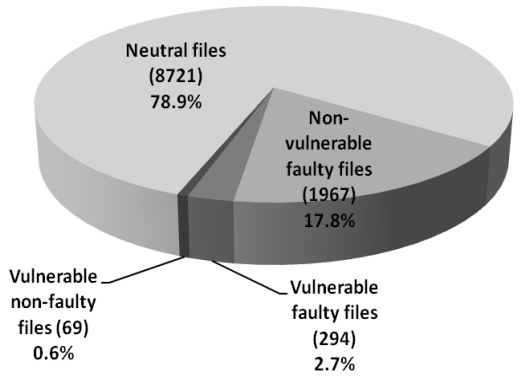

Figure 4. Distribution of faulty and vulnerable files in Firefox 2.0

TABLE III. COMPARISON OF MEAN AND MEDIAN VALUES OF METRICS

	Neutral Files		Faulty Files		Vulnerable Files	
	Mean	**Med.**	**Mean**	**Med.**	**Mean**	**Med.**
LOC	136	33	480	193	867	363
SumCyclomaticStrict	30	4	136	33	250	105
NumChanges	1	0	10	4	25	11
LinesChanged	61	0	451	52	1081	216
NumPriorFaults	1	0	8	4	19	8

B. *Classification of Fault and Enhancement*

We performed automated text classification to classify bug reports as enhancements or faults because classifying the bug reports manually would take too long time and 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. [29] has also performed automated text classification of the bug reports for faults and enhancements of the Mozilla project. Their logistic regression model provided 76% recall and 82% precision. We performed a similar approach to their study. However, our classification has been performed only for the bug reports that have patches written in C/C++ for Firefox 2.0. We performed the bug report classification in the following four steps:

Step 1. Create a training data set. To create a training data set, we manually classified 600 sample bug reports as fault or enhancement using the titles and bug descriptions as texts. A few examples of the terms that frequently appear in the bug reports for faults are *failure*, *error*, and *memory corruption*. A few examples of the terms frequently appear in the bug reports for enhancements include *add*, *implement*, and *improve*.

Step 2. Test the feasibility of automated bug classification with the sample bug reports. The process is performed in the following sub-steps.

Step 2.1. Preprocess the texts. In this step, we first removed punctuations and converted all words to lower case. Then, we applied the standard Porter stemmer [30] that removes suffixes from various forms of verbs (-ed, -ing, etc.) or plural forms of words. Finally we created a term frequency vector that includes frequency of words in each bug report.

Step 2.2. Split the term frequency vector into training and testing sets. In this step, we split the 90% of the instances in the term frequency vector as a training data set and 10% of them as a test data set.

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

Step 2.3. Build classification models and perform cross-validation. We chose 100 words as independent variables using the InfoGain variable selection method and built a logistic regression model using the training data set. Then we classified bug reports in the test data set using the model. We repeated Step 2.2 and Step 2.3 100 times for 10x10 cross-validation. The logistic regression model provided 88% recall and 85% precision. We consider this performance is enough for our purpose of fault classification. Therefore, we continue to Step 3.

Step 3. Perform bug classification of all 6,965 bug reports. The process is performed in the following two sub-steps.

Step 3.1. Preprocess the texts. Use the same method in Step 2.1.

Step 3.2. Build a classification model and perform classification. In this step, we used the whole set of manually-classified sample bug reports from Step 1 to train the model. Then we classified the remaining 6,365 bug reports using the trained model.

We used Weka 3.7 for the preprocessing and classification.

C. Prediction Results

***Q1**: Can the traditional fault prediction metrics predict faults with reasonable recall and precision?*

To answer this question, we measured $recall_{FF}$, $precision_{FF}$, FI_F, and LI_F. As shown in Fig. 5, the fault prediction model predicted 74% of the total faulty files ($recall_{FF}$) in 33% of the total files (FI_F,) and in 67% of the total lines of code (LI_F). Among the files predicted as faulty, 47% was correctly predicted ($precision_{FF}$). Compared with random file selection, the fault prediction model reduced the number of files to be inspected by 56% to detect 74% of the total faulty files.

In summary, the FF prediction provided reasonable recall (over 70%), but low precision (below 70%). Although the precision is not high enough, we continue to answer Q2 and Q3 to see whether the traditional fault prediction metrics are more (or less) effective in vulnerability prediction than in fault prediction.

***Q2**: Can fault prediction models predict vulnerabilities with reasonable recall and precision?*

To answer this question, we measured $recall_{VF}$ and $precision_{VF}$. Because we use the model that we used to answer Q1, the amount of inspection (FI_F and LI_F) is the same as the results from Q1. As shown in Fig. 6, the fault prediction model predicted 92% of the total vulnerable files ($recall_{VF}$) in 33% of the total files (FI_F,) and in 67% of the total lines of code (LI_F). Among the files predicted as vulnerable, 9% was correctly predicted ($precision_{VF}$). Compared with random file selection, the fault prediction model reduced the number of files to be inspected by 64% to detect 92% of the total vulnerable files.

In summary, the VF prediction provided very high recall (92%), but much lower precision (9%) than our 70% criterion and the precision from the FF prediction. Although the VF prediction effectively reduced the number of files to be inspected, still a large amount of unnecessary inspection and testing is expected because of the low precision.

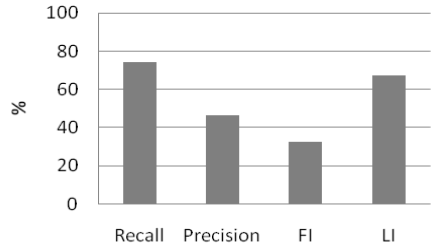

Figure 5. Fault prediction results using fault prediction models (FF prediction)

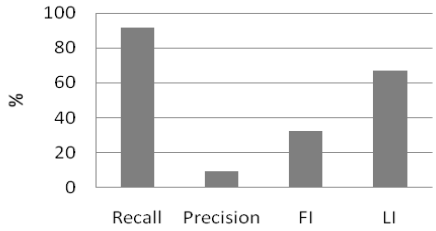

Figure 6. Vulnerability prediction results using fault prediction models (VF prediction)

***Q3**: Can vulnerability prediction models using traditional fault prediction metrics predict vulnerabilities with reasonable recall and precision?*

To answer this question, we measured $recall_{VV}$, $precision_{VV}$, FI_V, and LI_V. The vulnerability prediction model correctly predicted 84% of the total vulnerable files ($recall_{VV}$) in 23% of the total files (FI_V,) and in 60% of the total lines of code (LI_V). Among the files predicted as vulnerable, 12% was correctly predicted ($precision_{VV}$). Compared with random file selection, the vulnerability prediction model reduced the number of files to be inspected by 72% to detect 84% of the total vulnerable files. Fig. 7 presents the results of the VV prediction and compares them with the results from the VF prediction in Q2.

Since the VV prediction provided worse performance in recall and better performance in precision than the VF prediction, we wondered whether the result will be the same if we change the threshold for the binary classification of vulnerable and non-vulnerable files. After we changed the classification threshold of the VV prediction to 0.7, all the performance measures between the two types of the models became very close (Fig. 8). The recall and precision for the VF prediction were 92% and 9%, respectively. The recall and precision for the VV prediction became 90% and 3%, respectively. This result suggests that fault prediction models can be used as a substitute for vulnerability prediction models when traditional metrics are used. We consider the reason that the two types of models provided the similar performance is because a high percentage of files with vulnerabilities also have faults in Firefox 2.0; 80% of the vulnerable files also have faults in Fig. 4.

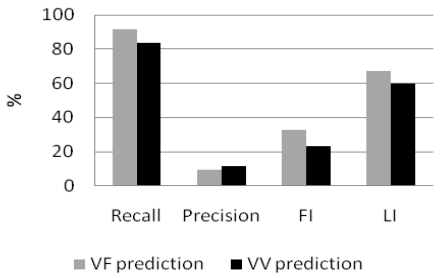

Figure 7. Comparison of the results of VF prediction and VV prediction

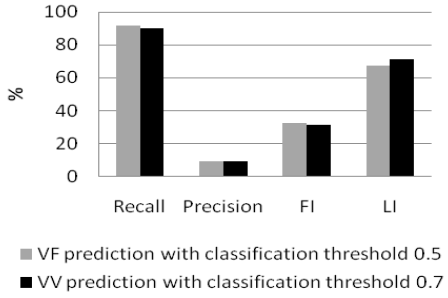

Figure 8. Comparison of the results of VF prediction and VV prediction with adjusted classification threshold

In summary, the VV prediction provided high recall, but low precision. Additionally the VV prediction provided similar performance to the VF prediction when the classification threshold was adjusted, suggesting fault prediction models can be used as a substitute for vulnerability prediction.

D. Analysis of the "Needle Effect"

Only 3% of the files and only 13% of the faulty files are vulnerable files (Fig. 4), making a vulnerability prediction a task of finding a needle in a haystack. Therefore, we expect the performance of fault prediction and vulnerability prediction will be different. At the same time, we observed the difference in the measurements of the three types of metrics between faulty files and vulnerable files, which also can lead to the difference in prediction performance. As we expected, we can observe a fairly large difference in performance between the FF prediction and the VV prediction from the answers for Q1 and Q3. Fig. 9 puts the results from Q1 and Q3 together. The VV prediction provides 18% higher recall and 38% lower precision than the FF prediction.

We investigated whether this difference occurs largely because vulnerabilities are more rare occasions compared with faults (needle effects) by adjusting the number of faulty files in fault prediction. Our hypothesis is that if the difference in the prediction performance is mainly because of the difference in the numbers of the reported faults and vulnerabilities, the performance of the FF prediction and the

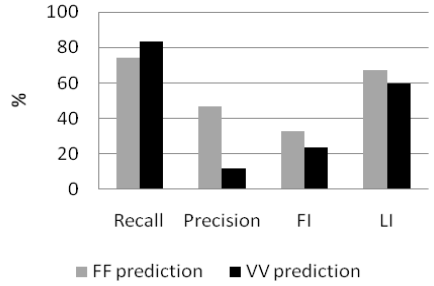

Figure 9. Comparison of the results of FF prediction and VV prediction

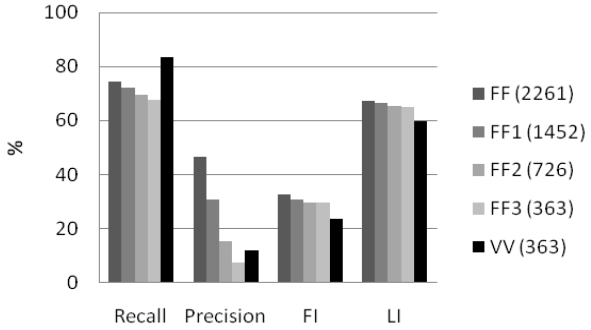

Figure 10. Effects of the number of faulty and vulnerable files

VV prediction will be similar if we adjust the number of faulty files. To test this hypothesis, we randomly selected a subset of faulty files and marked them as neutral assuming those files have not been reported as faulty. In Fig. 10, FF and VV represent the fault prediction model and the vulnerability prediction model that we used to answer Q1 and Q3. FF1, FF2, and FF3 are the fault prediction models built with as three times, as twice, and the same number of faulty files as the number of vulnerable files, respectively, after the remaining faulty files are marked as neutral. The number of faulty and vulnerable files used for the models are presented in Fig. 10.

In Fig. 10, $recall_{FF}$, FI_{FF}, and LI_{FF} from the fault predictions (FF, FF1, FF2, and FF3) are only slightly different depending on the difference in the numbers of faulty files. However, $precision_{FF}$ becomes dramatically lower when the number of faulty files becomes small. These results show that precision is greatly affected by the needle effect in general. However, the noticeable difference in recall, FI, and LI between the four FF predictions and the VV prediction may have been caused by the actual difference in the characteristics between faulty and vulnerable files that we have already observed in Table III. Especially, the performance from the VV prediction is better than the performance from the FF prediction in all the performance measurements when the numbers of the faulty files and the vulnerable files are the same. This result suggests that the traditional fault prediction metrics are even more effective for vulnerability prediction than fault prediction depending on the numbers of reported faults and vulnerabilities.

In summary, precision is largely affected by the amount of reported faults and vulnerabilities. However, the differences in the measures of complexity, code churn, and fault history metrics between faulty and vulnerable files also brings the difference in prediction performance.

V. Discussion

Overall, the traditional fault prediction metrics provided similarly high recall and low precision in vulnerability prediction from both the VV prediction and the VF prediction. When the numbers of faulty files and vulnerability 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 reason of the high recall from the vulnerability predictions can be attributed to the fact that files with high complexity, frequent and large changes, and many past faults tend to have more vulnerabilities than files with low complexity, less frequent and small changes, and small past faults. However, the precision of the vulnerability predictions was much lower (9% after adjusting the classification threshold) than the precision of the fault prediction (47%) 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 large portion of the false positives could be actually true positives that will be reported as vulnerable files as time passes. If so, it is worth to spend extra efforts to inspect and test the predicted vulnerable files. Second, if the number of reported vulnerabilities is actually small even after enough time has passed after the release of software, it will be difficult to expect high precision from a vulnerability prediction using the traditional fault prediction metrics in general.

Alhazmi and Ray [19] reported that the ratio of vulnerabilities to the total number of faults was 1-5% 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 16%. Although the number of files to be inspected is reduced by over 64% compared to random file selection in both the VF prediction and the VV prediction, 33% of 11,051 files is still many files (3,647). Therefore, further prioritization should be followed from 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 [31, 32].

VI. Threats To Validity

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 faults and vulnerabilities in a file can vary depending on the methods of fault and vulnerability collection. For Firefox, we counted the number of faults and vulnerabilities by counting the number of bug reports with bug patches for a file. However, patches may have not been committed to the source code repository, or vulnerabilities and faults have been fixed but the patches have not been posted on the bug database. However, considering the maturity of the development process of 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 enhancements for Firefox is not perfect and has room to be improved. However, we believe that the 88% recall and 85% precision for the training data set is reasonably high.

Since our study has been done to a single project, further replicated studies are required to generalize our observations to other projects. However, we believe our study increases the understanding on vulnerability prediction and reveals concerns that should be cared in the application of vulnerability prediction in organizations and in future research.

VII. 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 results for vulnerability prediction. Both the fault prediction model and the vulnerability prediction model predicted vulnerabilities with high recall of over 90% and effectively reduced the number of files to be inspected after adjusting the classification threshold. However, precision was very low (9%) leading to a waste of resources in security inspection and testing primarily because that the number of reported vulnerabilities was small. Our analysis on Firefox 2.0 indicates that fault prediction models based upon traditional metrics can be substituted for specialized vulnerability prediction models, but requires significant improvement to reduce false

positives. Finding better metrics that predict vulnerable code locations is our ongoing research.

ACKNOWLEDGMENT

This work is 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 and especially Andy Meneely for their careful reviews and helpful suggestions on the paper.

REFERENCES

[1] B.W. Boehm, Software engineering economics, Prentice-Hall Inc., 1981.

[2] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Software Eng., vol. 33, no. 1, 2007, pp. 2-13.

[3] V.R. Basili, L.C. Briand, and W.L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Trans. Software Eng., vol. 22, no. 10, 1996, pp. 751-761.

[4] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Predicting the location and number of faults in large software systems," IEEE Trans. Software Eng., vol. 31, no. 4, 2005, pp. 340-355.

[5] N. Nagappan, and T. Ball, "Use of relative code churn measures to predict system defect density," Proc. the 27th International Conference on Software Engineering, St. Louis, MO, USA, May 15-21 2005, pp. 284-292.

[6] T. Zimmermann, and N. Nagappan, "Predicting defects using network analysis on dependency graphs," Proc. the 13th International Conference on Software Engineering, Leipzig, Germany, 10 - 18 May 2008, pp. 531-540.

[7] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, and N. Goel, "Early quality prediction: A case study in telecommunications," IEEE Software, vol. 13, no. 1, 1996, pp. 65-71.

[8] A. Meneely, and L. Williams, "Secure open source collaboration: An empirical study of linus' law" computer and communications security," Proc. Computer and Communications Security (CCS), Chicago, IL, November 2009, pp. p453-462.

[9] M. Gegick, P. Rotella, and L. Williams, "Toward non-security failures as a predictor of security faults and failures," Proc. International Symposium on Engineering Secure Software and Systems, Leuven, Belgium, February 04-06 2009, pp. 135-149.

[10] Y. Shin, and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," Proc. International symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany2008, pp. 315-317.

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

[12] Y. Shin, and L. Williams, "Is complexity really the enemy of software security?," Proc. the 4th ACM Workshop on Quality of Protection, Alexandria, Virginia, USA, Oct. 27 2008, pp. 47-50.

[13] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification through code-level metrics," Proc. 4th ACM workshop on Quality of protection, Alexandria, Virginia, Oct. 27 2008, pp. 31-38.

[14] G. McGraw, Software security: Building security in, Addison-Wesley, 2006.

[15] D. Rice, Geekonomics: The real cost of insecure software, Addison-Wesley Professional, 2007.

[16] T.J. McCabe, "A complexity measure," IEEE Trans. Software Eng., vol. 2, no. 4, 1976, pp. 308-320.

[17] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Trans. Software Eng., vol. 26, no. 7, 2000, pp. 653-661.

[18] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," Proc. the 28th International Conference on Software Engineering, Shanghai, China, May 20-28 2006, pp. 452-461.

[19] O.H. Alhazmi, Y.K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," Computers & Security, vol. 26, no. 3, 2007, pp. 219-228.

[20] IEEE, "IEEE std 982.1-1988 IEEE standard dictionary of measures to produce reliable software," The Institute of Electrical and Electronics Engineers, 1988.

[21] I.V. Krsul, "Software vulnerability analysis," PhD dissertation, Purdue University, 1998.

[22] E. Arisholm, and L.C. Briand, "Predicting fault-prone components in a java legacy system," Proc. the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil, Sep. 21-22 2006, pp. 8-17.

[23] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," Proc. the 15th International Symposium on Software Relaibility Engineering (ISSRE'04), Saint-Malo, Bretagne, France2004, pp. 417-428.

[24] R.L. Ott, and M. Longnecker, An introduction to statistical methods and data analysis, 5th ed., Duxbury, 2001.

[25] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Trans. Software Eng., vol. 34, no. 4, 2008, pp. 485-496.

[26] I.H. Witten, and E. Frank, Data mining: Practical machine learning tools and techniques, 2nd ed., Morgan Kaufmann Publishers, 2005.

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

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

[29] G. Antoniol, K. Ayari, M.D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? A text-based approach to classify change requests," Proc. the 2008 conference of the center for advanced studies on collaborative research, Ontario, Canada, Oct. 27-30 2008.

[30] M.F. Porter, "An algorithm for suffix stripping," Program, vol. 16, no. 3, 1980, pp. 130-137.

[31] S. Kim, and M.D. Ernst, "Which warnings should i fix first?," Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Sep. 3-7 2007, pp. 45 - 54.

[32] S. Heckman, and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," Proc. 2nd International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, Oct. 9-10 2008, pp. 41-50.