ABSTRACT

SHIN, YONGHEE. Investigating Complexity Metrics as Indicators of Software Vulnerability. (Under the direction of Laurie Williams.)

A single exploited software vulnerability can cause severe damage to an organization legally and financially. Early detection of software vulnerabilities can prevent the damage caused by late detection. Security experts claim that complexity is the enemy of security. A complex software system is difficult to understand, maintain, and test by software engineers resulting in errors in code including vulnerabilities. As a result, finding metrics that can measure software complexity and can point toward the code locations that are likely to have vulnerabilities early in the development life cycle is beneficial.

The goal of this research is to investigate complexity metrics that can indicate vulnerable code locations to improve the efficiency of security inspection and testing. For this purpose, this research conducts empirical evaluation of four types of complexity metrics: code complexity; OO design complexity; dependency network complexity; and execution complexity metrics as indicators of vulnerability. The evaluation is performed on four widely used open source projects by testing whether complexity metrics can discriminate vulnerable and neutral code locations and whether the prediction models built using those complexity metrics can predict vulnerable code locations. While complexity metrics have long been used for fault prediction, faults have different distributions from vulnerabilities. Therefore, this research additionally compares the ability of traditional fault prediction models and vulnerability prediction models to see whether traditional fault prediction models can also effectively predict vulnerabilities. Finally, software metrics that quantify code change history

and developer collaboration history have been effective for fault prediction. Therefore, this research compares the ability of complexity metrics and other types of metrics obtained from development history as indicators of vulnerabilities.

This research improves our understanding on the relationship between software complexity and vulnerability, contributing to the body of empirical knowledge as follows:

- This research provides empirical evidence that **complexity metrics can indicate vulnerable code locations**.
- This research provides empirical evidence that **vulnerable code is more complex, has large and frequent changes, and has more past faults than faulty code**.
- This research provides empirical evidence that **fault prediction models that are trained to predict faults can predict vulnerabilities at the similar prediction performance to the vulnerability prediction models that are trained to predict vulnerabilities** despite the difference in the distribution of faults and vulnerabilities.
- This research provides empirical evidence that **code execution frequency and duration based on software usage patterns by a normal user can indicate vulnerable code locations.**
- This research provides empirical evidence that **process metrics are better indicators of vulnerabilities than complexity metrics** when process metrics are available.
- This research **defines and uses simple and useful measures of code inspection cost and code inspection reduction efficiency obtained from a prediction model**.
- This research demonstrates that **automated text classification is feasible and useful to classify bug reports for faults and enhancements**.

- This research reveals that **a careful analysis of the relationship between faults/vulnerabilities and software metrics is required because the analysis results largely depend on the distribution of faults/vulnerabilities and the distribution of faults/vulnerabilities is specific to each project.**

Investigating Complexity Metrics as Indicators of Software Vulnerability

by
Yonghee Shin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

_____	_____
Dr. Dennis R. Bahler	Dr. Jason A. Osborne
_____	_____
Dr. Mladen Vouk	Dr. Laurie Williams
Co-Chair of Advisory Committee	Chair of Advisory Committee

DEDICATION

To my father who rests in heaven now and who gave me great confidence by always being proud of me and to my mother who has encouraged and supported me in prayers during this long journey.

BIOGRAPHY

Yonghee Shin was born in Kyungnam province in Korea. She obtained a Bachelor of Science degree in Computer Science from Sookmyung Women's University in Seoul, Korea in 1992. She worked as a software engineer in Daewoo Telecommunications, Ltd. and Samsung SDS, Ltd, in Seoul, Korea for eight years. She returned to academia to earn a Master of Science degree in Computer Science at Texas A&M University. After finishing her master's program in 2003, she began her Ph.D. study at North Carolina State University in 2004. Between every major step of her career paths and during all this progress, she had some time seemingly wasted struggling to find her best way between the reality and the ideal career paths for her, but the final destination always happens to be where she could satisfy her desire to improve knowledge and to enjoy some intellectual delight.

ACKNOWLEDGMENTS

I have felt my mother's prayer far from where she was during this whole time of study in U.S.A. She consistently gave me great comfort, encouragement, and advice in every difficult situation. I thank my brother and sister-in law for their cheer for me and for caring for my mother when I could not be with her for this long time. I thank my twin nieces, Yeji and Yeun, because they were such a source of joy for me during my study. I will never forget the time we have traveled together and I wish we do that again. Sangman Lee, my uncle, was a motivator of my study as a role model who was always positive and willing to take challenges. I think my generosity toward my nieces was inherited from him who supported me from time to time at the right moment during my study.

Laurie Williams, my advisor, was the first woman boss that I worked for and the best boss that I ever worked for. She was my role model academically and in life as a woman with her positive thinking, relentless energy to pursue higher goals, and excellent time management. I cannot express my gratefulness enough with words for her great advice, encouragement, funding, all the opportunities she provided me with, and most of all, her patience for my slow progress mostly because of my weak health.

Mladen Vouk, Dennis Bahler, and Jason Osborne, members of my committee, provided insightful comments for my Ph.D. proposal and guided me to pursue my research with more

scientific rigor. I especially thank Jason Osborne who generously spent time with me to explain statistical matters and gave me great help via co-authoring a paper.

Aldo Dagnino, my mentor at ABB, during my first summer internship, gave me an opportunity that I could get more sense on research when I did not fully understand the essence of research yet.

Thomas Ostrand, Elaine Weyuker, Robert Bell, and Mary Fernandez, my mentors and supervisor at AT&T Labs during my second summer internship, gave me unforgettable research experience. Not long after I changed my research topic from software testing to software metrics, these established researchers showed me how to think logically in terms of data analysis. I greatly enjoyed the collaboration with them via brainstorming and paper co-authoring. I was really lucky and am proud of having been a part of such a gentle, smart, and pleasant collaboration team.

Graham Holmes, Rich Livingstone, John Qian, especially Erick Lee, my team members and mentor at Cisco during my third summer internship, gave me a great opportunity to learn the software security practice in an industrial company and also helped me improve knowledge in security area.

The members of our Realsearch group, Nachi, Hema, Jiang, Mark, Dright, Lucas, Sarah, Michael, Stephen, Andy, Lauren, Ben, Raza, Andrew, John, Jerrod, Patrick, and Will were friends, collaborators, teachers, competitors, and also motivators when I was in a slump. As well said in the Bible, "As iron sharpens iron, so one man sharpens another," numerous

discussions and suggestions during our reading group sharpened the way I think for research. Especially, I appreciate Andy Meenely, who was a great collaborator while we were writing our journal paper. Thank you, Ben. Because we cleaned our office by your suggestion, my allergy has stopped and I could study in our office and my performance improved. I did not know that the dust in the office was the main cause of my allergy.

I thank the reviewers and editors of my IEEE Transactions on Software Engineering journal paper. Over 7000 of words of critiques were initially daunting to me. But ironically their passions to improve the quality of research by helping the authors gave me encouragement and helped me improve my research and writing skills by following their advices resulting in a publication.

Most of all I thank God who always leads my way wherever I go and whatever I do and who made this moment that I am writing this acknowledges the best part of my dissertation writing.

This work is supported in part by the National Science Foundation Grant No. 0716176 and the CAREER Grant No. 0346903. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

AUC – Area under the ROC curve.

discriminative power – the ability to discriminate between high-quality software components and low-quality software components [34].

entity – a unit of code analysis. An entity can be a file, a class, or a component depending on studies.

error – "human action that results in software containing a fault" [33].

failure – "the termination of the ability of a functional unit to perform its required function" [33]. "A failure may be produced when a fault is encountered" [33] during code execution.

failure proneness – the probability that an entity contains one or more failures.

failure-prone entity – an entity that is likely to fail due to the execution of code that has faults.

fault – **"**1) an accidental condition that causes a functional unit to fail to perform its required function. 2) a manifestation of an error in software" [33].

faulty entity – an entity that has at least one reported fault.

fault prediction model – 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.

fault-proneness– the probability that an entity contains one or more faults [17].

fault-prone entity – an entity that is likely to have one or more faults.

file inspection ratio – the ratio of entities predicted as vulnerable (that is, the number of entities to be inspected) to the total number of entities.

file inspection reduction – the ratio of the reduced number of entities to be inspected by using a prediction model compared to a random classification.

LOC inspection ratio – the ratio of lines of code to be inspected to the total lines of code.

LOC inspection reduction – the ratio of reduced lines of code to be inspected by using a prediction model compared to a random classification.

module – a smallest unit of functionality such as a function or a method [51].

neutral entity – an entity that has no discovered vulnerabilities or faults. In Chapter 5 where both faults and vulnerabilities are investigated, a neutral entity means an entity that has neither a discovered fault nor a discovered vulnerability. In other chapters that discusses only vulnerabilities, a neutral entity has the same meaning as *non-vulnerable entity*.

non-faulty entity – an entity that has no discovered faults.

non-vulnerable entity – an entity that has no discovered vulnerabilities.

operational profile – "a quantitative characterization of how a system will be used" [54].

ROC curve – Receiver Operating Characteristic curve. A graph of recall versus probability of false alarm at various classification thresholds.

precision – the ratio of correctly predicted vulnerable entities to all predicted vulnerable entities.

predictability – the ability to predict a quality factor value with the required accuracy [34].

process metrics – metrics that are collected from software-related activities such as time, effort, number of requirement changes [21].

product metrics – metrics that are collected from software artifacts, deliverables or documents [21].

recall – the ratio of correctly predicted vulnerable entities to actual vulnerable entities.

vulnerable entity – an entity that has at least one reported vulnerability.

vulnerability – an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [43]. 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.

vulnerable entity – an entity that is likely to have one or more vulnerabilities.

vulnerability prediction model – 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.

CHAPTER 1

Introduction

A single exploited software vulnerability can cause severe damage to an organization. Annual worldwide losses caused from cyber attacks have been reported to be as high as $226 billion [11]. Loss in stock market value in the days after an attack is estimated at $50 million to $200 million per organization [11]. The importance of detecting and mitigating software vulnerabilities early in the software development life cycle is paramount to prevent such damage and to reduce costs that are caused by late fixes [6]. With limited time and budget in development teams, efficient allocation of inspection and testing effort is also critical. Hence, finding software characteristics that can indicate the code locations that are likely to have vulnerabilities and to prioritize inspection and testing effort with the information that can be obtained early in the development life cycle is beneficial.

Software complexity is often hypothesized to be the enemy of software security [25, 35, 46]. The wisdom of security experts is that complexity leads to security problems [25, 35, 46, 67]. Schneier [67] claimed that “complexity is the worst enemy of security” among the features of technology that can increase security vulnerabilities such as automation (of attacks) and remote connectivity. Geer [25] stated that “complexity provides both opportunity and hiding places for attackers” when he addressed the importance of

cybersecurity in a hearing at the Subcommittee of Homeland Security on Emerging Threats, Cybersecurity, and Science and Technology on 23rd, April, 2007. McGraw [46] also points out complexity as one of three major causes of software security problems (the "Trinity of Trouble"); the other causes together are connectivity and extensibility. The wisdom of these experts, though, has not been substantiated by empirical evidence using quantifiable metrics in terms of software security in depth.

The experts' claims are intuitive because complex software systems are difficult to understand, test, and maintain [15, 45, 76]. Therefore, the more complex a software system is, the more chance that software engineers cannot fathom the security problems in the system and the more chance for software engineers to ignore security concerns. Hence, finding complexity metrics that can indicate software complexity and examining whether those complexity metrics can indicate vulnerable code locations may help organizations to find software vulnerabilities before software release and to efficiently allocate security inspection and testing effort. *The goal of this research is to investigate complexity metrics that can indicate vulnerable code locations to improve the efficiency of security inspection and testing.*

1.1 Research Questions and Hypotheses

Extensive fault prediction studies including [5, 10, 36, 38, 51, 53, 57, 58, 70, 81] have used complexity metrics that can be obtained statically from software artifacts to predict

fault-prone code locations in a software system based on the hypothesis that software engineers can produce more errors in complex code than in simple code. While the same hypothesis applies to vulnerabilities, whether those static complexity metrics are also effective to find vulnerable code locations has not been investigated in depth. From this lack of evidence, our first research question arises:

Research Question 1. *Can static complexity metrics that reflect difficulty in program understanding and maintenance indicate vulnerable code locations?*

To answer this question, this research examines whether static complexity metrics can discriminate *vulnerable code* locations and *neutral code* locations and whether static complexity metrics can predict vulnerable code locations using three types of static complexity metrics; code complexity; Object Oriented (OO) design complexity; and dependency network complexity metrics. Here *neutral code* means the code whose vulnerabilities have not been discovered yet. The associated hypothesis is:

Hypothesis 1. Because complex code is difficult to understand, test, and maintain, developers tend to introduce more security errors to complexity code than to simple code. Therefore, complexity metrics statically obtained from software artifacts can discriminate vulnerable code locations and neutral code locations and can predict vulnerable code locations.

Initial testing results on Hypothesis 1 using code complexity metrics are provided in Chapter 4 together with a comparison of discriminative power and predictive power with two process metrics. The two process metrics are code churn metrics that quantify code change history [29, 41, 55, 56] and developer activity metrics that quantify developer collaboration history [48]. These process metrics also have been effective for fault prediction. Chapter 6 tests Hypothesis 1 with three static complexity metrics including code complexity, OO design complexity, and dependency network complexity metrics and a dynamic complexity metric that will be discussed in Hypothesis 3.

Even when the complexity metrics are useful to predict vulnerable code locations, whether an organization needs a separate vulnerability prediction model when they already have a fault prediction model is questionable. Although complex code may have more faults and vulnerabilities than simple code, faults and vulnerabilities are different in two ways. First, vulnerabilities are discovered with much lower frequency than faults [2]. Hence, finding vulnerabilities is akin to finding a "needle in a haystack", which may weaken the ability of complexity metrics as indicators of vulnerabilities. Second, attackers actively seek vulnerabilities with malicious or criminal intent, while non-vulnerable faults are exposed based upon the normal use of software. This difference necessitates software engineers who can think like an attacker and have special security skills in conducting vulnerability

detection. Therefore, the characteristics of *discovered* vulnerabilities and faults may be different. Therefore, our second research question is:

Research Question 2. *Can fault prediction models be used to predict vulnerabilities with equal to or better prediction performance than vulnerability prediction models?*

To answer this question, this research builds a fault prediction model that is trained to predict faults and a vulnerability prediction model that is trained to predict vulnerabilities using traditional fault prediction metrics including code complexity, code change history, and past fault history metrics. Then, the vulnerable files included in the predicted files by both models are compared. The associated hypothesis is:

Hypothesis 2. Despite the similar characteristics between faulty code and vulnerable code, different intentions and usage patterns between normal users and attackers/security researchers may result in different distribution of software metric values between faulty code and vulnerable code. Therefore, vulnerability prediction models trained to predict vulnerabilities can provide better performance than fault prediction models trained to predict faults in the ability to predict vulnerable code locations.

Hypothesis 2 is tested in Chapter 5 by comparing the predictability between fault prediction models and vulnerable prediction models.

While the static complexity metrics that are intended to capture the difficulty in understanding, testing, and maintaining code from the software engineer's perspective [15,

45, 76] seem to be promising, some complex code may be rarely executed by attackers. Instead, metrics that are collected during software execution to reflect usage patterns of software by attackers may be more effective to identify vulnerable code locations. However, considering the difficulty in obtaining usage patterns of software by attackers, collecting execution metrics that reflect usage patterns of software by normal users first and examining the relationships between execution metrics and vulnerabilities can still provide benefits; if the execution metrics obtained from the normal user's usage patterns can indicate vulnerable code locations, organizations need to focus their security inspection and testing effort to the most frequently used code areas; otherwise, the software usage patterns by attackers may be different from the usage patterns by normal users and operational profile based on attacker's behavior should be sought. Hence the third question in this research is:

Research Question 3. *Can the software execution metrics obtained from usage patterns by normal users indicate vulnerable code locations?*

As a way to measure common usage patterns by normal users, runtime complexity metrics that measure the frequency and the duration of code execution can be used. Since this research question is not substantiated well by previous observations, this research pursues to answer this question without setting up a hypothesis. The modified research question is:

Research Question 3'. *Can the execution complexity metrics obtained from usage patterns by normal users indicate vulnerable code locations?*

Question 3' is answered in Chapter 6 by examining discriminative power and predictability of execution complexity metrics.

1.2 Approach

This research utilizes empirical case studies using four open source projects; the Mozilla Firefox web browser[1]; the Red Hat Enterprise Linux kernel[2]; the Wireshark network protocol analysis tool[3]; and the Apache Tomcat servlet container[4] to answer the proposed research questions. The metrics and vulnerability information are collected automatically using commercial code analysis tools detailed in Section 2.2 and the tools that are developed for this research. For the hypothesis testing, discriminative power is tested by comparing the mean difference in metric values from vulnerable code and neutral code at 0.05 significance level. Predictability is tested on the prediction results from a logistic regression model. The prediction results are validated using *n*-fold stratified cross-validation. The significance of predictability is tested against the predictability of a random classification model. Additionally, the significance of predictability is tested against performance benchmarks found from fault prediction literature.

---

[1] http://www.mozilla.com/firefox/
[2] http://www.redhat.com/rhel/
[3] http://www.wireshark.org/
[4] http://tomcat.apache.org/

1.3 Contributions

The contributions of this dissertation work are as follows[5]:

- This research provides empirical evidence that complexity metrics can indicate vulnerable code locations.
- This research provides empirical evidence that vulnerable code is more complex, has large and frequent changes, and has more past faults than faulty code.
- This research provides empirical evidence that fault prediction models that are trained to predict faults can predict vulnerabilities at the similar prediction performance to the vulnerability prediction models that are trained to predict vulnerabilities despite the difference in the distribution of faults and vulnerabilities.
- This research provides empirical evidence that code execution frequency and duration based on a normal user's software usage patterns can be used to identify vulnerable code locations.
- This research provides empirical evidence that process metrics are better indicators of vulnerabilities than complexity metrics when process metrics are available.
- This research defines and uses simple and useful measures of code inspection cost and code inspection reduction efficiency obtained from a prediction model.

[5] Andy Meenely collaborated for the work in Chapter 4 by suggesting and collecting developer activity metrics.

- This research demonstrated that automated text classification is feasible and useful to classify bug reports for faults and functional enhancements.
- This research reveals that a careful analysis of the relationship between faults/vulnerabilities and software metrics is required because the analysis results largely depend on the distribution of faults/vulnerabilities and the distribution of faults/vulnerabilities is specific to each project.

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the complexity metrics that are used for this research. Chapter 3 presents related work on the use of complexity metrics as indicators of faults and vulnerabilities. Chapter 4 explains the complexity metrics this research utilizes and details the criteria for hypothesis testing. Chapter 4 also explains the modeling techniques and validation methods for prediction of vulnerable code locations. Chapter 5 analyzes the ability of code complexity metrics as indicators vulnerable code locations to answer Research Question 1 and to test Hypothesis 1. Two process metrics are also compared with code complexity metrics. Chapter 6 examines whether fault prediction models can be used for vulnerability prediction with the same or better predictability as vulnerability prediction models to answer Research Question 2 and to test Hypothesis 2. Chapter 7 compares execution complexity metrics and static

complexity metrics to answer Research Quesiton3'. Chapter 8 summarizes this research and provides directions for future research.

CHAPTER 2

Background

This chapter provides the concept of complexity and discusses the complexity metrics that this research is interested in, the reason why those metrics matter in terms of software security and the method of complexity metrics collection.

2.1 Software Complexity

Complex is defined as "1. composed of many interconnected parts; compound; composite, 2. characterized by a very complicated or involved arrangement of parts, units, etc. 3. so complicated or intricate as to be hard to understand or deal with" [13]. This definition tells that complexity arises from the amount of entities and the degree of interaction between entities and those complexity results in difficulty in understanding when the amount of entities and the degree of interaction reach a certain level.

In the view of software, an entity can be a bit or an executable file at binary level. An entity can be a variable, a code line, a function, a file, or a component at code level. An entity can be a word, a diagram, or a document at design level. The first definition of complexity tells that an entity with many sub-entities is more complex than an entity with a few sub-entities. For example, a file with many lines of code is more complex than a file with a few

lines of code; a function with many parameters is more complex than a function without a parameter; a function with many decision statements is more complex than a function with a few decision statements. The second definition of complexity tells us that an entity with high interaction with other entities is more complex than an entity with low interaction. Software entities can have a dependency relationship via data flow, control flow, function calls, or class inheritance. The third definition of complexity tells us that these compositions of entities and interactions between entities result in complex software and make software engineers difficult to understand code.

The above discussion emphasizes the structural complexity that affects the software engineer's cognitive perception [15, 45, 76] resulting in errors in software during development or testing. However, some complex code may be rarely executed by attackers and security researchers. Therefore, information at runtime of software based on the usage patterns by attackers and security researchers may better reveal the code locations that are likely to have vulnerabilities than the complexity metrics statically collected. In this research, the frequency and the duration of execution measured at runtime is called *execution complexity*. Note that time complexity [14, 18] in computational complexity theory also measures the time to execute an algorithm, but time complexity measures the theoretical boundary of time that can be taken by any input for a given algorithm. Similar to time

complexity, execution complexity depends on input, but is measured on specific code with specific input.

2.2 Four Complexity Metric Types

This research investigates the following four types of complexity metrics as potential indicators of vulnerabilities:

- *Code complexity metrics* measure complexity of an entity at code level.
- *OO design complexity metrics* measure complexity that can be obtained at OO design level.
- *Dependency network complexity metrics* measure the complexity of interaction between entities from a dependency network constructed from data dependency and call dependency between entities.
- *Execution complexity metrics* measure complexity at runtime.

In this research, an entity can be a file, a class, or a component depending on studies in the following chapters.

This section provides a high level view of these complexity metrics by categorizing the metrics in multiple dimensions. Subsections 2.2.1 through 2.2.4 define individual metrics collected for this study and provide informal hypotheses of using these complexity metrics as indicators of vulnerabilities.

Categorizing complexity types depending on the dimensions of metrics can help us conceptualize similarities and differences between different complexity metric types and also help us find other possible metrics. This research categorizes the complexity metrics used in this research in four dimensions depending on the stages in the software development lifecycle that metrics can be obtained, data collection method, relationships between entities, and programming language dependency. Investigating metrics from as many categories as possible is beneficial because metrics from the same category can provide similar information each other [40]. Instead, complexity metrics from diverse categories can give more complete understanding on the relationship between complexity and vulnerabilities than complexity metrics that provide only a narrow view of the total complexity.

The first dimension we use to categorize complexity metrics is the stage in the software development lifecycle that metrics can be obtained. Metrics that are collected for this research can be obtained at coding, design, or integration testing time. Code metrics can be obtained during coding. Although OO design complexity and dependency network complexity metrics are obtained from code in practice in most cases, those metrics can be obtained during design time in theory and may help earlier detection of vulnerabilities than coding time metrics. Execution complexity metrics we are collecting in this research should reflect the usage patterns of users. To reflect the realistic scenario of software usage, the earliest time that execution complexity metrics can be obtained is during integration testing.

The second dimension of metrics classification is the data collection methods. Static metrics can be obtained from software artifacts such as code or design documents without code execution, while dynamic metrics are collected during code execution. Static metrics measure the complexity in the structure of entities such as the number of sub-entities in an entity and the interaction between entities without code execution. The benefit of using static metrics is that static metrics do not require execution environments for a software system including hardware, third-party libraries, and input data specification. As mentioned earlier, however, static metrics do not show how those statically complex or simple entities behave at runtime. On the other hand, dynamic metrics can provide more precise information on the behavior of a software system, but the information can be collected only from the execution paths that have been actually executed. Therefore, depending on the operational profile that has been used for the execution, the values of execution metrics collected can change.

The third dimension of metrics classification is the entity relationships. While some metrics represent the complexity within an entity (intra-entity metrics), some metrics represent the complexity in the relationship with other entities (inter-entity metrics). Code complexity, OO design complexity, and execution complexity metrics include both inter- and intra-entity metrics. However, most code complexity metrics are intra-entity metrics while most OO design metrics are inter-entity metrics. All dependency network complexity metrics are inter-entity metrics.

Finally, metrics can be classified by the dependency on programming languages. While code, network, and execution complexity metrics can be obtained from the software projects that use any programming languages, OO design metrics can be obtained only from the software projects that use OO languages such as C++ or Java.

Table 1 shows the categories of complexity metrics according to the four dimensions. As seen in Table 1, each complexity type represents different combination of categories from the four dimensions.

Table 1. Classification of complexity metrics

Complexity Type	**Stage in Development Life Cycle**	**Data Collection Method**	**Entity Relationship**	**Language-Dependency**
Code complexity	Coding	Static	Intra-entity (+Inter-entity)	None
OO design complexity	Design	Static	Inter-entity (+ Intra-entity)	OO languages
Dependency network complexity	Design	Static	Inter-entity	None
Execution complexity	Integration Testing	Dynamic	Intra-entity / Inter-entity	None

Each individual metric is explained in the following subsections in detail.

2.2.1. Code complexity metrics

Table 2 provides the definitions of code complexity metrics. Complex code is difficult to understand, maintain, and test [15, 45, 76]. Therefore, complex code would have a higher chance of having faults undetected than simple code. Since attackers exploit the faults in a program, complex code would be more vulnerable than simple code. Code with many

lines, many variables, many function definitions, and many preprocessing declaratives is more difficult to understand, maintain, and test than simple code. In Table 2, LOC, LOCVarDecl, NumFunctions, and NumLinePreprocessor measure these aspects, respectively.

EssentialComplexity, CyclomaticStrictComplexity, and MaxNesting measure complexity from control flow. Structured code is easier to understand and maintain than unstructured code [45]. EssentialComplexity measures the structuredness of a function. Code with many branches requires more test cases than code with a few branches. CyclomaticStrictComplexity measures the upper bound of required test cases by counting the number of conditional branches in a function. Code with deep levels of control blocks with `if` or `while` statements is more complex than the code with shallow levels of control blocks. MaxNesting measures the depth of control blocks.

Highly coupled code has a higher chance of having input from external sources that are difficult to trace where the input came from. In highly coupled code, a change in an entity can incur changes in many other entities and likely to introduce errors. Moreover, developers can use interfaces to modules implemented by other developers or by a third party without properly understanding of the security concerns and assumptions of the modules. Therefore, highly coupled code may be more vulnerable than less coupled code. In Table 2, FanIn and FanOut measure coupling at the function level.

Communication by comments between developers is important especially in open source projects for which many developers can contribute on the same code segment without central control. Novice developers who do not understand security concerns and do not follow secure coding practice might comment less often. Furthermore, code developed in a

Table 2. Definitions of code complexity metrics

Metric Name	Metric Definition
LOC	The number of lines of code in an entity.
LOCVarDecl	The number of lines of code with variable declarations.
NumFunctions	The number of functions defined in an entity.
NumLineProcessor	The number of lines of code in a file devoted to preprocessing.
EssentialComplexity	The number of branches in a function after reducing all the programming primitives such as a for loop in a function's control flow graph into a node iteratively until the graph cannot be reduced any further. Completely well-structured code has essential complexity 1 [45, 75]. Sum and maximum values of EssentialComplexity in each entity were used in this research.
CyclomaticStrictComplexity	The number of conditional statements in a function. Sum and maximum values of CyclomaticStrictComplexity in each entity were used in this research.
MaxNesting	The maximum nesting level of control constructs such as if or while statements in a function. Sum and maximum values of MaxNesting in each entity were used in this research.
CommentDensity	The ratio of lines of comments to lines of code in an entity.
FanIn	The number of inputs that a function uses such as parameters and global variables. Sum and maximum values of FanIn in each entity were used in this research.
FanOut	The number of outputs that are set in a function to parameters or global variables. Sum and maximum values of FanOut in each entity were used in this research.

hurry (perhaps directly prior to release) might have fewer comments and be more vulnerable. Therefore code with a few comments may be more vulnerable than code with many comments. In Table 2, CommentDensity measures this aspect.

Code complexity metrics were collected using a commercial code analysis tool, Understand C++[6] .

2.2.2. OO design complexity metrics

This research uses the six OO design complexity metrics proposed by Chidamber and Kemerer [12]. Table 3 provides the definitions of OO design complexity metrics. For the same reason mentioned in Section 2.2.1, highly coupled code may have more chance to introduce vulnerabilities than less coupled code because of difficulty in tracing input sources, change propagation, and possible misunderstanding of security concerns in a third party component. In Table 3, Coupling Between Objects (CBO) measures the degree of coupling between classes.

Number Of Children (NOC) and the Depth of Inheritance Tree (DIT) measure the complexity incurred from the hierarchical design of classes. The Weighted Methods per Class (WMC) and the Lack of Cohesion of Methods (LCOM) measure the intra-entity complexity within a class. Lacking cohesiveness indicates the irrelevant code co-exists in the same class. Lack of cohesion can make code difficult to understand and increases the

[6] http://www.scitools.com/

likelihood of errors during development. Excessively high values of NOC, DIT, WMC, and LCOM indicate the necessity of refactoring to make code easier to understand, maintain, and test. Response for a Class (RFC) counts the number of methods that can potentially be executed by an event received by an object of a class.

Even though OO design metrics were obtained from source code in many studies including ours, OO design metrics can be obtained just from OO design documents without actual implementation, which enables earlier estimation of vulnerabilities than code level metrics. Table 3 provides the definitions for the OO design complexity metrics.

OO design complexity metrics were collected using a commercial code analysis tool, Understand C++.

Table 3. Definitions of OO design complexity metrics

Metric Name	Metric Definition
CBO	(Coupling Between Objects) The number of other classes that use methods or instance variables in a class.
DIT	(Depth of Inheritance Tree) The depth of inheritance in the class inheritance tree.
NOC	(Number of Children) The number of immediate children nodes (subclasses) in the class inheritance tree.
LCOM	(Lack of Cohesion of Methods) The degree of lack of similarities between methods in a class measured by 100% minus the average of the percentage of the methods that access the same instance variable in a class.
WMC	(Weighted Methods per Class) The number of methods defined in a class excluding inherited methods.
RFC	(Response for a Class) The number of methods including inherited methods.

2.2.3. Dependency network complexity metrics

Dependency network complexity metrics quantitatively measure the complexity in the relationships between nodes in a network represented by a graph [8]. In this research, a network graph is constructed from software dependency including data dependency and call dependency. A data dependency between two code entities occurs when an entity defines a variable and the other entity uses it [58]. A call dependency between two code entities occurs when an entity defines a method and the other entity calls it [58]. A node is an entity. An edge connecting two nodes represents that there is a dependency between two entities. Among dependency network complexity metrics, centrality metrics measure the relative importance of a node in a graph. A highly central node in a software dependency network graph is a node that depends or is depended on by many other nodes. In terms of information flow, a central node sends or receives directly or indirectly more information to and from many other nodes than non-central nodes. Because vulnerability can be exploited from maliciously manipulated input, the code with higher information flow may be more vulnerable than the code with lower information flow.

Table 4 provides the definitions of dependency network metrics that this research uses. In Table 4, InDegree and InDegree_w measure how many and how frequently other entities depend on an entity directly. OutDegree and OutDegree_w measure how many and how frequently an entity depends on other entities directly. SP_Betweenness measures how many times an entity sits in the middle of information flow via the geodesic path (shortest paths) between pairs of nodes in the dependency network [22]. Although SP_Betweenness is the most well-known and frequently used metric as a mediator of information, information does not necessarily flow via a shortest path. Alternatively, *Flow_Betweenness* considers maximum information flows between all possible paths that pass a node [24]. Different from

Table 4. Definitions of dependency network complexity metrics

Metric Name	Metric Definition
InDegree	The number of unique incoming edges to a node in a network graph.
InDegree_w	The number of incoming edges to a node in a network graph.
OutDegree	The number of unique outgoing edges from a node in a network graph.
OutDegree_w	The number of outgoing edges from a node in a network graph.
SP_Betweenness	The number of times that a node sits on the shortest paths of other node pairs [23].
Flow_ Betweenness	The maximum information flow between all pairs of nodes in a weighted graph in which a node is in the path of the maximum information flow [24].
RW_ Betweenness	The number of random walks between all pairs of nodes in a graph in which a node is in the paths of the random walks [61].
EvCent	(Eigenvector centrality). The value computed from an eigenvalue of the adjacency matrix of a network graph in a way that a node connected to nodes with high eigenvector centralities has higher eigenvector centrality than a node connected to nodes with lower eigenvector centralities [7].

SP_Betweenness that considers only geodesic paths in a simple graph, Flow_Betweenness considers a weighted graph. In our research, the number of dependencies between two entities is the weight of an edge in the network graph. However, Flow_Betweenness also has a limitation because information does not necessary flow in a way that the flow is maximized in a graph. Alternatively, RW_Betweenness measures how often information passes a node via a random walk between pairs of nodes in the dependency network [61]. EvCent measures the importance of a node by assigning higher value to a node that is connected to other high valued nodes than a node connected to other low valued nodes [7].

Dependency network complexity metrics were collected using multiple tools including a commercial network analysis tool, UCINET[7], an open source network analysis library, Jung[8], and the igraph[9] package in R based on the dependency information obtained using Understand C++.

2.2.4. Execution complexity metrics

Execution complexity metrics measure the frequency of function calls and the duration of execution of functions. This research is interested in whether execution complexity metrics collected from common usage patterns can indicate vulnerable code locations. Specific use cases used in this research to represent common usage patterns are

---

[7] http://www.analytictech.com/ucinet/
[8] http://jung.sourceforge.net/
[9] http://igraph.sourceforge.net/

explained in Chapter 7 where execution complexity metrics were collected for two open source projects, Firefox and Wireshark. NumCalls measures the frequency of function calls by counting the number of function calls to the functions (or methods) in a file, a component, or a class. Both InclusiveExeTime and ExclusiveExeTime measure the duration of execution. While InclusiveExeTime measures the total execution time that is spent by a function and all the functions called by the function, ExclusiveExeTime measures the time spent within a function excluding the execution time spent by the called functions. Table 5 provides the definitions of execution complexity metrics.

Execution complexity metrics were collected from the execution profile obtained by an open source profiling tool, Callgrind, which is a component of an instrumentation framework for building dynamic analysis tools, Valgrind[10].

Table 5. Definitions of execution complexity metrics

Metric Name	Metric Definition
NumCalls	The number of calls to the functions defined in an entity.
InclusiveExeTime	Execution time for the set of functions, S, defined in an entity including all the execution time spent by the functions called directly or indirectly by the functions in S.
ExclusiveExeTime	Execution time for the set of functions, S, defined in an entity excluding the execution time spent by the functions called by the functions in S.

[10] http://valgrind.org/

CHAPTER 3

Related Work

Extensive studies, including [5, 10, 36, 38, 51, 53, 57, 58, 70, 81], have investigated the relationship between faults/failures and software metrics including complexity metrics. Some studies [26, 27, 59, 60, 68, 69, 71, 74, 82] have investigated the relationship between vulnerabilities and software metrics. This chapter provides the related work that utilized complexity metrics as predictors of software faults and failures. Compared to the vast amount of studies on the relationship between faults/failures and software metrics, vulnerabilities have been investigated by only a few groups of researchers. This chapter also discusses the studies that investigated the relationship between vulnerabilities and software metrics in general in Section 3.2.

3.1 Complexity Metrics as Indicators of Faults and Failures

Khoshgoftaar and Munson [38] built fault prediction models using eight code complexity metrics on small number of modules (20 and 8, respectively) from two software systems. Their regression models using code complexity metrics explained a large portion of variance showing the feasibility of building fault prediction models using code complexity metrics. However, their study did not provide prediction results. Nagappan et al. [57]

investigated the relationships between complexity metrics and software failures with larger sizes of Microsoft projects between 37 thousand lines of code (KLOC) and 511 KLOC. In their study, the complexity metrics include both code complexity metrics and OO design complexity metrics. Their multiple linear regression model using principle components of complexity metrics resulted in statistically significant positive correlations between predicted failures and actual failures, but the set of effective code complexity and OO design complexity metrics varied depending on projects. Menzies et al. [51] also built models to predict fault-prone modules using multiple machine learning techniques on eight projects from NASA MDP data repository [1]. Their Naïve Bayes model predicted 71% of fault-prone modules with 25% false positive rate on average. However, as Nagappan et al's finding [57], the set of effective code complexity metrics varied depending on projects. Additionally they found that a prediction model with only three metrics provided as good prediction performance as a prediction model with 38 metrics.

Basili et al. [5] evaluated the six OO design complexity metrics proposed by Chidamber and Kemerer [12] as predictors of fault-proneness. In their study performed on eight projects developed by students, five of the investigated OO design complexity metrics were statistically significant in their prediction model. Their study also showed OO design complexity metrics are better predictors than the best set of code metrics that they used including the number of function calls and the number of function definitions. Subramanyam

and Krishnan [72] confirmed that a subset of OO design complexity metrics are effective for fault prediction even after controlling for size of code, but the effects of the metrics varied depending on programming languages. Briand et al. [10] also confirmed that OO design complexity metrics are effective for fault prediction with a large set of OO design complexity metrics including the variants of Chidamber and Kemerer's six OO design complexity metrics. Zhou and Leung performed a similar study with OO design metrics, but for faults with different severity levels [80]. Their study also showed that OO design complexity metrics are effective indicators of faults, but the effective set of metrics varied depending on the severity of faults.

More recently software dependency network complexity metrics have been studied as indicators of failures. Nagappan and Ball [58] showed that software dependencies are good indicators together with code churn metrics. However, this study does not provide the effectiveness of software dependency separately. Zimmermann and Nagappan [81] used dependency network complexity metrics to predict faults on Windows Server 2003. The fault prediction model with dependency network complexity metrics predicted failure-prone entities 10% better than the prediction model with code complexity and OO design complexity metrics.

Execution complexity metrics have rarely been used in fault prediction studies. Khoshgoftaar and Allen [40] measured software execution time of four releases of a large

telecommunications system together with code complexity and process metrics. Their classification tree model predicted fault-prone modules with less than 30% fault positive rate. In all the three types of metrics, there were subsets of metrics that were chosen by the classification model.

All these positive relationships between faults and complexity metrics suggest the possibility of using complexity metrics as indicators of vulnerabilities if vulnerabilities and faults have similar characteristics.

3.2 Software Metrics as Indicators of Vulnerabilities

Neuhaus et al. predicted vulnerabilities on the entire Mozilla open source project (not specific to Firefox) by analyzing the header file inclusion and function calls between components [59], where a component in their study is defined as a C/C++ file and its header file of the same name. They analyzed similar patterns of frequently used header files and function calls in vulnerable components and used the occurrence of the patterns as predictors of vulnerabilities. Their model using import and function call metrics correctly predicted 45% of the total vulnerable components and among the predicted vulnerable components, 70% were correctly predicted. Neuhaus and Zimmermann [60] also investigated the relationships between vulnerabilities and package dependency of the Red Hat Linux distribution and found certain dependencies are more related with vulnerabilities than other dependencies. Their support vector machine model using the dependency metrics correctly

predicted 65% of the total vulnerable packages and among the predicted vulnerable packages, 83% were correctly predicted.

Gegick et al. modeled vulnerabilities using the regression tree model technique with source lines of code, alert density from a static analysis tool, and code churn information [26]. They performed a case study on 25 components in a commercial telecommunications software system with 1.2 million lines of code. Their model identified 100% of the vulnerable components with an 8% false positive rate at best.

Walden et al. [74] measured the correlations between vulnerabilities and three complexity metrics together with other process metrics from 14 open source web applications written in PHP. The correlations were very different depending on the projects. For example, average cyclomatic complexity was between -0.96 and 0.88. Although Nagappan et al. [57] also found that there is no single metric that is correlated with failures across projects, the variance of correlations in Walen et al.'s study is much larger than the variance of correlations from Nagappan et al.'s study and also from the results from this research. This fact indicates that complexity in the programs written in script languages such as PHP may differ from the complexity in the programs written in programming languages such as C/C++ and Java.

Meneely and Williams [49] investigated the relationships between vulnerabilities and the structure of developer collaboration. Their empirical study with the Red Hat Enterprise

Linux kernel showed that files changed by nine or more developers are more likely to have a vulnerability than files changed by fewer developers.

Zimmermann et al. [82] investigated the relationships between vulnerabilities and a set of metrics including code churn, dependency, code complexity, and organizational structure on Windows Vista. Compared to their failure prediction study using Windows Server 2003 [58, 81], the correlations between vulnerabilities and the metrics were very low only up to 0.3 because of the rarity of vulnerabilities. The models with each type of metrics and with the combined set of the metrics correctly predicted around 40% to 70% (median of 100 runs random splits for three fold cross-validation) of the total vulnerable binaries (such as Windows DLL files) depending on the types of metrics. Among the predicted binaries, around 10% to 40% were correctly predicted. However, the variance of each run of cross-validation was very large indicating the prediction is not reliable enough.

The limited availability of projects that can obtain discovered vulnerability data and the rarity of discovered vulnerabilities hamper the research on the relationships between vulnerabilities and software complexity and make it difficult to generalize findings from research. This research contributes to the empirical body of knowledge by collecting vulnerability information from four large scale open source projects and by investigating the relationships between vulnerabilities and software complexity in depth.

CHAPTER 4

Evaluation Criteria, Prediction Modeling, and Validation Method

This chapter provides the evaluation criteria for the hypotheses provided in Chapter 1 and explains the validation methods of prediction results.

4.1 Hypothesis Test for Discriminative Power

This research investigates the discriminative power of complexity metrics in the context of software security. Discriminative power [34] is defined as the ability to "discriminate between high-quality software components and low-quality software components". In this research, discriminative power is the ability to discriminate the vulnerable entities from neutral entities.

To evaluate the discriminative power of the metrics, this research tests the null hypothesis that the means of metric values for vulnerable entities and neutral entities are equal. While a usual way to test the difference in mean is Student's t-test, Welch's t-test is more appropriate when data have unequal variance [22]. A preliminary analysis of data showed that the data used in this research had unequal variance. Therefore, this research uses Welch's t-test to evaluate discriminative power. In this research, a hypothesis for discriminative power is considered to be supported when the result from the Welch's test is

statistically significant at the $p<0.05$ level. When multiple hypotheses are tested simultaneously, the probability of falsely rejecting at least one of the null hypothesis increases [64]. To deal with this problem, Bonferroni correction [64] is performed . That is, the number of the hypotheses that have been tested is multiplied to the p-value of a Welch's t-test before it is compared with 0.05. The t statistic for Welch's t-test is defined in Equation (1).

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}} \tag{1}$$

where $\overline{X_i}$, s_i^2, and N_i are the sample mean, the sample variance, and the sample size of the two comparing groups i=1 and 2.

4.2 Predictability and Binary Classification Criteria

For this research, predictability [34] is defined as the ability of a metric to identify vulnerable entities before software release. Note that discriminative power is measured for a single metric mainly for the purpose of controlling the metric values by redesign or re-implementation or to investigate the feasibility of building prediction models using the given metrics. On the other hand, predictability can be measured using a model built with multiple

metrics together to predict the vulnerable entities in the future and mainly to allocate inspection and testing effort appropriately.

To predict code locations with vulnerabilities, this research performs binary classification using logistic regression. In binary classification, an entity is classified as vulnerable when the estimated probability of an entity having a vulnerability is over a certain classification threshold. Otherwise, an entity is classified as neutral. The default classification threshold is 0.5 in this research unless noted otherwise. In the projects used in this research, over 94% of files with vulnerabilities have less than three vulnerabilities across the four projects. Therefore, rather than ranking the files based on the number of predicted vulnerabilities, binary classification based on the predicted probability is more appropriate.

Figure 1 shows the possible four outcomes from binary classification. A binary classifier can make two possible errors: false positives (FP) and false negatives (FN). In this research, a FP is the classification of a neutral entity as a vulnerable entity, and a FN is the classification of a vulnerable entity as neutral. False positives represent excessive entities to inspect or test, and false negatives increase the chances of vulnerabilities escaping to the field without inspection or testing. A correctly classified vulnerable entity is a true positive (TP), and a correctly classified neutral entity is a true negative (TN).

	Predicted vuln.	Predicted non-vuln.
Actual vuln.	TP	FN
Actual non-vuln.	FP	TN

Figure 1. Binary classification results

Among the many classification performance measures that can be derived from these four classification results, *recall*, *precision*, and *probability of false alarm* are most widely used to measure the performance of fault prediction or vulnerability prediction.

Recall, also known as *probability of detection*, or *true positive rate*, is the ratio of correctly predicted vulnerable entities to actual vulnerable entities as defined in Equation (2) :

$$recall = \frac{TP}{TP + FN} \tag{2}$$

Precision is the ratio of correctly predicted vulnerable entities to all predicted vulnerable entities as defined in Equation (3) :

$$precision = \frac{TP}{TP + FP} \tag{3}$$

Probability of False alarm (PF), also known as *false positive rate* is the ratio of entities incorrectly predicted as vulnerable to actual neutral entities as defined in Equation (4) :

$$PF = \frac{FP}{FP + TN} \tag{4}$$

The desired result is to have a high recall, high precision, and a low PF to find as many vulnerabilities as possible without wasting inspection or testing effort. Having a high recall is especially important in software security considering the potentially high impact of a single exploited vulnerability.

All the performance measures discussed above set the classification threshold at a single threshold. However, prediction models can provide their best performance at different thresholds. To summarize the prediction performance at various thresholds, AUC (Area Under the Curve) [79] can be used. The curve, called a Receiver Operating Characteristic (ROC) curve, consists of points of recall (y-axis) and PF (x-axis) measured at different classification thresholds. AUC measures the area under the ROC curve. Figure 2 shows an example of two ROC curves. If a classification result is randomly chosen instead of using a prediction model, recall and PF will be approximately the same at each threshold. For example, if the threshold is 0.5 for random classification, 50% of actual vulnerable entities will be classified as vulnerable resulting in recall of 0.5 and 50% of actual non-vulnerable entities will be also classified as vulnerable entities resulting in PF of 0.5. If the threshold is 0.9 for random classification, 90% of actual vulnerable entities will be classified as vulnerable resulting in recall of 0.9 and 90% of actual non-vulnerable entities will be also classified as vulnerable entities resulting in PF of 0.9 [20]. This is true regardless of the distribution of vulnerable and non-vulnerable entities. The diagonal line (ROC A) is the ROC

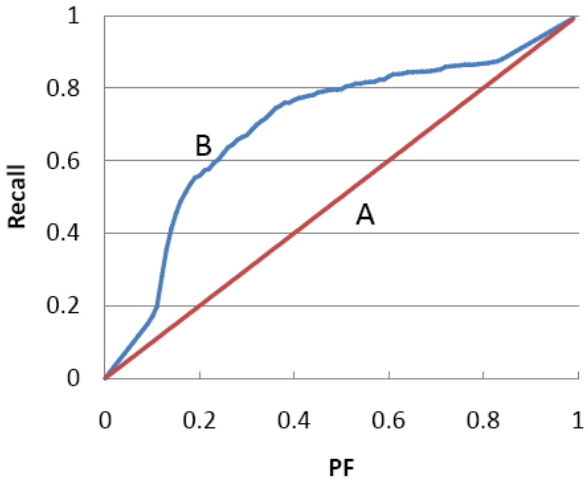

Figure 2. ROC curve

curve by a random classification model. When the ideal prediction performance is achieved with recall of 1 and PF of 0, AUC is 1. AUC over 0.5 (ROC B) indicates the prediction model can predict better than a random classification model.

4.3 Hypothesis Test for Predictability

To statistically support a hypothesis for predictability, a prediction model using complexity metrics should be able to predict vulnerable code location better than a random classification model. This research tests the null hypothesis that the AUC from a prediction model using complexity metrics is equal to or less than 0.5. If this hypothesis is rejected, the model has predictability.

Because the criterion of just being better than a random classification models seems to be too loose, this study also seeks a way to compare the prediction results with a more

stringent threshold. Since there is no universally applicable standard for the threshold of desired vulnerability prediction performance, this study uses the average prediction performance found from literature [44, 51, 52] as a *performance benchmark*. By comparing the results from this research with results from other recent studies, we can roughly judge whether the prediction performance in this research is at least comparable to the current state of the art fault prediction performance.

The study in Chapter 5 (which was performed earlier than the study in Chapter 7) uses the performance benchmark with recall of 0.7 and PF of 0.25 based on the averaged results of Naïve Bayes classification models over eight projects from the Menzie's et al.'s study [51, 52]. The study in Chapter 7 uses the performance benchmark of AUC of 0.785 based on the averaged results of logistic regression models over ten projects from Lessman et al.'s study [44]. The reason why the results from the two studies were selected as performance benchmarks is that those studies use multiple publicly-available, non-trivial sized projects (8KLOC to 30KLOC) from the NASA MDP data repository [1] that have been also used for many other fault prediction studies [30, 36, 47, 73] and provide the percentage of faulty modules. As will be discussed in Chapter 6 and Chapter 7, the prediction performance is radically different depending on the percentage of faulty or vulnerable entities. The projects used in the two studies had faults in 0.5% to 49% of the modules with

average of 11%, while the projects used in Chapter 7 in this research had vulnerabilities in 1.4% to 7.8% of the files and 14% to 69% of the components.

Note that although performance benchmark is useful for comparison purpose, desirable levels of recall and PF depend on varying domains and business goals.

4.4 Code Inspection Cost and Inspection Reduction Efficiency Measurements

Practitioners must consider how effective the prediction model is in prioritizing the code locations to inspect and test and in reducing the effort for code inspection and testing. This study uses the number of entities and the lines of code to inspect as partial and relative estimators of the inspection effort. Such measures have been used as cost estimators in prior studies [4, 63]. This research formally defines the formula to measure inspection cost and the efficiency in inspection reduction based on the results from vulnerability prediction models. Overall inspection is reduced if the percentage of the entities to be inspected is smaller than the percentage of the entities predicted as vulnerable and if the percentage of lines of code to be inspected is smaller than the percentage of vulnerabilities in the entities predicted as vulnerable [4, 63]. For example, if we randomly choose files to be inspected, we need to inspect 80% of the total files to obtain recall of 0.8. If a prediction model provides recall of 0.8 with less than 80% of the total files, the model reduced the cost for inspections compared to a random classification model. The two cost measurements are formally defined as below:

File Inspection (FI) ratio is the ratio of entities predicted as vulnerable (that is, the number of entities to be inspected) to the total number of entities for the reported recall as defined in Equation (5):

$$FI = \frac{TP + FP}{TP + FP + TN + FN} \tag{5}$$

For example, *recall*=0.8 and *FI*=0.2 mean that within the 20% of files inspected based on the prediction results, 80% of vulnerable files can be found.

LOC Inspection (LI) ratio is the ratio of lines of code to inspect to the total lines of code for the predicted vulnerabilities as defined in Equation (6). First, we define lines of code in the entities that were true positives, as TP_{LOC}, similarly with TN_{LOC}, FP_{LOC}, and FN_{LOC}. Then *LI* is defined below:

$$LI = \frac{TP_{LOC} + FP_{LOC}}{TP_{LOC} + FP_{LOC} + TN_{LOC} + FN_{LOC}} \tag{6}$$

While *FI* and *LI* estimate how much effort is involved, we need measures to provide how much effort is reduced. We define two cost-reduction measurements.

File Inspection Reduction (FIR) is the ratio of the reduced number of entities to be inspected by using the model with CCD metrics compared to a random selection to achieve the same recall as defined in Equation (7):

$$FIR = \frac{recall - FI}{recall} \tag{7}$$

LOC Inspection Reduction (LIR) is the ratio of reduced lines of code to be inspected by using a prediction model compared to a random selection to achieve the same *PV* as defined in Equation (8):

$$LIR = \frac{PV - LI}{PV} \tag{8}$$

where *PV* is defined as below.

Predicted Vulnerability (PV) ratio is the ratio of the number of vulnerabilities in the entities predicted as vulnerable to the total number of vulnerabilities. First, we define the number of vulnerabilities in the entities that were true positives, as TP_{Vuln}, similarly with TN_{Vuln}, FP_{Vuln}, and FN_{Vuln}. Then *PV* is defined in Equation (9):

$$PV = \frac{TP_{Vuln}}{TP_{Vuln} + TN_{Vuln}} \tag{9}$$

Note that a small number of large entities may contain many entities 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.

4.5 Prediction Modeling and Result Validation

This study used logistic regression to predict vulnerable entities in Chapters 5, 6, and 7. Logistic regression models the probability of occurrence of an outcome event from given independent variables by mapping the linear combination of independent variables to the probability of outcome using the log of odds ratio (logit). An entity is classified as vulnerable when the estimated outcome probability is greater than 0.5. As a preliminary study, other classification techniques including J48 decision tree [79], Random forest [9], Naïve Bayes [79], and Bayesian network [79] also have been used. For the study in Chapter 5, the J48, Random forest, and Bayesian network models provided similar results to the logistic regression model, while the Naïve Bayes model provided a higher recall with a lower precision than other techniques. Lessmann et al. also reported that no significant difference in prediction performance was found between the 17 classification techniques they investigated [44]. Therefore, this dissertation presents the results from only logistic regression.

To validate a model's predictability, this research used two types of validation methods: next-release validation and cross-validation. For the study in Chapter 5, this research collected data from multiple releases of Mozilla Firefox and performed next-release validation. For RHEL4 kernel studied in Chapter 5 and for all other projects in Chapters 6 and 7, 10x10 cross-validation [79] was used at file level and 3x3 cross-validation was used at component level because the number of components was too few to split into ten folds. For

next-release validation in Chapter 5, data from the most recent three previous releases were used to train against the next release (i.e. train on releases R-3 to R-1 to test against release R). Using only recent releases was to accommodate for process change, technology change, and developer turnovers. For 10x10 cross-validation, we randomly split the data set into 10 folds and used one fold for testing and the remaining folds for training, rotating each fold as the test fold. Each fold was stratified to properly distribute vulnerable files to both the training data set and the test data set. The entire process was then repeated ten times to account for possible sampling bias in random splits. Overall, 100 predictions were performed for 10x10 cross-validation and 9 predictions for 3x3 cross-validation.

Using many metrics in a model does not always improve the prediction performance since metrics can provide redundant information [51]. Therefore, each study in this research selected only a small set of variables using the information gain ranking method [79]. As a preliminary study, the correlation-based greedy feature selection method [79] was also investigated. While both methods provided similar prediction performance, the set of chosen variables by the two selection methods were different. This dissertation presents the results from the models using the information gain ranking method.

This research performed the prediction on both the raw metric values and the log transformed metric values. For logistic regression, recall was improved at the sacrifice of precision after log transformation. This dissertation presents results using log transformation.

The data used in this research is heavily unbalanced between majority class (neutral entities) and minority class (vulnerable entities), when an entity is at the file level or at the class level. Prior studies have shown that the performance is improved (or at least not degraded) by "balancing" the data [37, 52]. Balancing the data can be achieved by duplicating the minority class data (over-sampling) or removing randomly chosen majority class data (under-sampling) until the numbers of data instances in the majority class and the minority class become equal [52]. This research used under-sampling since under-sampling provided better results than using the unbalanced data and reduced the time for evaluation.

Figure 4 and Figure 3 provide the pseudo code of our experimental design explained above for next-release validation and 10x10 cross-validation. The whole process of validation is repeated ten times (three times at component level) to account for possible sampling bias due to random removal of data instances. The ten times of repetition also account for the bias due to random splits in cross-validation. In Figure 4 and Figure 3, *performance* represents the set of evaluation criteria of prediction results, cost-reduction measurements, and their relevant measurements defined in Chapter 4. For next-release validation, the input is three prior releases for training a model and one release for testing. For cross-validation, the input is the whole data set.

Prediction models were built using Weka 3.7[11] with default options except for limiting the number of variables to be selected.

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

```
next_release_validation (Set R1, Set R2, Set R3, Set R4) {
    S_train ← R1 ∪ R2 ∪ R3
    S_test ← R4
    performance ← 0
    repeat 10 times {
        S_train_vulnerable ← vulnerable entities in S_train
        N ← |S_train_vulnerable|
        S_train_neutral ← N neutral entities remaining after random removal from S_train
                                    for under-sampling
        S_train ← S_train_vulnerable ∪ S_train_neutral
        V ← select variables using the InfoGain variable selection method from S_train
        Train the model M on S_train and variables V
        performance ← performance + prediction performance of M tested on S_test
    }
     return  performance ← performance / 10
}
```

Figure 4. Pseudo code for next-release-validation

```
cross_validation (Set R) {
    performance ← 0
    repeat 10 times {
        Randomly split R into 10 stratified bins, B = {b1, b2, ..., b10}
        for each bin bi {
            S_train ←B – { b_i }
            S_test ← b_i
            S_train_vulnerable← vulnerable entities in S_train
            N ← |S_train_vulnerable|
            S_train_neutral ← N neutral entities remaining after random removal from S_train
                                    for under-sampling
            S_train ←S_train_vulnerable ∪ S_train_neutral
            V ← select variables using the InfoGain selection method from S_train
            Train the model M on S_train and variables V
            performance ← performance + prediction performance of M tested on S_test
        }
    }
    return  performance ← performance / 100
}
```

Figure 3. Pseudo code for 10x10 cross-validation

CHAPTER 5

Code Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities

Complexity can make code difficult to understand and to test. Therefore, software developers can introduce more errors in complex code than in simple code and the errors in complex code are more likely to remain undetected during testing [15, 45, 76]. For this reason, complexity metrics has been used to predict faults [5, 10, 36, 38, 51, 53, 57, 58, 70, 81]. While complexity metrics has been investigated as indicators of faults in general, applicability of complexity metrics as indicators of vulnerabilities has not been investigated in depth yet.

Experience indicates that the detection and mitigation of vulnerabilities is best done by engineers specifically trained in software security and who "think like an attacker" in their daily work [46]. Therefore, security testers need to have specialized knowledge in and a mindset for what attackers will try. If we could predict which parts of the code are likely to be vulnerable, security experts can focus on these areas of highest risk. Although prediction models trained using complexity metrics to find faults have been known to be effective, prediction models may need to be trained on what they are intended to look for. Rather than

arming the security expert with all the modules likely to contain faults, a security prediction model can point toward the set of modules likely to contain what a security expert is looking for: security vulnerabilities. Establishing predictability in a security prediction model is challenging because security vulnerabilities and non-security-related faults have similar symptoms. Differentiating a vulnerability from a fault can be nebulous even to a human, much less a statistical model. Additionally, the number of reported security vulnerabilities with which to train a model are few compared to non-security-related faults. Colloquially, security prediction models are "searching for a needle in a haystack."

The goal of this chapter is to investigate whether code complexity metrics can be used to discriminate between vulnerable and neutral files, and to predict vulnerable code locations.

As an initial study to investigate the relationship between software complexity and vulnerabilities, this study focuses on code complexity metrics. Other types of complexity metrics including OO design complexity, dependency network complexity, and execution complexity metrics are examined in Chapter 7.

Although the main interest of this research is on complexity metrics as indicators of vulnerabilities, other process metrics that have been used for fault prediction [4, 29, 41, 48, 55, 56, 62, 77] are also interesting for the comparison purpose with complexity metrics. This study investigates two process metrics, code churn (code change) [29, 55, 56] and developer

activity metrics [48], as indicators of vulnerabilities together with code complexity metrics. Code churn and developer activity metrics have been known to be effective for fault prediction [29, 48, 55, 56]. These metrics are also interesting as indicators of vulnerabilities because frequent or large amount of code change can introduce errors including vulnerabilities. Poor developer collaboration can diminish project-wide secure coding practices. Hence, metrics that measure code churn and developer activity also may be able to indicate vulnerable code locations. For convenience, we call Code complexity, Code churn, and Developer activity metrics *CCD metrics* in this study.

To achieve the goal of this study, empirical case studies have been conducted on two widely-used, large scale open source projects: the Mozilla Firefox web browser and the Linux kernel as distributed in Red Hat Enterprise Linux. This study analyzed Mozilla Firefox and Red Hat Enterprise Linux (each of them containing over two million lines of source code), and evaluated the adequacy of using CCD metrics as indicators of security vulnerabilities. Because the ultimate gold is to efficiently allocate inspection and testing effort, this study also measured the reduction in code inspection effort using CCD metrics against random file selection.

Note that while code complexity metrics are always available, process metrics are not always available especially when a software project is an initial project. Therefore, both the availability of metrics and the ability of the metrics as indicators of vulnerabilities are the

factors to be considered to determine whether CCD metrics can be used to find vulnerable code locations for projects.

5.1 Hypotheses and Evaluation Criteria

This section provides the evaluation criteria of hypotheses for discriminative power and predictability.

5.1.1. Hypotheses for Discriminative Power

This subsection provides the hypotheses of using the CCD metrics as indicators of vulnerabilities.

5.1.1.1 Code Complexity

As discussed in Section 2.2, complexity can lead to subtle vulnerabilities that are difficult to test and diagnose, providing more chances for attackers to exploit. Therefore, complex code would be more vulnerable than simple code. Therefore, the hypothesis for discriminative power for code complexity metrics in this study is:

$H_{CodeComplexity\_D}$*: Vulnerable files have a higher code complexity than neutral files.*

5.1.1.2 Code Churn

Code is constantly evolving throughout the development process. Each new change in the system brings a new risk of introducing a vulnerability [29, 55]. Therefore, the hypothesis for discriminative power for code churn metrics in this study is:

$H_{CodeChurn\_D}$: Vulnerable files have more frequent and large changes than neutral files.

Table 6 defines the code churn metrics that we use in this study.

Table 6. Definitions of code churn metrics

Related Hypothesis	Metric	Definition
$H_{CodeChurn\_D,P}$	NumChanges	The number of check-ins for a file since the creation of a file
	LinesChanged	The cumulated number of code lines changed since the creation of a file
	LinesNew	The cumulated number of new code lines since the creation of a file

5.1.1.3 Developer Activity

Software development is performed by development teams working together on a common project. Lack of team cohesion, miscommunications, or misguided effort can all result in security problems [46]. Version control data can be used to construct a developer network and a contribution network based upon "which developer(s) worked on which file." This study uses network analysis to represent developer network and contribution network.

The centrality metrics this study uses for developer activity are degree, closeness, and betweenness. The degree metric is defined as the number of neighbors directly connected to a node. The closeness centrality of node *v* is defined as the average distance from *v* to any

other node in the network that can be reached from *v*. The betweenness centrality [8] of node *v* is defined as the number of geodesic paths that include *v*.

Cluster metrics are used to measure the strength of interconnection between groups of nodes. A cluster of nodes is a set of nodes such that there are more edges within a set of nodes (intra-set edges) than edges between a set and other sets of nodes (inter-set edges). The cluster metric we use for developer activity is edge betweenness [28]. The edge betweenness of edge *e* is defined as the number of geodesic paths that pass through e. Since clusters have many intra-cluster edges, edges within clusters have a low betweenness; conversely, edges between two clusters have a high betweenness [28].

5.1.1.3.1 Developer Network Centrality

In a developer network, two developers are connected if they have both made a change to at least one file in common during the period of time under study. The result is an undirected, unweighted, and simple graph where each node represents a developer and edges are based on whether or not they have worked on the same file during the same release. Central developers, measured by a high degree, high betweenness, and low closeness, are developers that are well-connected to other developers relative to the entire group. Refer to [48] for a more in-depth example of how centrality metrics are derived from developer networks. A central developer would likely have a better understanding of the group's

secure coding practices because of his/her connections to the other developers of the team. Therefore:

$H_{DeveloperCentrality\_D}$: Vulnerable files are more likely to have been worked on by non-central developers than neutral files.

The metrics used for these hypotheses are shown in Table 7. Note that DNMaxDegree, DNMinCloseness, and DNMaxBetweenness have been excluded from this study because, for example, a high DNMaxDegree means at least one central developer

Table 7. Meaning of developer activity metrics

Related Hypothesis	Metric	Problematic When	Meaning
$H_{DeveloperCentrality\_D}$	DNMinDegree	Low	File was changed by developers who are not central to the network
	DNAvgDegree	Low	
	DNMaxCloseness	High	
	DNAvgCloseness	High	
	DNMinBetweenness	Low	
	DNAvgBetweenness	Low	
$H_{DeveloperCluster\_D}$	DNMaxEdgeBetweenness	High	File was contributed to by more than one cluster of developers, with few other files being worked on by each cluster
	DNAvgEdgeBetweenness	High	
$H_{ContributionCentrality\_D}$	NumDevs	High	File was changed by many developers
	CNCloseness	High	File was changed by developers who focused on many other files

worked on the file, which is not as helpful as knowing that high DNMinDegree denotes that all developers who worked on a file were central. Note also that a high turnover rate in a project results in many non-central developers: when new developers are added to the project, they initially lack connections to the other developers.

5.1.1.3.2 Developer Network Cluster

Metrics of developer centrality give us information about individual developers, but the relationship between groups of developers is also interesting. In a developer network, a file that is between two clusters was worked on by two groups of developers, and those two groups did have many other connections in common. Clusters with a common connection, but few other connections, may not be communicating about improving the security of the code they have in common. Therefore:

$H_{DeveloperCluster\_D}$*: Vulnerable files are more likely to be changed by multiple, separate developer clusters than neutral files.*

Since edges and files have a many-to-many relationship, this study uses the average and maximum of edge betweenness on the developer network as provided in Table 7.

5.1.1.3.3 Contribution Network

A contribution network [65] is a quantification of the focus made on the relationship between a file and developers instead of relationship between developers as in developer

networks. The contribution network uses an undirected, weighted, and bipartite graph with two types of nodes: developers and files. An edge exists where a developer made changes to a file, where the weight is equal to the number of check-ins that developer made to the file. If a file has high centrality, then that file was changed by many developers who made changes to many other files, referred to as an "unfocused contribution." [65] Files with an unfocused contribution may not get the attention required to prevent the injection of vulnerabilities. Therefore:

$H_{ContributionCentrality\_D}$: Vulnerable files are more likely to have an unfocused contribution than neutral files.

Note that developers with high centrality can work with many people, but still work on one small part of the system. Being a central developer means being central in terms of people and not necessarily central in terms of the system. Contribution networks, on the other hand, are structured around the system.

Table 7 provides the meaning of the contribution network metrics.

5.1.2. Hypotheses for Predictability

This study sets up four hypotheses for each type of CCD metrics and the combined set of CCD metrics as below:

$H_{CodeComplexity\_P}$: A model with a subset of code complexity metrics can predict vulnerable files.

$H_{CodeChurn\_P}$: A model with a subset of code churn metrics can predict vulnerable files.

$H_{Developer\_P}$: A model with a subset of developer activity metrics can predict vulnerable files.

H_{CCD_P}: A model with a subset of combined CCD metrics can predict vulnerable files.

Additionally this study investigates whether the individual metrics can predict vulnerable files.

$Q_{Individual\_P}$: Can a model with individual CCD metric predict vulnerable files?

5.1.3. Evaluation Criteria for Hypotheses Testing

To support a hypothesis for discriminative power defined in this section, the Welch's t-test should be significant at 0.05 level after Bonferroni correction as discussed in Section 4.1. Additionally, the association direction (i.e. positively or negatively correlated) also should match with the hypothesis. To see the association direction, this study compared the means and medians of the measures of CCD metrics for the vulnerable and neutral files.

A metric has predictability when AUC is over 0.5. The prediction performance is also tested against the performance benchmark: recall of 0.5 and PF of 0.25.

5.2 Case Study 1: Mozilla Firefox

The first case study is Mozilla Firefox, a widely-used open source web browser. Mozilla Firefox had 34 releases at the time of data collection developed over four years. Each release consists of over 10,000 files and over two million lines of source code.

5.2.1. Data Collection

To measure the number vulnerabilities fixed in a file, this study counted the number of bug reports that include the details on vulnerabilities and on how the vulnerabilities have been fixed for the file. This study collected vulnerability information from Mozilla Foundation Security Advisories (MFSAs). Each MFSA includes bug ids that are linked to the Bugzilla bug tracking system[12]. Mozilla developers also add bug ids to the log of the CVS version control system[13] when they check in files to the CVS after the vulnerabilities have been fixed. This study searched the bug ids from the CVS log to find the files that have been changed to fix vulnerabilities, similar to the approach found in other studies [59]. The number of MFSAs for Firefox was 197 as of 2 August, 2008. The vulnerability fixes for the MFSAs were reported in 560 bug reports. Among them, 468 bug ids were identified from the CVS log. Although some of the files were fixed for regression, all those files were also fixed for vulnerabilities. Therefore, those files were counted as vulnerable in this study.

In this study, code complexity metrics are collected from only C/C++ and their header files, excluding other files types such as script files and make files. Code churn and developer activity metrics were collected from the CVS version control system.

At the time of data collection, Firefox 1.0 and Firefox 2.0.0.16 were the first and the last releases that had vulnerability reports. The gap between Firefox releases ranged from one

---

[12] https://bugzilla.mozilla.org/
[13] https://developer.mozilla.org/en/Mozilla_Source_Code_Via_CVS

to two months. Since each release had only a few vulnerabilities (not enough to perform analysis at each Firefox release), this study combined the number of vulnerabilities for three consecutive releases, and will refer to those three releases as a combined release, denoting each combined release as R*n*. For each release, we used the most recent three combined releases (RN-3 to RN-1) to predict vulnerable files in RN. This study collected metrics for 11 combined releases. Table 8 provides the project statistics for the combined 11 releases.

Table 8. Project statistics for Mozilla Firefox

No.	Firefox Release	# of Files	LOC	Mean LOC	Files with Vulns.	Total Vulns.	Vulns. per File	% of Files with Vulns.
R1	1.0 / 1.0.1 / 1.0.2	10,320	2,060,908	200	70	84	0.008	0.678
R2	1.0.3 / 1.0.4 / 1.0.5 / 1.0.6[14]	10,321	2,063,960	200	123	134	0.013	1.192
R3	1.0.7 / 1.5 / 1.5.0.1	10,321	2,064,747	200	93	159	0.015	0.901
R4	1.5.0.2 / 1.5.0.3 / 1.5.0.4	10,956	2,226,540	203	100	138	0.013	0.913
R5	1.5.0.5 / 1.5.0.6 / 1.5.0.7	10,961	2,230,313	204	109	153	0.014	0.994
R6	1.5.0.8 / 2.0 / 2.0.0.1	10,961	2,232,890	204	87	124	0.011	0.794
R7	2.0.0.2 / 2.0.0.3 / 2.0.0.4	11,060	2,294,287	207	114	162	0.015	1.031
R8	2.0.0.5 / 2.0.0.6 / 2.0.0.7	11,060	2,299,054	208	55	72	0.007	0.497
R9	2.0.0.8 / 2.0.0.9 / 2.0.0.10	11,076	2,301,398	208	14	15	0.001	0.126
R10	2.0.0.11 / 2.0.0.12 / 2.0.0.13	11,077	2,301,832	208	84	110	0.010	0.758
R11	2.0.0.14 / 2.0.0.15 / 2.0.0.16	11,080	2,304,048	208	27	46	0.004	0.244

5.2.2. Discriminative Power Test and Univariate Prediction Results

Table 9 shows the results of hypothesis testing for discriminative power and the univariate prediction using logistic regression for individual metrics from release R4, the first

[14] The vulnerabilities for Firefox 1.0.5 and Firefox 1.0.6 were reported together. Therefore, we considered those two versions as one version.

test data set that uses the most three recent releases to train a prediction model. In Table 9, '√' for Welch's t-test indicates that the null hypothesis that the metric values for vulnerable files and neutral files are the equal is rejected at $p < 0.05$ after Bonferroni correction; the plus sign in the Association column indicates that the vulnerable files had a higher measure than neutral files; '√' for H_D indicates both the Welch's t-test result and the association direction match with the corresponding hypothesis defined in Section 5.1.

DNAvgDegree was the only sign that did not completely agree with its hypothesis. We discuss the reason in Section 5.4. 27 of the 28 metrics showed a statistically significant difference between vulnerable and neutral files using the Welch's t-test after a Bonferroni correction as shown in Table 9.

Table 9. Results of discriminative power test and univariate prediction for Mozilla Firefox

		Discriminative Power			Predictability						
					Mean for R4				Std. for R4		
Related Hypothesis	**Metric**	**Welch-t**	**Association**	**H_D***	**AUC**	**R***	**PF***	**Q_P***	**R**	**PF**	**N(Q_P)***
	Code Complexity										
$H_{IntraCode\_D}$	LOC	√	+	√	0.84	0.82	0.28	X	0.004	0.009	1
	NumDeclFunction	√	+	√	0.79	0.87	0.68	X	0.000	0.000	0
	LOCVarDecl	√	+	√	0.85	0.79	0.25	√	0.000	0.008	47
	NumLinePreprocessor	√	+	√	0.80	0.76	0.26	X	0.009	0.009	0
	SumEssential	√	+	√	0.80	0.86	0.60	X	0.003	0.030	0
	SumCyclomaticStrict	√	+	√	0.79	0.86	0.60	X	0.000	0.012	0
	MaxCyclomaticStrict	√	+	√	0.76	0.87	0.68	X	0.000	0.000	0
	SumMaxNesting	√	+	√	0.79	0.82	0.53	X	0.000	0.000	0
	MaxMaxNesting	√	+	√	0.76	0.82	0.53	X	0.000	0.000	0
$H_{Coupling\_D}$	SumFanIn	√	+	√	0.81	0.87	0.57	X	0.000	0.000	0
	SumFanOut	√	+	√	0.79	0.86	0.62	X	0.000	0.000	0
	MaxFanIn	√	+	√	0.79	0.87	0.57	X	0.000	0.000	0
	MaxFanOut	√	+	√	0.75	0.86	0.62	X	0.000	0.000	0
$H_{Comments\_D}$	CommentDensity	X	–	X	0.59	0.59	0.45	X	0.154	0.218	0
	Code Churn										
$H_{NumChanges\_D}$	NumChanges	√	+	√	0.88	0.86	0.23	√	0.008	0.012	80
$H_{ChurnAmount\_D}$	LinesChanged	√	+	√	0.88	0.85	0.25	√	0.003	0.014	76
	LinesNew	√	+	√	0.83	0.88	0.58	X	0.009	0.091	0
	Developer Activity										
$H_{DeveloperCentrality\_D}$	DNMinDegree	√	–	√	0.30	0.74	0.84	X	0.042	0.010	0
	DNAvgDegree	√	+	X	0.54	0.98	0.77	X	0.000	0.009	0
	DNMaxCloseness	√	+	√	0.77	1.00	0.95	X	0.000	0.001	0
	DNAvgCloseness	√	+	√	0.55	1.00	0.95	X	0.000	0.000	0
	DNMinBetweenness	√	–	√	0.55	0.53	0.45	X	0.005	0.307	0
	DNAvgBetweenness	√	–	√	0.46	0.99	0.87	X	0.003	0.008	0
$H_{DeveloperCluster\_D}$	DNMaxEdgeBetweenness	√	+	√	0.84	0.88	0.30	X	0.000	0.000	0
	DNAvgEdgeBetweenness	√	+	√	0.80	0.88	0.30	X	0.000	0.000	0
$H_{ContributionCentrality\_D}$	NumDevs	√	+	√	0.87	0.84	0.24	√	0.015	0.016	58
	CNCloseness	√	+	√	0.78	0.82	0.22	√	0.000	0.000	80
	CNBetweenness	√	+	√	0.78	0.64	0.9	X	0.000	0.000	20

*. H_D: Hypothesis test for discriminative power, R: Recall, PF: Probability of false alarm, Q_P: Question for predictability, N(Q_P): Number of predictions that provided recall of over 0.7 with PF of less than 0.25 from 80 predictions

We also compared the mean and median of each measurement for vulnerable and neutral files to find the association direction (positive or negative). Figure 5 shows the boxplots of comparisons of log scaled measures between vulnerable and neutral files for the three representative metrics from each type of CCD metrics: LOC, NumChanges, and NumDevs. The medians of the three metrics for the vulnerable files were higher than the ones for the neutral files, as hypothesized in this study. From Figure 5, we can also observe that vulnerable files and neutral files have different distribution of metric values; the metric values for neutral files show skewed distribution with long tails for high metric values different from the metric values for vulnerable files.

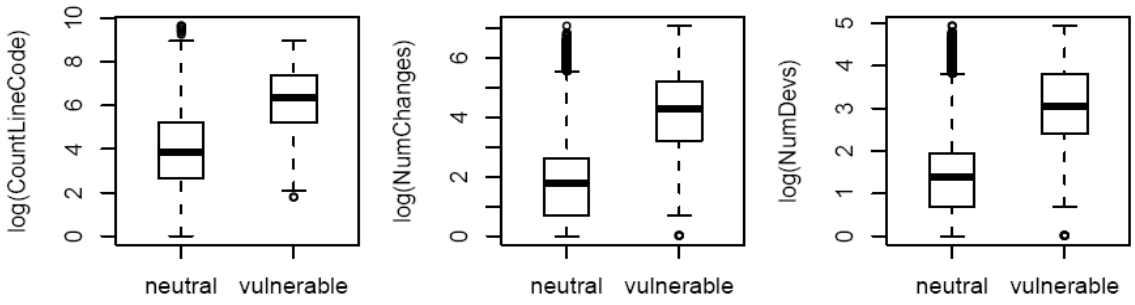

Figure 5. Boxplots for metric values for vulnerable and neutral files

In Table 9, the means of recall, PF, and AUC are averaged over ten repetitions of predictions for R4 according to the algorithm provided in Figure 4. $N(Q_P)$ represents the number of predictions that achieve the performance benchmark (recall of 0.7 and PF of 0.25) in the 80 predictions (ten repetitions for each of the eight combined releases). Most metrics

provided better than random prediction performance based on AUC except for DNMinDegree and DNAvgBetweenness. However, only four metrics (NumChanges, LinesChanged, NumDevs, and CNCloseness) achieved the performance benchmark in over half of the predictions. Most metrics provided very small variations (less than 0.03 of standard deviation) between predictions in both recall and PF except for CommentDensity, LinesNew, and DNMinBetweenness. Note that this study provides N(Qp) instead of averaging the prediction results across all releases because averaging the results for different data sets can mislead the interpretation of the results from each data set [16].

Note that a single metric can provide such a high recall and a low PF. Next subsection investigates the predictability of CCD metrics when they are used together in a model.

5.2.3. Results from Multiple Regression Prediction Models

Although metrics can have low predictability individually, combining metrics into a model can result in better predictability [31]. Therefore, this study created four types of models using code complexity, code churn, developer activity, and combination of the CCD metrics. From each set of the metrics, three variables were selected using InfoGain variable selection method as discussed in Section 4.5.

Figure 6 shows recall, PF, FI, LI, FIR, and LIR across releases where the results from each release were averaged over the results from the ten repetitions of next-release validation using logistic regression. All the models with code churn, developer activity, and combined

CCD metrics provided similar results across releases; recall was between 0.68 to 0.88 and PF was between 0.17 to 0.26; File Inspection ratio (FI) was between 0.17 to 0.26 resulting in 66% to 80% reduction in the files to inspect; Line Inspection ratio (LI) was between 0.49 to 0.65 resulting in 16% to 42% of reduction in lines of code to inspect. The models using code complexity metrics provided recall of 0.76 to 0.86 and PF of 0.22 to 0.34. FI for the models using code complexity metrics was between 0.22 to 0.34 resulting in 61% to 72% reduction in file inspection. LI for the models using code complexity metrics was much higher than LI for the other models (0.74 to 0.85) resulting in only 7% reduction in code inspection at best. In the worst case, 3% more code is required to be inspected than the percentage of lines of code chosen by random selection. Note that PF and FI showed almost identical performance for all models. We discuss the similarity between PF and FI in Section 5.4.

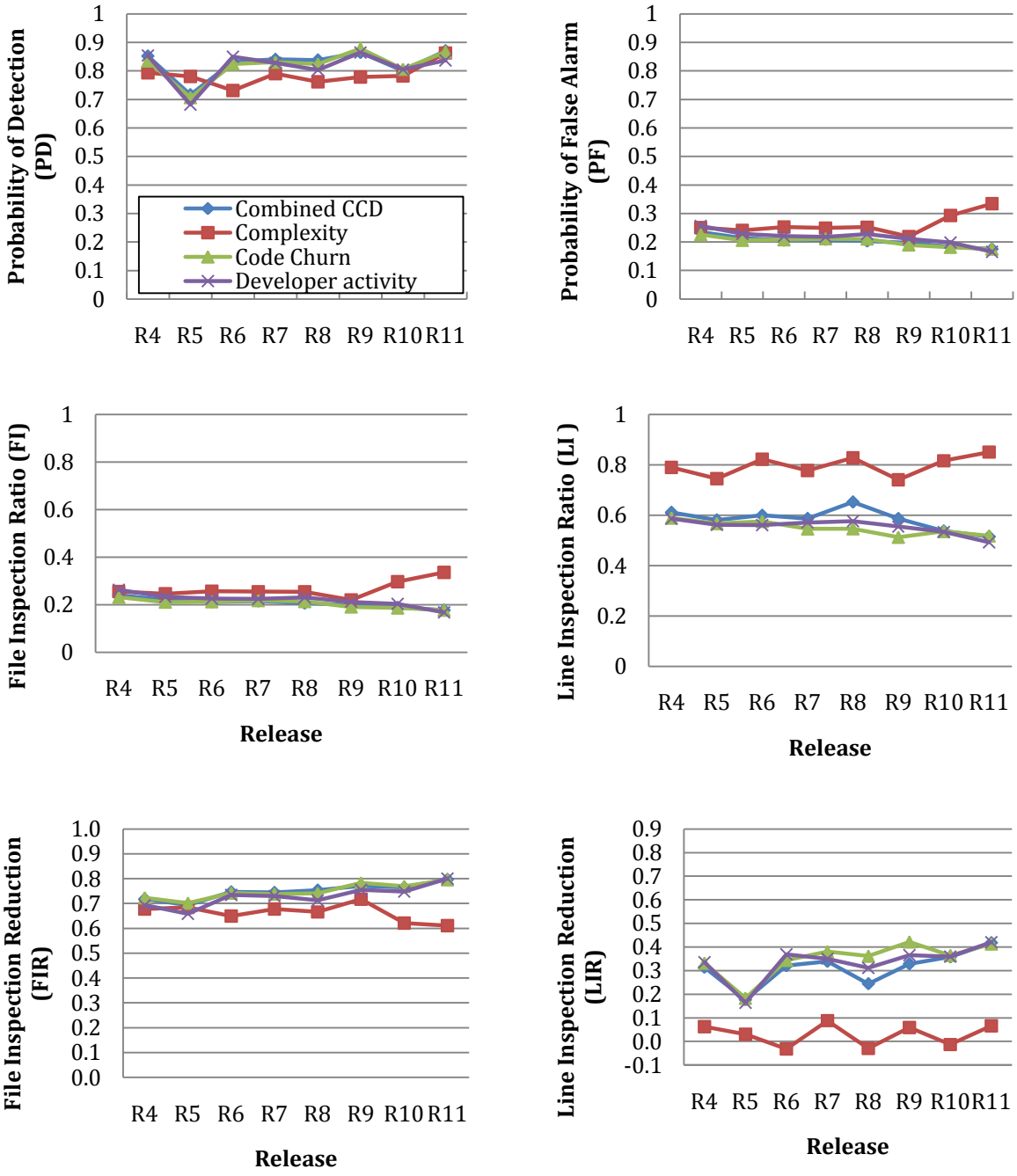

Figure 6. Prediction results for Mozilla Firefox across releases

The three types of models generally provided consistent recall except for R5 and R10. The models using code complexity metrics provided lower recall than other models, but provided consistent results even in R5 and R10. Instead, PF from the models using code complexity metrics suddenly increased in R10. We conjecture the reason of the sudden decrease in performance for R10 is because the number of vulnerable files in R10 suddenly increases when the model was trained with two recent releases (R8 and R9) with relatively few vulnerabilities. However, we were not able to find out any particular reason for the sudden decreases in performance for R5. All the prediction results from the three types of models gradually improved as the system matures, but the results from models using code complexity metrics stayed constant or degraded.

Table 10 provides detailed results for release R4 including standard deviations as illustration of the results from a combined release. AUC was over 0.85 for all models indicating the prediction is much better than a random classification model. The models with code churn metrics and combined CCD metrics achieved the performance benchmark (recall of 0.7 and PF of 0.25) in over 90% of predictions out of 80 predictions (ten repetitions for each of the eight combined releases). All of the models satisfied the prediction criteria in over 50% of the 80 predictions. Standard deviations in recall and PF were less than 0.03 in all models for release R4. Interestingly, no models with multiple metrics were noticeably better than some of the best univariate models.

Table 10. Results from multiple regression prediction models for Mozilla Firefox

	Predictability for R4							Inspection Costs for R4				
	Mean				Std.			Costs			Cost Reduction	
	AUC	R	PF	Prec.	R	PF	$N(H_P)$	PV	FI	LI	FIR	LIR
Code complexity	0.85	0.79	0.25	0.03	0.007	0.008	40	0.84	0.26	0.79	0.68	0.06
Code churn	0.87	0.84	0.23	0.03	0.012	0.013	76	0.88	0.23	0.59	0.72	0.33
Developer activity	0.88	0.85	0.26	0.03	0.017	0.028	62	0.88	0.26	0.59	0.69	0.34
Combined CCD	0.88	0.85	0.24	0.03	0.018	0.017	74	0.89	0.24	0.61	0.72	0.31

The amount of files to inspect reduced by 68% to 72% depending on the models for R4. The amount of lines of code to inspect reduced by 31% to 34% with code churn, developer activity, and combined CCD metrics, and only 6% with code complexity metrics for release R4.

5.3 Case Study 2: Red Hat Enterprise Linux Kernel

The second case study has been performed on the Linux kernel as it was distributed in the Red Hat Enterprise Linux 4 (RHEL4) operating system. The RHEL4 kernel consists of 13,568 C files with over three million lines of code. The details of the project, data collection, and the prediction results are described in this section.

5.3.1. Data Collection

Gathering security data involved tracing through the development artifacts related to each vulnerability reported in the Linux kernel. This study collected our vulnerability data from the Red Hat Bugzilla database and the Red Hat package management system (RPM). Since some vulnerability patches affect only certain releases, we examined each defect report

manually to ensure that developers had decided that patch was, in fact, required. Instead of scanning developer commit logs for defect IDs, this study used the RPM system to determine the exact patch that was issued to fix each of the 258 known vulnerabilities. Since we are only interested in vulnerabilities that existed at the time of release, this study did not include vulnerabilities introduced by post-release patches (a.k.a "regressions") in the collected data set. For the few vulnerabilities that did not have all of the relevant artifacts (e.g. defect reports, patches), We consulted the director of the RHSR team to correct the data and the artifacts. This study collected vulnerabilities reported from February 2005 through July 2008.

To obtain code churn and developer activity data, this study used the Linux kernel source repository[15]. The RHEL4 operating system is based on kernel version 2.6.9, so this study used all of the version control data from kernel version 2.6.0 to 2.6.9, which was about approximately 15 months of development and maintenance. Table 11 provides the project statistics.

Table 11. Project statistics for the RHEL4 kernel

# of Files	LOC	Mean LOC	Files with Vulns.	Total Vulns.	Vulns. per File	% of Files with Vulns.
13,568	3,068,453	226	194	258	0.019	1.4%

[15] http://git.kernel.org/

5.3.2. Discriminative Power Test and Univariate Prediction Results

Table 12 provides the results of hypotheses tests for discriminate power and the results of the univariate prediction for individual metrics. Among the 28 metrics, 27 metrics showed a statistically significant difference between vulnerable and neutral files using the Welch's t-test after Bonferroni correction except for CommentDensity. The association directions for DNAvgDegree and DNAvgBetweenness did not agree with their hypotheses. We discuss the reason in Section 5.4.

In vulnerability prediction, AUC shows that most metrics predicted better than random except for CommentDensity and DNMinBetweenness. However, only NumDevs achieved the performance benchmark in over half of the 100 predictions for 10x10 cross-validation. RHEL4 had greater standard deviations in recall and PF than Mozilla Firefox.

5.3.3. Results from Multiple Regression Prediction Models

Table 13 provides the results for the multiple regression predictions by 10x10 cross-validation using logistic regression. The predictions were performed using the three variables selected by the information gain ranking method. AUC was over 0.8 for all models. The models using code churn, developer activity, and combined CCD metrics achieved the benchmark performance (recall of 0.7 and PF of 0.25) in over 50% of the 100 runs of cross-validation. However, none of the predictions using code complexity metrics satisfied the prediction criteria.

Table 12. Results of discriminative power test and univariate prediction for the RHEL4 kernel

		Discrim. Power			Predictability						
					Mean				Std		
Related Hypothesis	Metric	Welch-t	Association	H_D*	AUC	R*	PF*	Q_P*	R	PF	N(Q_P)*
	Code complexity										
$H_{IntraCode\_D}$	LOC	√	+	√	0.82	0.87	0.38	X	0.087	0.028	0
	NumDeclFunction	√	+	√	0.80	0.94	0.57	X	0.045	0.015	0
	LOCVarDecl	√	+	√	0.79	0.87	0.44	X	0.095	0.038	0
	NumLinePreprocessor	X	+	X	0.66	0.67	0.41	X	0.095	0.017	0
	SumEssential	√	+	√	0.83	0.93	0.53	X	0.056	0.029	0
	SumCyclomaticStrict	√	+	√	0.82	0.93	0.52	X	0.058	0.019	0
	MaxCyclomaticStrict	√	+	√	0.81	0.94	0.56	X	0.055	0.021	0
	SumMaxNesting	√	+	√	0.81	0.90	0.49	X	0.066	0.012	0
	MaxMaxNesting	√	+	√	0.79	0.90	0.49	X	0.066	0.012	0
$H_{Coupling\_D}$	SumFanIn	√	+	√	0.83	0.93	0.52	X	0.050	0.016	0
	SumFanOut	√	+	√	0.82	0.92	0.52	X	0.064	0.017	0
	MaxFanIn	√	+	√	0.84	0.93	0.54	X	0.050	0.013	0
	MaxFanOut	√	+	√	0.81	0.93	0.53	X	0.052	0.012	0
$H_{Comments\_D}$	CommentDensity	√	–	√	0.44	0.88	0.75	X	0.079	0.019	0
	Code Churn										
$H_{NumChanges\_D}$	NumChanges	√	+	√	0.88	0.83	0.25	√	0.088	0.029	27
$H_{ChurnAmount\_D}$	LinesChanged	√	+	√	0.79	0.83	0.39	X	0.091	0.020	0
	LinesNew	√	+	√	0.80	0.90	0.52	X	0.060	0.013	0
	Developer Activity										
$H_{DeveloperCentrality\_D}$	DNMinDegree	√	–	√	0.53	0.86	0.67	X	0.085	0.022	0
	DNAvgDegree	√	+	X	0.68	0.98	0.59	X	0.037	0.027	0
	DNMaxCloseness	√	+	√	0.73	0.98	0.71	X	0.073	0.069	0
	DNAvgCloseness	√	+	√	0.58	0.99	0.74	X	0.016	0.011	0
	DNMinBetweenness	√	–	√	0.49	0.69	0.63	X	0.118	0.030	0
	DNAvgBetweenness	√	+	X	0.70	0.97	0.57	X	0.037	0.015	0
$H_{DeveloperCluster\_D}$	DNMaxEdgeBetweenness	√	+	√	0.70	0.49	0.11	X	0.112	0.008	4
	DNAvgEdgeBetweenness	√	+	√	0.69	0.49	0.11	X	0.112	0.008	4
$H_{ContributionCentrality\_D}$	NumDevs	√	+	√	0.86	0.80	0.22	√	0.091	0.011	84
	CNCloseness	√	+	√	0.62	0.99	0.74	X	0.016	0.011	0
	CNBetweenness	√	+	√	0.84	0.92	0.41	X	0.062	0.013	0

*. H_D: Hypothesis test for discriminative power, R: Recall, PF: Probability of false alarm, Q_P: Question for predictability, N(QP): Number of predictions that provided recall of over 0.7 with PF of less than 0.25 from 80 predictions

Table 13. Results from multiple regression prediction models for the RHEL4 kernel

	Predictability							Inspection Costs				
	Mean				Std.			Costs			Cost Reduction	
	AUC	Recall	PF	Prec.	Recall	PF	N(HP)	PV	FI	LI	FIR	LIR
Code complexity	0.81	0.90	0.43	0.03	0.072	0.068	0	0.92	0.44	0.85	0.51	0.07
Code churn	0.87	0.82	0.24	0.05	0.092	0.019	59	0.85	0.25	0.58	0.70	0.32
Developer activity	0.86	0.81	0.23	0.05	0.092	0.029	76	0.84	0.24	0.58	0.70	0.31
Combined CCD	0.88	0.84	0.24	0.05	0.085	0.023	71	0.87	0.25	0.62	0.71	0.28

When we changed the classification threshold to 0.6 for the models using code complexity metrics, the averaged recall and PF were 0.77 and 0.29, respectively, and 15 of the 100 predictions satisfied the prediction criteria.

The reduction in file inspection compared to a random file selection was between 51% and 71%. The reduction in lines of code inspection was over 28% for code churn, developer activity, and combined CCD metrics and only 7% for code complexity metrics.

5.4 Summary of Two Case Studies and Discussion

Table 14 provides the summary of our hypotheses testing. The hypotheses for discriminative power were supported by at least 13 of the 14 code metrics and at least 25 of all the 28 metrics for both projects except for CommentDensity and DNAvgDegree for Firefox, and NumLinePreprocessor, DNAvgDegree, and DNAvgBetweenness for RHEL 4. Among these, NumLinePreprocessor and CommentDensity were not discriminative of neutral and vulnerable files.

Table 14. Summary of hypothesis testing for CCD metrics

	Hypotheses	**Firefox**	**RHEL 4**
$H_{CodeComplexity\_D}$	Vulnerable files are more complex than neutral files.	Yes for 13 of 14 metrics.	Yes for 13 of 14 metrics.
$H_{CodeChurn\_D}$	Vulnerable files have a higher code churn than neutral files.	Yes for all 3 metrics.	Yes for all 3 metrics.
$H_{Developer\_D}$	Vulnerable files are more likely to have been changed by poor developer activity than neutral files.	Yes for 10 of 11 metrics.	Yes for 9 of 11 metrics.
$Q_{Individual\_P}$	Can a model with individual CCD metric predict vulnerable files?*	5 of 28 metrics achieved the performance benchmark in over half of the 80 predictions.	1 of 28 metrics achieved the performance benchmark in over half of the 100 predictions.
$H_{CodeComplexity\_P}$	A model with a subset of code complexity metrics can predict vulnerable files.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 40 of 80 predictions.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 0 of 100 cross-validations.
$H_{CodeChurn\_P}$	A model with a subset of code churn metrics can predict vulnerable files.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 76 of 80 predictions.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 59 of 100 cross-validations.
$H_{Developer\_P}$	A model with a subset of developer metrics can predict vulnerable files.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 62 of 80 predictions.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 76 of 100 cross-validations.
H_{CCD_P}	A model with a subset of combined CCD metrics can predict vulnerable files.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 74 of 80 predictions.	- AUC was over 0.5 for all predictions. - Achieved the performance benchmark in 71 of 100 cross-validations.

*. The performance benchmark in this study is recall of over 0.7 with PF of less than 0.25.

DNAvgDegree (for both projects) and DNAvgBetweenness (for RHEL 4) disagreed with our hypotheses in the direction of association. While DNMinDegree was negatively correlated with vulnerabilities and supported our hypothesis, DNAvgDegree was positively correlated in both projects. This means that files are more likely to be vulnerable if they are changed by developers who work on many other files with other developers on average, but when all of the developers are central, the file is less likely to be vulnerable. The reason why the association direction of DNAvgBetweeenness does not match with its hypothesis was not clearly identified.

Overall, 80 predictions were performed for the eight releases (R4 – R11) of Firefox with ten repetitions to account for sampling bias and 100 predictions for RHEL 4 for 10x10 cross-validation. In the univariate predictions, five of the 28 metrics satisfied the criteria of recall 0.7 and PF 0.25 in 50% of the total predictions for Firefox and one of the 28 metrics for RHEL 4. Considering only the small number of metrics satisfied the prediction criteria in univariate prediction, relying on a single metric is a dangerous practice. In multiple regression predictions, all the models provided over 0.5 of AUC value showing that the CCD metrics have predictability. With more stringent criteria of recall 0.7 and PF 0.5, the models using code churn, developer activity, and combined CCD metrics satisfied the criteria in over 50% of the total predictions for both project. Even though the models using code complexity metrics satisfied the criteria of recall 0.7 and PF 0.5 in over 50% of the total predictions for

Firefox, none of the predictions were successful for RHEL 4. Considering this result together with the surprisingly low (even negative) inspection cost reduction seen in Figure 6, the effectiveness of code complexity metrics as indicators of vulnerabilities is weak for the code complexity metrics collected for this study compared with other process metrics. However, whether the lines of code is an effective cost measurement of effort depends on situation, requiring judicious interpretation of our results [63].

Precision from all the models for both projects was strikingly low (precision < 5.0) as a result of large numbers of false positives. This result is especially interesting because almost all of the individual metrics had statistically significant discriminative power according to Welch's t-test. This discrepancy can be explained from the boxplots in Figure 5 where the mean values in the individual metrics show clear difference, but the considerable numbers of neutral files are still in the expected range of vulnerable files leading to a large number of false positives. Organizations can improve precision by raising the threshold for binary classification to reduce false positives. However, recall can become lower in that case as recall and precision tend to trade-off each other. Whether a model that provides high recall and low precision is better than a model that provides high precision and low recall is arguable. An organization may prefer to detect many vulnerabilities because they have a large amount of security resources and security expertise. On the other hand, other

organizations may prefer a model that provides high precision that reduces the waste of effort even with low recall.

Preference for high recall and low precision requires some caution in terms of cost-effectiveness. From the results in Table 10 and Table 13, organizations are guided to inspect only less than 26% of files to find over around 80% of vulnerable files in most models. However, since the projects have a large amount of files, the number of files to inspect is still large. For example, Firefox release R4 has around 11,000 files and the FI from the model with combined CCD metrics was 0.24 identifying 2,640 files to inspect. If security inspection requires one person-day per file, over seven full years would be spent for one security engineer to inspect the 2,640 files. If the files to inspect are reduced to 10% of the predicted files (264 files) by further manual prioritization, the overall inspection would take essentially a year for one security engineer only to find a further reduced set of vulnerabilities. In that case, pursuing high precision and low recall might be a more cost-effective approach than pursing high recall and low precision. However, because the inspection time can greatly vary depending on the ability and the number of security engineers involved, organizations should use this illustration and our prediction results only to make an informed decision.

Among the metrics investigated in this study, history metrics such as code churn and developer activity metrics provided higher prediction performance than the code complexity. Therefore, historical development information is a favorable source for metrics and using

historical information is recommended whenever possible. However, the results from this study are limited to the metrics collected for this study. Other complexity metrics may provide better results.

A few releases of Firefox showed sudden changes in prediction performance. This result cannot be observed with cross-validation. Therefore, this study reveals the importance of next-release validation to validate metrics for vulnerability prediction whenever possible.

For both projects, PF and FI are almost equal. We believe the reason for this was that the percentage of files with vulnerabilities is very low (< 1.4%) and precision is also very low. The low percentage of vulnerable files means that the total number of files (TP+TN+FP+FN) is almost the same as the number of non-vulnerable files (FP+TN). The low precision means that the number of positive predictions (TP+FP) is very close to FP. Therefore, FI computed by (TP+FP)/(TP+TN+FP+FN) and PF computed by FP/(FP+TN) are almost equal. Knowing this fact provide us a useful hint to guess the number files to inspect when both the percentage of vulnerable files and precision are very low.

Interestingly, NumDevs was effective in the vulnerability prediction in our study while another study [78] observed that NumDevs did not improve the prediction performance significantly. The two major differences between the studies are (a) their study was closed-source and ours was open-source; and (b) they were predicting faults and not vulnerabilities. Further study on the difference in open- and closed-source projects and on the difference

between fault and vulnerability prediction may further improve our understanding on faults and vulnerabilities on various types of projects and better guide code inspection and testing efforts.

5.5 Threats to Validity

Since only known vulnerabilities can be collected and analyzed, this study does not account for latent (undiscovered) vulnerabilities. Additionally, only fixed vulnerabilities are publicly reported in detail by organizations to avoid the possible attacks from malicious users; unfixed vulnerabilities are usually not publicly available. However, considering the wide use of both projects, the currently-reported vulnerabilities seem not too limited to jeopardize our results.

This study combined every three releases and predicted vulnerabilities for next three releases for Mozilla Firefox. Using this study design, the predictions will be performed on the every third release. However, considering the short time periods between releases (one or two months), the code and process history information between the three releases within a combined release is relatively similar and those releases share many similar vulnerabilities. We combined the three releases to increase the percentage of vulnerabilities in each release because the percentage of vulnerabilities for the subject projects was too low to train the prediction models. In fact, once enough training data is accumulated during a few initial releases, one could predict vulnerabilities in actual releases rather than in combined releases.

For Mozilla Firefox, not all of the bug ids for vulnerability fixes were identified from the CVS log because comments in the CVS log are written by developers, but CVS does not enforce developers to enter bug ids relevant to a commit, which could lead to the lower prediction performance.

Actual security inspections and testing are not perfect, so our results are optimistic in predicting exactly how many vulnerable files will be found by security inspection and testing.

As with all empirical studies, the results in this study are limited to the two projects we studied. To generalize the observations from this study to other projects in various languages, sizes, domains, and development processes, further studies should be performed.

5.6 Summary

The goal of this study was to guide security inspection and testing by analyzing if code complexity metrics can indicate vulnerable files. Specifically, this study evaluated if code complexity metrics can discriminate between vulnerable and neutral files, and predict vulnerabilities. We compared code complexity and process metrics and our findings indicate that process metrics are better indicators of vulnerable code locations. At least 13 of the 14 code metrics and 25 of all 28 CCD metrics supported the hypotheses for discriminative power between vulnerable and neutral files for both projects. A few univariate models and the models using process metrics such as code churn, developer activity, and combined CCD

metrics predicted vulnerable files with high recall and low PF for both projects. However, the models with code complexity metrics alone provided the weakest prediction performance, indicating that metrics available from development history are stronger indicators of vulnerabilities than code complexity metrics we collected in this study.

The results from this study indicate that code churn, developer activity, and combined CCD metrics can potentially reduce the vulnerability inspection effort compared to a random selection of files. However, considering the large size of the two projects, the quantity of files and the lines of code to inspect or test based on the prediction results are still large. While a thorough inspection of every potentially vulnerable file is not always feasible, the results from this study show that using CCD metrics to predict files can provide valuable guidance to security inspection and testing efforts by reducing code to inspect or test.

The study contributed to the body of knowledge in software engineering by providing empirical evidence that CCD metrics are effective in discriminating and predicting vulnerable files and in reducing the number of files and the lines of code for inspection. The results from this study were statistically significant despite the presence of faults that could weaken the performance of a vulnerability prediction model.

While the results in this study show that predictive modeling can reduce the amount of code to inspect, much work needs to be done in applying models like ours to the security

inspection process. Examining the underlying causes behind the correlations found in this paper would assist even further in guiding security inspection and testing efforts.

CHAPTER 6

Fault Prediction Metrics as Indicators of Software Vulnerabilities

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 and design [45] and in changes to the code [29]. Hence, complexity metrics and code churn metrics have been used for fault prediction [4, 5, 10, 29, 36, 38, 51, 53, 55-58, 70, 81]. Additionally, problematic code areas in one release tend to be also problematic again in later releases [62]. These similarities between vulnerabilities and faults allow the possibility of the use of the traditional fault prediction metrics – complexity, code churn, and fault history metrics for vulnerability prediction. Chapter 4 showed the empirical evidence that complexity and code churn metrics are helpful to predict a high portion of vulnerable code locations but also include many false positives. At the same time, a frequently asked question about vulnerability prediction is whether organizations need a separate vulnerability prediction model when they already have a fault prediction model.

Although vulnerabilities and faults can have similar characteristics, only a small portion of faults are vulnerabilities. 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 can think like an attacker and have special

security skills in conducting vulnerability detection. Therefore, the characteristics of *discovered* vulnerabilities and faults may be different. Additionally the reported vulnerabilities are much fewer than the reported faults in many projects [2].

With these similarities and differences, can fault prediction models be used to predict vulnerabilities with equal to or better prediction performance than vulnerability prediction models? *The goal of this 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 code complexity, code churn, and fault history.*

To achieve this goal, this study performed an empirical case study on a widely-used open source project, the Mozilla Firefox web browser. Both fault prediction models and vulnerability prediction models have been built using the three types of traditional fault prediction metrics and the ability of both types of models to predict vulnerable files has been measured. This study 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.

6.1 Study Design

To obtain measurable results, the goal has been broken into three specific questions. Section 6.1.1 discusses the research questions. Section 6.1.2 provides the metrics used in this study. Section 6.1.3 provides the measurements of prediction performance specific to this study additional to the ones provided in Chapter 4. The logistic regression modeling technique discussed in Section 4.5 was used with 10x10 cross-validation.

Since there is no standard on prediction performance that indicates software quality is good enough, this study consider recall of 0.7 and precision of 0.7 as *desired* level of prediction performance, as has been reported in other fault and vulnerability prediction studies [30, 51, 59, 81]. Note that not many studies reported both recall and precision or provided both high recall and high precision at the same time. Menzies et al. provided recall (0.7 on average) and PF, but not precision in their fault prediction study [51]. In a Neuhaus et al's study, precision was 0.7, but recall was 0.45 in their vulnerability study [59]. Only a Zimmermann and Nagappan's study provided recall of 0.7 and precision of 0.7 in their failure prediction study [81]. In a Guo's fault prediction study, among the 56 predictions using various classification techniques on five projects, only 14 predictions provided over recall of 0.7, but precision was not reported [30]. Note that all these studies have been performed in different contexts; the studies have been performed using different subject projects, 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 which is why this study requires its own baseline.

6.1.1. Research Questions

Q1*: Can fault prediction models using traditional fault prediction metrics predict faults with desirable recall and precision of 0.7 or more?*

This study answers this question to provide a baseline of comparison with the performance of vulnerability prediction. A fault prediction model is built by training it with the fault status of a file as a dependent variable and the traditional fault prediction metrics as independent variables. Then, the recall and precision of the predicted faulty files is measured. This *fault* prediction using a *fault* prediction model is named as *FF prediction* in this study.

Figure 7 shows the relationship between metrics, the model, and the purpose of prediction in FF prediction.

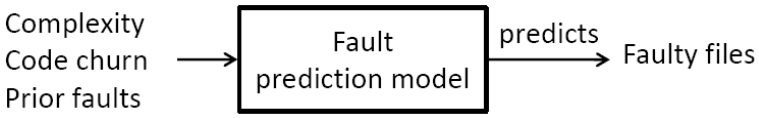

Figure 7. Fault prediction using a fault prediction model (FF prediction)

Q2*: Can fault prediction models predict vulnerabilities with desirable recall and precision of 0.7 or more?*

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, this study builds the same fault prediction model as the one used to answer Q1 and counts 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 and precision of those predicted vulnerable files. The *vulnerability* prediction using a *fault* prediction model is named as *VF prediction*.

Figure 8 shows the relationship between metrics, the model, and the purpose of prediction in VF prediction.

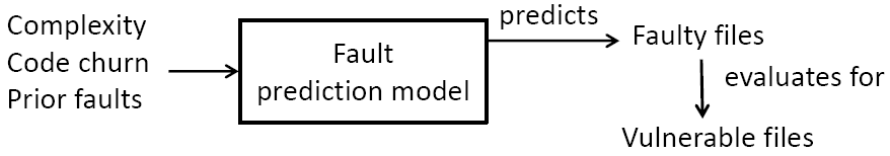

Figure 8. Vulnerability prediction using a fault prediction model (VF prediction)

***Q3**: Can vulnerability prediction models using traditional fault prediction metrics predict vulnerabilities with desirable recall and precision of 0.7 or more?*

To answer this question, this study builds 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, the recall and precision of the predicted

vulnerable files are measured. The *vulnerability* prediction using a *vulnerability* prediction model is named as *VV prediction*.

Figure 9 shows the relationship between metrics, the model, and the purpose of prediction in VV prediction.

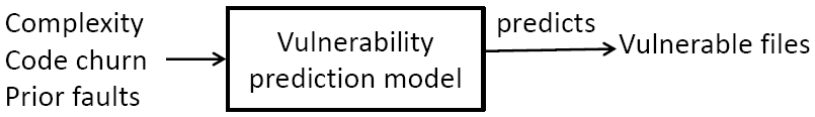

Figure 9. Vulnerability prediction using a vulnerability prediction model (VV prediction)

6.1.2. Traditional Fault Prediction Metrics

This study uses three types of traditional fault prediction metrics in this study: complexity, code churn, and past fault history metrics. Table 15 provides the definitions of the code churn and fault history metrics. For complexity metrics, this study uses the code complexity metrics defined in Section 2.2.1.

Table 15. Definitions of code complexity, code churn, and past fault history metrics

Metric	Definition
Code complexity metrics	
	Refer to the code complexity metrics in Section 2.2.1.
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.

6.1.3. Evaluation Criteria for Prediction Performance

To clarify recall and precision from the three types of predictions (FF, VF, and VV), recall, precision, FI, and LI that have defined in Chapter 4 are redefined in this study 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.

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

$Precision_{VF}$, and $precision_{VV}$ are the ratios of the correctly classified vulnerable files to the predicted vulnerable files using a fault prediction model and a vulnerability prediction model, respectively

FI_F and FI_V are the ratios of files to be inspected to the total files as a result of fault prediction and vulnerability prediction, respectively. LI_F and LI_V are the ratios of lines of code to be inspected to the total lines of code as a result of fault prediction and vulnerability prediction, respectively.

6.2 A Case Study: Mozilla Firefox

Mozilla Firefox is a widely-used open source web browser developed and evolved for several years. Firefox is written in C/C++ and consists of over 10,000 files and over two million lines of source code. Different from Chapter 4 that has used multiple releases of Firefox and performed next-release validation, this study combines the data from Firefox 2.0 and their minor releases and performs cross-validation. The vulnerability data collection is also different from Chapter 4. In Chapter 4, the vulnerable files were identified by analyzing CVS logs. In this study, the vulnerable files are identified by analyzing the patch files attached in the bug reports that contain vulnerability reports. The reason was because fault information available only by the latter method and to use the same data collection method

for both faults and vulnerabilities. Details of data collection are explained in the following subsection.

6.2.1. Data Collection

This subsection describes the method used in this study to collect faults and vulnerabilities in a file and to collect complexity, code churn, and fault history metrics. This section also provides descriptive statistics of the metrics. This study uses the faults reported since the release of Firefox 2.0 and before the release of Firefox 3.0 from the Mozilla Bugzilla 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. The bug reports include both faults and enhancements. However, we are interested only in faults. The bug reports do not have explicit information that we can automatically distinguish between faults and enhancements. To facilitate the classification of enhancements and faults, this study used automated text classification. The details of the automated text classification are described in Section 6.2.2. Only the bug reports that were classified as faults by the

automated text classification and whose resolution method and whose status are “FIXED and RESOLVED” or “FIXED and VERIFIED” are used in this study. The number of the bug reports that have bug patches for a file is used as a surrogate measure of the number of faults in a file in this study. 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 “FIXED and RESOLVED” or “FIXED and VERIFIED” and had patches written in C/C++.

This study collected vulnerabilities reported for Firefox 2.0 and its minor releases from the Mozilla Foundation Security Advisories (MFSAs)[16]. 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. This study counts the number of the bug reports that have bug patches for a file from those bug reports as a surrogate measure of the number of vulnerabilities in the file.

Firefox 2.0 consists of 11,051 files. Among them, 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 (13% of the faulty files and 81% of the vulnerable files) as in Figure 10.

---

[16] http://www.mozilla.org/security/known-vulnerabilities/

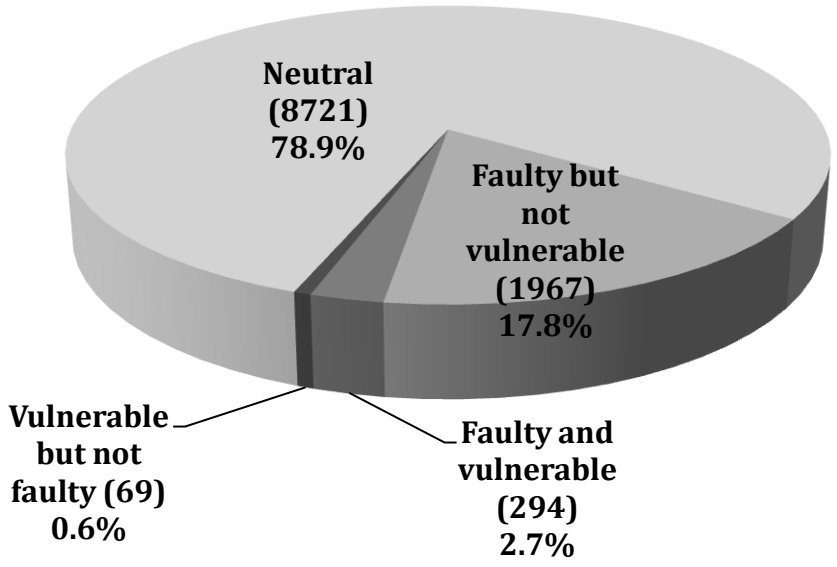

Figure 10. Distribution of faulty files and vulnerable files in Firefox 2.0

To collect the code churn metrics, this study searched the files modified for the period of 12 months before the release of Firefox 2.0 from the CVS source code repository. For the measurement of faults, vulnerabilities, and code churn, this study 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, this study collected faults from Firefox 1.0 and its minor releases using the same procedure used to collect faults in Firefox 2.0.

Although a large portion of vulnerable files also have faults (81% of the vulnerable files), those vulnerable files corresponds to only 13% of the total faulty files. Therefore, the distribution of measures for the metrics of faulty files and vulnerable files may differ. Table 16 shows the means and medians of five representative metrics per file. In Table 16, 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 are around twice as complex as faulty files, have around twice as large and frequent changes as faulty files, and also have around twice as more past faults as faulty files in Firefox 2.0. Other remaining metrics also show similar tendency. These statistics show that predicting faulty files and vulnerable files using the traditional fault prediction metrics is a feasible approach.

Table 16. Comparison of mean and median values of metrics for Firefox 2.0

	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

6.2.2. Automated Text Classification of Faults and Enhancements

This study performed automated text classification to classify bug reports as faults or enhancements. 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. [3] has also performed automated text classification of the bug reports for faults and enhancements of the Mozilla project. Their logistic regression model provided recall 0.76 and precision 0.82 in the classification of faults and enhancements. This study performed a similar approach to their study. However, this study used the bug reports that have patches written in C/C++ for Firefox 2.0. The basic method of text classification is similar to the method for vulnerability prediction described in Section 3.3 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. In this study, recall was 0.88 and precision was 0.85 for the test data set. The procedure of bug report classification is described below:

Step 1. Create a training data set. To create a training data set, this research manually classified 600 sample bug reports as faults or enhancements using the titles and bug descriptions as texts. A few examples of the terms that frequently appear in the bug reports for faults are *fail, failure, bug, problem, error, crash, shutdown, regression, incorrect,* and *memory corruption*. Examples of the terms frequently appear in the bug reports for enhancements include *add, implement, make, useless, redundant, remove, need, future,* and *improve*.

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

Step 2.1. Preprocess the texts. In this step, this study first removed punctuations and converted all words to lower case. Then, this study applied the standard Porter stemmer [66] that removes suffixes from various forms of verbs (-ed, -ing, etc.) or plural forms of words. Finally this study created a term frequency vector that included frequency of words in each bug report.

Step 2.2. Split the term frequency vector into training and testing sets. In this step, this study 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.

Step 2.3. Build classification models and perform cross-validation. This study chose 100 words as independent variables using the InfoGain variable selection method and built a logistic regression model using the training data set. Then this study classified bug reports in the test data set using the model. This study repeated Step 2.2 and Step 2.3 100 times for 10x10 cross-validation. The logistic regression model provided recall of 0.88 and precision of 0.85. Although it is not perfect, this performance seems to be enough for our purpose of fault classification. Therefore, Step 3 is performed as below:

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, this study used the whole set of manually classified sample bug reports from Step 1 to train the model. Then this study classified the remaining 6,365 bug reports using the trained model.

Weka 3.7 was used for the preprocessing and classification.

6.3 Prediction Results

This subsection answers the three research questions presented in Section 3.

Q1*: Can fault prediction models using traditional fault prediction metrics predict faults with desirable recall and precision of 0.7 or more? (FF prediction)*

$Recall_{FF}$, $precision_{FF}$, FI_F, and LI_F were measured to answer this question. As shown in Figure 11. Fault prediction results using fault prediction models (FF prediction), the fault prediction model predicted 74% of the total faulty files ($recall_{FF}$ of 0.74) 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 ($precision_{FF}$ of 0.47). 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 desirable recall (over 0.7), but low precision (below 0.7). 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.

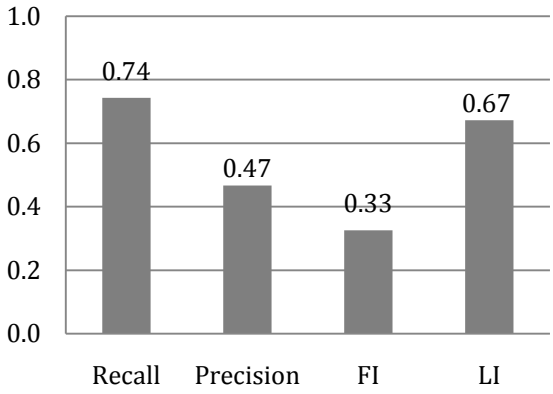

Figure 11. Fault prediction results using fault prediction models (FF prediction)

***Q2**: Can fault prediction models predict vulnerabilities with desirable recall and precision of 0.7 or more? (VF prediction)*

$Recall_{VF}$ and $precision_{VF}$ were measured to answer this question. Because both Q1 and Q2 use the same fault prediction model, the amount of inspection (FI_F and LI_F) is the same as the results from Q1. As shown in Figure 12. Vulnerability prediction results using fault prediction models (VF prediction), the fault prediction model predicted 92% of the total vulnerable files ($recall_{VF}$ of 0.92) 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$_{VF}$ of 0.09). 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 (0.92), but much lower precision (0.09) than our criterion and the precision from the FF prediction that has been preformed for Q1. 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.

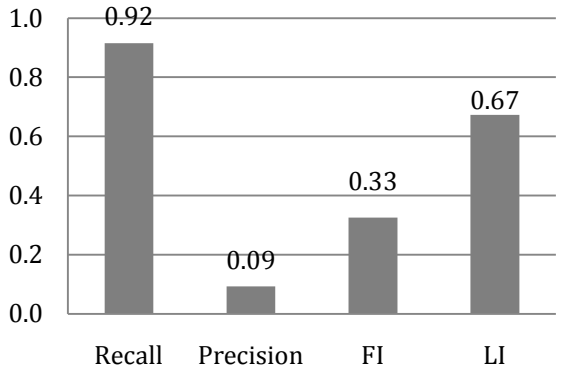

Figure 12. Vulnerability prediction results using fault prediction models (VF prediction)

***Q3**: Can vulnerability prediction models using traditional fault prediction metrics predict vulnerabilities with desirable recall and precision of 0.7 or more? (VV prediction)*

$Recall_{VV}$, $precision_{VV}$, FI_V, and LI_V were measured to answer this question. The vulnerability prediction model correctly predicted 84% of the total vulnerable files ($recall_{VV}$ of 0.84) 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 ($precision_{VV}$ of 0.12). 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. Figure 13 presents the results of the VV prediction and compares them with the results from the VF prediction in Q2.

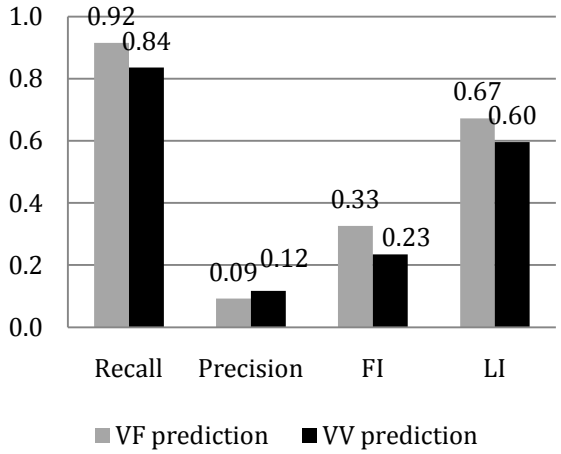

Figure 13. Comparison of the results of VF prediction and VV prediction

Since the VV prediction provided worse performance in recall and better performance in precision than the VF prediction, the results from the two predictions may become similar if we lower the threshold of classification for VV prediction to generate more TPs and FPs and to result in higher recall and lower precision than when threshold 0.5 is used. After the

classification threshold of the VV prediction has changed to 0.3, all the performance measures between the two types of models became very close (Figure 14). The recall and precision for the VF prediction were 0.92 and 0.09, respectively. The recall and precision for the VV prediction became 0.90 and 0.09, respectively. This result suggests that fault prediction models can be used as a substitute for vulnerability prediction models when traditional fault prediction metrics are used for Firefox 2.0.

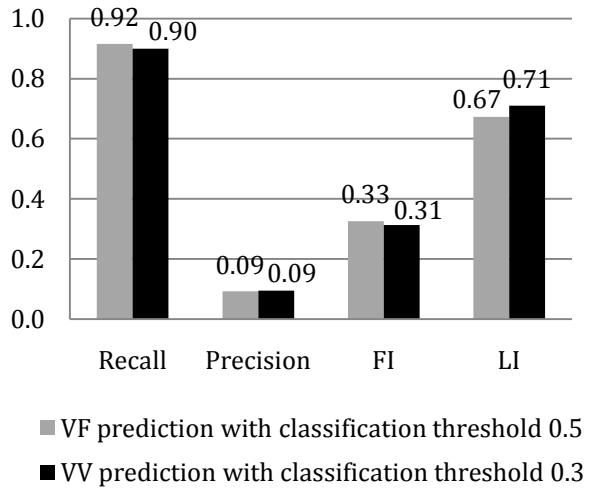

Figure 14. Comparison of the results of VF prediction and VV prediction with adjusted classification threshold

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 when the three types of traditional metrics are used.

6.4 Analysis of the "Needle Effect"

The fact that only small percent of files are vulnerable (3% of the total files and 13% of the faulty files in Fig. 4) 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 seen in Q1 through Q3 in Section 4.3. At the same time, the difference in the measurements of the three types of metrics between faulty files and vulnerable files (see Table 3) can lead to the difference in prediction performance. Figure 15 puts the results from Q1 and Q3 together. The VV prediction provides 18% higher recall and 38% lower precision than the FF prediction.

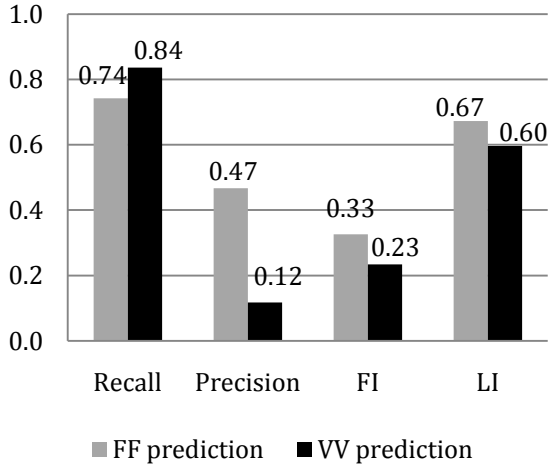

Figure 15. Comparison of the results of FF prediction and VV prediction

This study 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. The hypothesis is that if the difference in the prediction performance mainly comes from the difference in the numbers of the reported faults and vulnerabilities, the performance of the FF prediction and the VV prediction will be similar if we adjust the number of faulty files. To test this hypothesis, this study randomly selected a subset of faulty files and marked them as neutral pretending 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 three times as many, twice as many, and the same number of faulty files as the number of vulnerable files, after the remaining faulty files are marked as neutral. The numbers of faulty files and vulnerable files used for the models are also presented in Figure 16.

In Figure 16, $recall_{FF}$, 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, we can observe that there are noticeable differences in recall, FI, and LI between the four FF predictions and the VV prediction. These differences may have been caused by

the difference in the distribution of measures of the metrics between faulty and vulnerable files as we have seen in Table 3. 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 (FF3 prediction vs. VV prediction). 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.

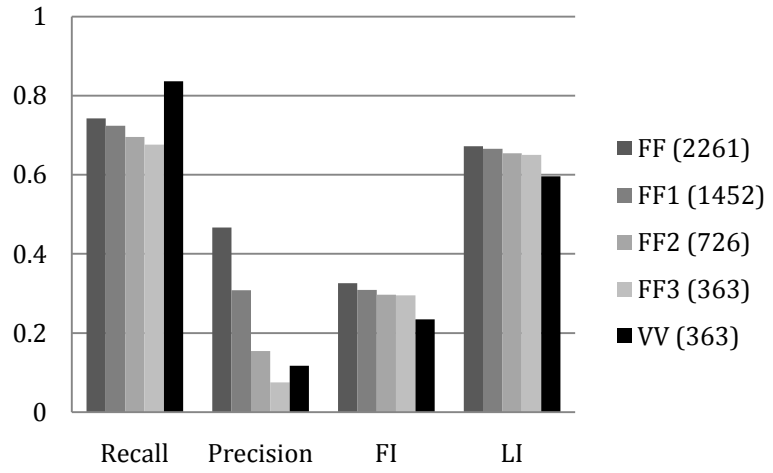

Figure 16. Effects of the number of faulty files and vulnerable files

The radical change in precision depending on the percentage of the minor class (faulty files or vulnerable files in our study) was also observed in Menzies et al.'s study and they claimed that precision is an instable performance measurement for the highly unbalanced data set [50]. For this reason, they preferred PF to precision as a measure of the

degree of false positives. This study also observed low precision and the instability of precision from both fault prediction and vulnerability prediction. However, this study chose to use precision as a measure of prediction performance because development teams would act on the files predicted as faulty or vulnerable rather 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.

In summary, the difference in the distribution of measures of complexity, code churn, and fault history metrics between faulty files and vulnerable files brings 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.

6.5 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, LOC, and LOCVarDecl.

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 the small number of past faults. However, the precision of the vulnerability predictions was much lower (0.09 after adjusting the classification threshold) than the precision of the fault prediction (0.47) 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 [2] 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 16%. Although the number of files to be inspected is reduced by over 64% compared with random file selection in both the VF prediction and the VV prediction, 33% of 11,051 files are still many files (3,647) for inspection. 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 [32, 42].

6.6 Threats to Validity

This study 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, this study 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.

This study 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.

For Firefox, the separation between faults and enhancements is not clear. To overcome this limitation, this study performed automated text classification. The classification of faults and enhancements for Firefox using the automated text classification technique is not perfect and has room to be improved. However, recall of 0.88 and precision of 0.85 for the training data set seem to be reasonably high.

6.7 Summary

This study 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. This study examined the effectiveness of those metrics for vulnerability prediction on the Mozilla Firefox 2.0 web browser. In this study, the fault prediction model and the vulnerability prediction model provided similar prediction performance for vulnerability prediction. Both the fault prediction model and the vulnerability prediction model predicted vulnerabilities with high recall of over 0.9 and effectively reduced the number of files to be

inspected. However, precision was very low (0.09) leading to a waste of resources in security inspection and testing. The reason of the low precision was primarily because that the number of reported vulnerabilities was small.

Although this study shows that fault prediction models provide similar ability to vulnerability prediction models in the ability of vulnerability prediction based upon traditional metrics, there are still reasons to create separate vulnerability prediction models. First, vulnerability data are easier to collect than fault data depending on projects. As we have seen in this study, the separation between faults and enhancements was not clear for Firefox, while vulnerability data provide clear link to vulnerable files. Because of increasing attention to security problems, more open source projects are providing vulnerability data at a level that can be traced to the vulnerable code locations than before. Additionally, other projects and other metrics types may have different distributions for faulty files and vulnerable files. Therefore, if vulnerability data are available, creating a vulnerability prediction model is a more reliable way to predict vulnerable code locations than to use a fault prediction model.

CHAPTER 7

Code, OO Design, Network, and Execution Complexity Metrics

This chapter investigates OO design complexity, network complexity, and execution complexity metrics explained in Section 2.2 as indicators of vulnerabilities. Although code metrics have been studied in Chapter 4, this chapter again tests the ability of code metrics as vulnerability indicators for the purpose of comparison with OO design complexity, network complexity, and execution complexity metrics. In addition to the projects that have been used in previous chapters (Firefox and RHEL), this chapter uses Wireshark and Tomcat as subject projects.

Different from previous chapters that only file level analyses were performed, all tests are performed at file, class, and component level in this chapter. Although analysis at a finer entity level such as file or class level is more useful for software engineers to find vulnerable code locations than analysis at a coarser entity level such as component level, analysis at different granularities can improve our understanding on the relationships between vulnerabilities and complexity metrics because analyzing projects at multiple levels essentially has similar effects to analyzing different projects with different size of entities and with different distribution of vulnerabilities.

7.1 Case Study Projects

The four types of complexity metrics have been collected from four open source project: the Mozilla Firefox 3.0 web browser; the Red Hat Enterprise Linux 4 (RHEL4) operating system kernel; the Wireshark 1.2.0 network protocol analyzer; and the Apache the Tomcat 6.0 servlet container. This study collected data from one of the recent releases of each project, but have reported vulnerabilities cumulated at least for one year at the time of data collection. Additional to the projects that have used in precious chapters, Wireshark and Tomcat are used for this study.

Wireshark is a popular open source network protocol analyzer that is used to find network issues, to locate bottlenecks, and to detect network intrusion. Wireshark captures every packet that arrives to the network interface, including unitcast, multicast, and broadcast packets, on which Wireshark is running. Among those packets, Wireshark can analyze over 1000 network protocols. Vulnerabilities in the protocol analyzer can allow security breaches when the packets are manipulated to include malicious data to exploit vulnerabilities in Wireshark. Wrireshark is written in C and consists of approximately 1000 files and over 1600 KLOC.

Tomcat is a widely used open source servlet container. A servlet is a Java class that dynamically generates HTML pages responding to HTTP requests. A servlet container maintains the lifecycle of servlets in a Web server such as locating a particular sevlet for the

requested URL. Tomcat 6.0 is written in Java and consists of 1,324 classes and over 150 KLOC.

Table 17 shows the summary of the four projects at different levels of granularities: file, component, and class. Class level information is available only for Firefox and Tomcat that were written in C/C++ and Java, respectively. A component in this study is a collection of files that are located within the same directory. Because the directory structures of the projects were different and some of the network metrics (Flow_Betweenness and RW_Betweenness) were not collectable for the large number of entities, this study uses different granularity of component. A Firefox component and a RHEL4 component are the set of files under the highest level of directories in the projects. These components are called *high level components* in this study. A Wireshark component and a Tomcat component are the set of files under the lowest level of directories in the projects. These components are called *low level components* in this study. As shown in Table 17, the percentages of vulnerable files are between 1.4% and 3.8%. The percentages of high level vulnerable

Table 17. Project statistics Firefox 3.0, RHEL4, Wireshark 1.2.0, and Tomcat 6.0

	Component Level	**LOC**	**# of files**	**% of vuln. files**	**# of comp.**	**% of vuln. comp.**	**# of classes**	**% of vuln. classes**
Firefox	High	1854877	7895	3.8%	42	54.8%	4840	3.7%
RHEL	High	3067750	13563	1.4%	16	68.8%		
Wireshark	Low	1648589	2330	7.8%	113	13.3%		
Tomcat	Low	153978	1076	3.2%	121	14.0%	1324	2.6%

components are over 50%, and the percentages of low level vulnerable components are below 14%. The percentages of vulnerable classes are close to the percentages of vulnerable files.

Note that the metrics collected are different depending on the entity type. Code complexity and network complexity metrics were available for all the four projects. However, OO design complexity metrics were available only for Firefox and Tomcat that were written in C/C++ and Java languages, respectively, at class level and component level. Execution complexity metrics were collected only for Firefox and Wireshark. NumLinePreprocessor were not available for Tomcat because the Java language does not perform preprocessing unlike C/C++. Flow_Betweenness and RW_Betweenness were not computable for a large number of entities by the tools this study used and were collected only at component level.

7.2 Execution Complexity Metrics

This subsection explains how execution complexity metrics have been collected in this study. The definition of the four types of complexity metrics and the metrics collection methods for other metrics are provided in Section 2.2. Execution complexity metrics were collected only from Firefox and Wireshark for this study.

7.2.1. Firefox

As discussed in Chapter 1, the execution complexity metrics are intended to capture the software's runtime complexity by usage patterns by a normal user. For this purpose, for Firefox, execution complexity metrics have been collected from the execution profile collected at runtime by performing the following usual tasks this researcher performs frequently:

- Enter a search query from www.google.com and browse retrieved results.
- Login to www.gmail.com email account and read an email.
- Login to www.facebook.com and browse the posted messages.
- Login to a bank account and check the transactions in the current month.

This study excluded the execution profile during initialization of the web browser assuming that initialization modules have less vulnerabilities, if any, than other modules.

7.2.2. Wireshark

For Wireshark, the execution complexity metrics have been collected by performing the following usual tasks that users of Wireshark perform:

- Capture the network packets while performing the four operations for Firefox provided in Section 7.2.1.
- Browse the detailed information of a packet from the captured packet list.

As with Firefox, execution profile during initialization of the GUI interface has been excluded. Most frequently captured protocols are TCP, HTTP, and DNS protocols. Figure 17 shows the screen of Wireshark while the packets are being captured.

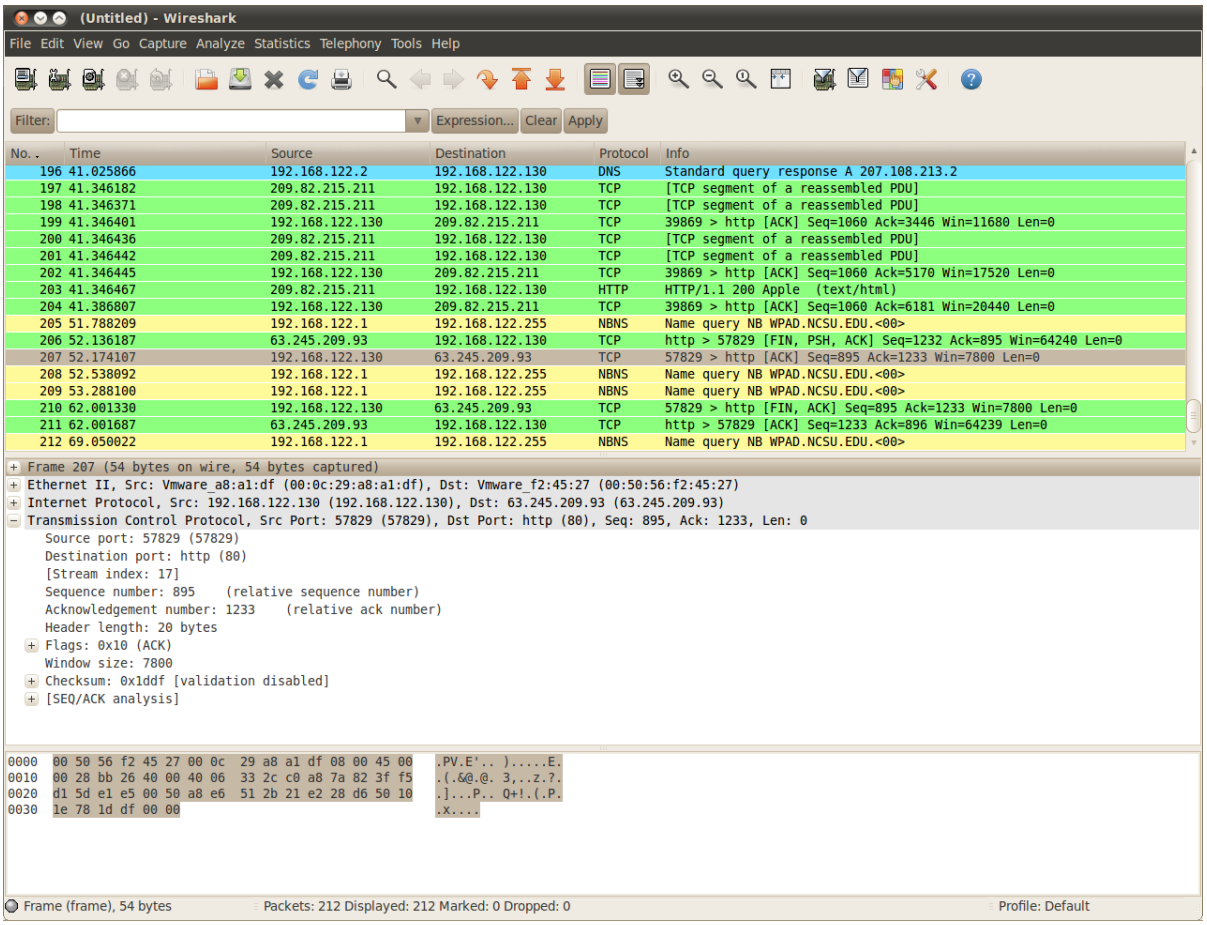

Figure 17. Screen shot of Wireshark 1.2.0

7.3 File Level Analysis

This section first examines the correlations between metrics and vulnerabilities, then, provides the results of discriminative power analysis and prediction of vulnerable files.

7.3.1. Correlation Analysis

Spearman rank correlation coefficient (ρ) and Pearson correlation coefficient (r) are the statistics often used to measure the strength of correlations between two variables. Pearson correlation coefficient assumes normal distribution of data, while Spearman rank correlation coefficient is a non-parametric test that does not assume any distribution. Spearman rank correlation is performed on the ranks of the values without considering the magnitudes of the values. Therefore, the Spearman rank correlation coefficient is not sensitive to outliers. Because the data used in this study was not normally distributed, this study uses Spearman rank correlation coefficient.

When the value of a Spearman rank correlation is negative, the two data sets are negatively associated (when the values of one data set go up, the values of the other data set go down); and the positive value of a Spearman rank correlation indicates that two data sets are positively associated. The magnitudes of the values tell the strengths of correlations.

Table 18 shows the Spearman correlation coefficients between the measures of metrics and the number of vulnerabilities. LOC has constantly been known to be a good indicator of faults [19, 39, 62]. If large code also indicates vulnerabilities and if a metric is

highly correlated with LOC, the metric may measure the same aspects as LOC. Therefore, we are also interested in the correlations between the measures of metrics and the number of vulnerabilities after removing the size effect. For this purpose, Table 18 also includes the partial correlations between the measures of metrics and the number of vulnerabilities after controlling for LOC. In Table 18, the correlations that are statistically significant at 0.05 significance level presented with '*'. The top three highest correlations in terms of absolute values in each project are in bold face.

In Table 18, overall the correlations between complexity metrics and vulnerabilities are very low (less than 0.3), but most of the correlations were statistically significant indicating the correlations are not by chance. In Table 18, OutDegree is most frequently included in the list of top three highest correlations across projects. Interestingly, the three execution complexity metrics provide the highest correlations for Firefox. The partial correlations after removing the size effect are also high for the execution complexity metrics from Firefox, showing the possibility that these metrics are strong candidate as good predictors of vulnerable code locations without the issue of multicollinearity with other size-based metrics. Execution complexity metrics, however, show lower correlations than other metrics for Wireshark.

Table 18. Correlations at file level

	Spearman Correlations				Partial Correlations Controlled for LOC			
	Firefox	**RHEL**	**Wireshark**	**Tomcat**	**Firefox**	**RHEL**	**Wireshark**	**Tomcat**
Code complexity metrics								
LOC	0.170*	0.131*	0.167*	0.217*	1	1	1	1
LOCVarDecl	0.174*	0.120*	0.140*	**0.218***	0.043*	-0.010*	-0.095*	0.037
NumFunctions	0.159*	0.131*	0.177*	0.150*	0.036*	0.038*	0.064*	-0.032
NumLinePreprocessor	0.183*	0.064*	0.192*		0.125*	0.032*	**0.118***	
SumEssential	0.154*	**0.143***	0.174*	0.183*	0.023*	**0.061**	0.055*	0.004
SumCyclomaticStrict	0.151*	**0.139***	0.169*	0.204*	0.016	0.053	0.040	0.016
MaxCyclomaticStrict	0.132*	0.134*	0.157*	0.205*	-0.003	0.046*	0.025	0.043
SumMaxNesting	0.111*	0.127*	0.172*	**0.218***	-0.010	0.035*	0.055*	0.058
MaxMaxNesting	0.106*	0.123*	0.143*	0.200*	-0.013	0.033*	0.011	0.039
SumFanIn	0.109*	0.132*	0.169*	0.184*	-0.016	0.043*	0.044*	0.011
MaxFanIn	0.096*	0.135*	0.154*	0.171*	-0.023*	0.051	0.028	0.015
SumFanOut	0.108*	0.128*	0.192*	0.217*	-0.021	0.034*	0.096*	0.045
MaxFanOut	0.099*	0.125*	**0.213***	0.217*	-0.025*	0.033*	**0.134***	0.066*
CommentDensity	-0.132*	-0.023*	-0.112*	-0.111*	-0.037*	-0.040*	-0.030	0.048
Dependency network complexity metrics								
InDegree	0.075*	0.086*	0.014	-0.021	0.063*	**0.061**	0.032	-0.009
InDegree_w	0.089*	0.064*	-0.017	0.020	0.062*	0.040*	-0.011	0.004
OutDegree	0.176*	**0.143***	**0.231***	**0.219***	0.076*	**0.063**	**0.168***	**0.118***
OutDegree_w	0.166*	0.135*	0.168*	**0.227***	0.045*	0.046*	0.035	**0.110***
SP_Betweenness	0.132*	0.119*	**0.194***	0.117*	0.068*	**0.077**	0.116*	0.052
EvCent	0.185*	0.128*	0.110*	0.211*	0.131*	0.044*	-0.022	**0.133***
Execution complexity metrics								
NumCalls	**0.234***		0.093*		**0.194***		0.073*	
InclusiveExeTme	**0.238***		0.092*		**0.196***		0.072*	
ExclusiveExeTime	**0.234***		0.093*		**0.194***		0.073*	

7.3.2. Discriminative Power

Table 19 shows the results of Welch's t-test and *mean ratio* (ratios of the means of metric values from vulnerable components to the means of metric values from neutral components for each metric) at the file level and the class level.

In Table 19, over 70% of the metrics in Firefox, RHEL, and Tomcat show statistically significant discriminative power. However, for Wireshark, only 43% of the metrics show

statistically significant discriminative power. Note that the means of metrics values for vulnerable files are higher than the ones for neutral files in most metrics regardless of statistical significance. A few exceptions are CommentDensity for all projects, as we already expected in Section 2.2, and InDegree and InDegree_w for Tomcat.

In Table 19, metrics that represent incoming information flow (SumFanIn, MaxFanIn, InDegree, InDegree_w) tend to not be discriminative of vulnerable files, while metrics that represent outgoing information flow (SumFanOut, MaxFanOut, OutDegree, and OutDegree_w) are discriminative of vulnerable files. This observation is similar to the findings by Briand et al. [10] in their study to investigate OO design metrics as indicators of faults.

To summarize, there are subsets of complexity metrics that are discriminative of vulnerable files in each set of code complexity and network complexity metrics. Although none of the execution complexity metrics used in this study show significant discriminative power, the means of execution complexity metrics from vulnerable files are at least four times higher than the means of execution complexity metrics from neutral files.

Table 19. Discriminative power at file level

	Firefox		RHEL		Wireshark		Tomcat	
	Welch's t-test	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**
Code complexity metrics								
LOC	√	4.3	√	3.1	X	1.8	√	4.2
LOCVarDecl	√	3.8	√	2.1	X	1.5	√	3.6
NumFunctions	√	4.3	√	3.1	X	1.8	X	4.2
NumLinePreprocessor	√	3.4	X	1.3	√	1.7		
SumEssential	√	4.8	√	4.4	X	1.9	√	3.5
SumCyclomaticStrict	√	4.9	√	4.0	√	2.0	√	4.0
MaxCyclomaticStrict	√	3.1	√	3.2	√	1.7	√	3.2
SumMaxNesting	√	2.5	√	3.3	√	2.1	√	4.3
MaxMaxNesting	√	1.9	√	2.4	√	1.6	√	2.3
SumFanIn	X	1.8	√	3.3	X	1.8	X	3.1
MaxFanIn	X	1.1	√	2.6	X	2.5	√	1.8
SumFanOut	√	2.2	√	3.2	√	2.1	√	4.5
MaxFanOut	√	1.8	√	2.8	√	2.3	√	3.4
CommentDensity	√	0.3	√	0.3	X	0.5	√	0.3
Dependency network complexity metrics								
InDegree	√	2.3	X	3.0	X	2.2	X	0.8
InDegree_w	√	2.3	X	1.8	X	1.7	X	0.9
OutDegree	√	3.2	√	3.1	√	2.3	√	3.0
OutDegree_w	√	5.2	√	3.3	√	2.1	√	5.2
SP_Betweenness	√	7.8	X	12.8	X	2.7	X	9.7
EvCent	√	3.2	√	2.6	√	1.5	√	3.3
Execution complexity metrics								
NumCalls	√	7.0			X	8.2		
InclusiveExeTime	√	26.7			X	4.4		
ExclusiveExeTime	X	7.4			X	8.6		

7.3.3. Predictability

To analyze predictability of the four types of complexity metrics, 10x10 cross-validation was performed. For each projects, a prediction model using each type of metrics and a prediction model using all types of metrics were built. Table 20 shows recall, precision,

PF, AUC, FIR, and LIR averaged over 100 runs of 10x10 cross-validation with standard deviation of AUC. The column for "AUC ≠ 0.5" shows the t-test results after Bonferroni correction of the null hypothesis that AUC is equal to 0.5. The column for "AUC ≠ 0.785" shows the test results of the null hypothesis that AUC is equal to 0.785". In both columns, "+" indicates AUC is significantly greater than the tested value. "-" indicates AUC is significantly lower than the tested value. "X" indicates there is no significant evidence to reject the null hypothesis.

In Table 20, recall is high in general (0.67 to 0.91); precision is very low (0.08 and 0.12); PF is between 0.23 and 0.52; AUC is between 0.70 and 0.87 except for the exceptionally different results for execution complexity metrics from Wireshark. In all predictions, AUC is significantly higher than 0.5 indicating the prediction results are better than random classification of files. AUC is also significantly higher than the performance benchmark (AUC of 0.785) for RHEL and Tomcat for the models with all metric types, but significantly lower than the performance benchmark for the model using execution metrics of Firefox and all the models for Wireshark. The models with execution metrics provided lowest performance in terms of AUC compared to the models with other metric types for both Firefox and Wireshrk. FIR was high (between 0.31 and 0.68) across projects, but LIR was low (between -0.17 and 0.37) except for the exception high LIR for the models with execution complexity metrics for Firefox and Wireshark. Note that for Firefox, only 24% of

the total files have been executed and 11% of the executed files were vulnerable. For Wireshark, only 4% of total files have been executed and 19% of the executed files were vulnerable. This low coverage of files during code execution explains the low prediction performance, and the relatively high percentage of vulnerable files within the executed files explains relatively high precision, FIR, and LIR compared to the results from other types of prediction models. Therefore, execution complexity metrics are better indicators of vulnerabilities at least for the executed files than other metrics.

Table 20. Predictability at file level

	Recall	**Precision**	**PF**	**AUC**	**Std (AUC)**	**AUC ≠ 0.5**	**AUC ≠ 0.785**	**FIR**	**LIR**
Firefox									
all	0.67	0.11	0.23	0.79	0.054	+	X	0.63	0.37
code	0.71	0.09	0.31	0.77	0.049	+	X	0.54	0.04
dependency	0.79	0.08	0.38	0.78	0.047	+	X	0.49	0.09
execution	0.67	0.11	0.21	0.76	0.044	+	-	**0.65**	**0.41**
RHEL									
all	0.86	0.03	0.35	0.84	0.037	+	+	0.58	0.08
code	0.91	0.03	0.45	0.82	0.044	+	+	0.49	0.06
dependency	0.85	0.04	0.33	0.84	0.037	+	+	0.60	0.09
Wireshark									
all	0.80	0.12	0.52	0.70	0.068	+	-	0.31	-0.04
code	0.78	0.11	0.52	0.70	0.061	+	-	0.31	-0.17
dependency	0.81	0.12	0.51	0.70	0.062	+	-	0.33	0.04
execution	0.10	0.28	0.03	0.54	0.037	+	-	**0.62**	**0.67**
Tomcat									
all	0.86	0.09	0.29	0.84	0.093	+	+	0.62	0.13
code	0.84	0.09	0.29	0.82	0.105	+	+	0.61	0.04
dependency	0.89	0.10	0.26	0.87	0.074	+	+	0.68	0.24

To check whether one metric type is significantly better than others, this study performed a pairwise comparison using pairwiese.t.test function in R with Bonferroni correction option. Table 21 provides the ordered list of metric types for each project

according to AUC presented in Table 20. In Table 21, "A > B" indicates A metric type provides better prediction performance than B metric type at 0.05 significance level. "A, B" indicates no significant difference in the prediction performance exists between the two metric types A and B. Overall the models with combined set of metrics, code complexity metrics, and network complexity metrics provide similar prediction performance, while the models with execution complexity metrics provide worst prediction performance.

Table 21. Pairwise comparison of AUC between metric types at file level

Project	Comparison of AUC
Firefox	all > complexity, execution all, dependency dependency, complexity dependency > execution
RHEL	all, dependency > code
Wireshark	all, code, dependency > execution
Tomcat	dependency > all, code
All projects	all, code, dependency > execution

Table 22 presents the top three most frequently selected metrics by the InfoGain variable selection method in the order of frequency. In Table 22, the number in the parenthesis is the number of times that a metric has selected by the InfoGain variable selection method. The most frequently selected metrics across projects are outDegree and outDegree_w. For Firefox, all of the execution complexity metrics are included in the top three list. Only two code metrics (SumEssential for RHEL and MaxFanOut for Wireshark) is seen in Table 22.

Table 22. Top three most frequently selected metrics at file level

Firefox 3.0	RHEL 4	Wireshark 1.2	Tomcat 6.0
InclusiveExeTime (86)	OutDegree (99)	OutDegree (94)	OutDegree_w (75)
NumCalls (52)	OutDegree_w (81)	MaxFanOut (68)	EvCent (55)
EvCent (47)	SumEssential (50)	SP_Betweenness (65)	OutDegree (52)

To summarize, the four types of complexity metrics are predictive of vulnerable code locations. The execution complexity metrics provided lower prediction performance. However, the efficiency in reduction of inspection effort is highest among the four types of models.

7.4 Component Level Analysis

For component level, code and execution complexity were computed by summing the metric values in all the files in each component. OO design complexity was computed by summing the metric values in all classes in each component. Network complexity was measured by analyzing the dependency between components.

7.4.1. Correlation Analysis

In Table 23, projects with high percentages of vulnerable components (Firefox 3.0 and RHEL 4) tend to have higher correlations (up to 0.9) than the projects with low percentages of vulnerable components (Wireshark 1.2 and Tomcat 6.0). In Table 23, for Firefox, RHEL, and Tomcat, OutDegree_w provides the highest correlation. For Wireshark,

all three execution complexity metrics provide the highest correlations with vulnerabilities and also provide the highest partial correlations with LOC.

Table 23. Correlations at component level

	Spearman Correlation				**Partial Correlation Controlled for LOC**			
	Firefox	**RHEL**	**Wireshark**	**Tomcat**	**Firefox**	**RHEL**	**Wireshark**	**Tomcat**
Code complexity metrics								
LOC	0.739*	0.830*	0.391*	**0.466***	-	-	-	-
LOCVarDecl	0.721*	0.783*	0.377*	0.484*	-0.022	-0.077	-0.010	0.152
NumFunctions	0.749*	0.831*	0.357*	0.369*	0.177*	0.119	-0.017	-0.167
NumLinePreprocessor	0.633*	0.761*	0.396*		-0.164	-0.037	0.165	
SumEssential	0.762*	**0.867***	0.364*	0.421*	0.275	**0.552***	-0.047	-0.076
SumCyclomaticStrict	**0.767***	0.857*	0.358*	0.449*	0.331*	0.463	-0.112	-0.035
MaxCyclomaticStrict	0.735*	0.795*	0.353*	0.458*	0.124	0.189	-0.028	-0.044
SumMaxNesting	0.596*	0.861*	0.343*	0.444*	-0.449*	0.418	-0.114	0.010
MaxMaxNesting	0.650*	**0.865***	0.326*	0.425*	-0.332*	0.446	-0.121	-0.030
SumFanIn	0.501*	0.818*	0.346*	0.406*	-0.421*	0.178	-0.084	-0.042
MaxFanIn	0.592*	0.849*	0.332*	0.392*	-0.227	0.358	-0.120	-0.034
SumFanOut	0.569*	0.857*	0.379*	0.464*	**-0.605***	0.390	0.035	0.058
MaxFanOut	0.648*	0.863*	0.357*	0.473*	-0.348*	0.500*	0.006	0.117
CommentDensity	-0.282	0.583*	-0.300*	-0.109	-0.300*	0.540*	0.108	-0.012
Dependency network complexity metrics								
InDegree	0.470*	0.531*	0.193*	0.068	0.111	0.127	0.180	-0.171
InDegree_w	0.447*	0.646*	0.219*	0.050	-0.030	0.336	0.164	-0.180*
OutDegree	0.672*	0.840*	0.358*	**0.464***	0.225	**0.566***	0.240*	**0.202***
OutDegree_w	**0.781***	**0.906***	0.395*	**0.491***	0.440*	**0.658***	0.136	**0.222***
SP_Betweenness	0.628*	0.796*	0.426*	0.253*	0.122	0.549*	0.323*	-0.026
Flow_ Betweenness	0.319*	0.839*	0.253*	0.135	0.196	0.479*	0.129	-0.150
RW_ Betweenness	0.244*	0.618*	0.311*	0.348*	0.078	0.251	0.120	0.045
EvCent	0.753*	0.719*	0.256*	0.392*	0.315*	0.241	0.233*	0.117
Execution complexity metrics								
NumCalls	0.734*		**0.491***		**0.485***		**0.399***	
InclusiveExeTme	**0.776***		**0.497***		0.472*		**0.399***	
ExclusiveExeTime	0.728*		**0.491***		**0.552***		**0.406***	
OO design metrics								
CBO	0.691*			0.454*	0.221			0.057
DIT	0.607*			0.331*	0.054			-0.112
NOC	0.597*			0.093	0.108			**-0.225***
LCOM	0.627*			0.368*	0.070			-0.129
WMC	0.691*			0.140	0.227			-0.172
RFC	0.591*			0.377*	0.074			-0.064

7.4.2. Discriminative Power

Table 24 shows the discriminative power tested by Welch's t-test and mean ratios.

Table 24. Discriminative power at component level

	Firefox		RHEL		Wireshark		Tomcat	
	Welch's t-test	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**
Code complexity metrics								
LOC	√	7.6	X	54.5	X	61.1	√	3.7
LOCVarDecl	√	7.3	X	47.8	X	94.8	√	3.4
NumFunctions	√	7.6	X	54.5	X	61.1	X	3.7
NumLinePreprocessor	√	5.7	X	153.2	X	49.1		
SumEssential	√	8.6	X	60.7	X	63.0	X	3.1
SumCyclomaticStrict	√	8.1	X	72.6	X	32.4	X	3.5
MaxCyclomaticStrict	√	10.3	X	49.7	X	30.2	√	3.5
SumMaxNesting	X	4.7	X	91.7	X	23.7	X	3.6
MaxMaxNesting	√	7.1	X	49.1	X	44.7	√	2.8
SumFanIn	X	5.2	X	105.1	X	63.5	***X***	2.9
MaxFanIn	X	11.1	X	74.5	X	43.5	X	2.6
SumFanOut	X	4.8	X	101.3	X	57.6	√	3.7
MaxFanOut	√	7.9	X	64.1	X	44.1	√	3.5
CommentDensity	X	0.7	X	2.1	√	0.4	√	0.6
Dependency network complexity metrics								
InDegree	X	2.3	X	2.3	X	10.3	X	1.2
InDegree_w	X	13.4	X	54.6	X	999.5	X	1.2
OutDegree	√	2.4	X	1.7	X	4.6	√	3.2
OutDegree_w	√	9.7	X	109.5	X	54.2	X	6.2
SP_Betweenness	√	13.2	X	N/A*	X	632.2	X	7.8
Flow_ Betweenness	X	4.5	X	85.8	X	33.3	X	13.8
RW_ Betweenness	√	1.3	X	1.2	X	3.7	X	1.8
EvCent	√	2.2	X	1.8	X	2.1	X	2.7
Execution complexity metrics								
NumCalls	X	12.8			X	169.6		
InclusiveExeTime	X	95.9			X	28613.6		
ExclusiveExeTime	X	8.6			X	169.0		
OO design metrics								
CBO	X	8.4					X	3.5
DIT	X	6.7					X	1.8
NOC	X	5.2					X	0.5
LCOM	√	6.7					X	2.4
WMC	X	6.1					X	1.2
RFC	X	5.3					X	2.4

*. The mean of SP_Betweenness for neutral files was zero and the mean ratio was not computable.

In Table 24, less than half of the metrics are statistically significant and none of the metrics are significant across the four projects. Despite of the lack of significance in discriminative power, means of metric values from vulnerable components were higher than the means of them from neutral components for most metrics with a few exceptions (CommentDensity for three projects and NOC from Tomcat). Section 7.6 further discusses the lack of significance in discriminative power at component level.

7.4.3. Predictability

Table 25 provides the prediction results at component level. For the prediction at component level, 3x3 cross-validation was performed because the number of components were too small to split into ten folds. In Table 25, the projects with high percentage of vulnerable components (Firefox 3.0 and RHEL 4) provide both high recall (0.78 to 0.88) and high precision (0.75 to 1) while the projects with low percentage of vulnerable components (Wireshark 1.2 and Tomcat 6.0) provide lower recall (0.4 to 0.82) and much lower precision (0.26 to 0.37) except for the model with execution complexity metrics from Wireshark whose precision is 0.79. In all predictions except the model with execution metrics for Wireshark, AUC is significantly higher than 0.5 indicating the prediction results are better than random classification. However, only four of the 16 models provided significantly higher AUC than performance benchmark. As with file level analysis, the models with execution complexity metrics provide highest efficiency in reducing inspection effort.

The pairwise comparison of AUC between the models with different types of metrics showed no significant difference.

Table 25. Predictability at component level

	Recall	Precision	PF	AUC	Std(AUC)	AUC ≠ 0.5	AUC ≠ 0.785	FIR	LIR
Firefox									
all	0.78	0.89	0.13	0.89	0.096	+	X	0.37	0.04
complexity	0.79	0.83	0.2	0.88	0.092	+	X	0.33	-0.07
dependency	0.78	0.92	0.11	0.88	0.098	+	X	0.40	0.03
execution	0.85	0.79	0.29	0.81	0.100	+	X	**0.30**	**0.16**
design	0.83	0.75	0.33	0.80	0.088	+	X	0.27	0.06
RHEL 4									
all	0.80	0.94	0.11	0.93	0.110	+	+	0.26	-0.1
complexity	0.80	0.9	0.17	0.90	0.120	+	+	0.23	-0.14
dependency	0.88	1	0	1	0	+	N/A	0.31	0.05
Wireshark									
all	0.53	0.26	0.24	0.69	0.093	+	X	0.43	0.07
complexity	0.62	0.26	0.29	0.70	0.115	+	X	0.39	0.05
dependency	0.62	0.28	0.25	0.74	0.149	+	X	0.45	-0.04
execution	0.40	0.79	0.02	0.69	0.137	X	X	**0.82**	**0.28**
Tomcat									
all	0.76	0.33	0.28	0.85	0.055	+	+	0.55	0.12
complexity	0.79	0.33	0.29	0.86	0.069	+	X	0.55	0.13
dependency	0.82	0.37	0.24	0.88	0.042	+	+	0.61	0.21
design	0.75	0.36	0.23	0.86	0.077	+	X	0.59	0.15

Table 26 presents the top three most frequently selected metrics by the InfoGain variable selection method at component level. The most frequently selected metric across projects is outDegree_w in Table 26. None of the execution complexity metrics and OO design complexity metrics were chosen by the InfoGain variable selection method.

Table 26. Top three most frequently selected metrics at component level

Firefox	RHEL	Wireshark	Tomcat
OutDegree_w (8)	OutDegree_w (5)	RW_Betweenness (3) EvCent (3) MaxMaxNesting (3)	OutDegree_w (5)
SumCyclomaticStrict (3) OutDegree (3) EvCent (3)	SumMaxNesting (4) MaxMaxNesting (4) SP_Betweenness(4)	SumMaxNesting (2) LOCVarDecl (2) MaxCyclomaticStrict (2) OutDegree (2) NumLinePreprocessor (2) MaxFanOut (2)	SumFanOut (3) LOCVarDecl (3)
SumEssential (2) SumCountLinePreprocesor (2)	OutDegree (3)	SumFanout (1) MaxFanIn (2) InDegree (1) SumFanIn (1) OutDegree_w (1) Flow_Between (1)	SumMaxNesting (2) MaxFanIn (2) MaxCyclomaticStrict (2) OutDegree (2) LOC (2) MaxFanOut (2) MaxMaxNesting (2)

7.5 Class Level Analysis

Class level analysis was performed using code complexity, network complexity, and OO design complexity metrics.

7.5.1. Correlation analysis

Table 27 shows the Spearman correlations between vulnerabilities and complexity metrics at class level. Similar to file level analysis, the correlations are less than 0.3, but most correlations are statistically significant at 0.05 significance level for most metrics. While OutDegree and OutDegree_w provided high at file and component level, LOC, SumCyclomaticStrict, and CBO were provided the highest correlations.

Table 27. Correlations at class level

	Spearman Correlation		Partial Correlation Controlled for LOC	
	Firefox	**Tomcat**	**Firefox**	**Tomcat**
Code complexity metrics				
LOC	**0.209***	**0.224***	-	-
LOCVarDecl	0.205*	0.218*	0.007	0.017
NumLinePreprocessor	0.187*		**0.107***	
SumEssential	0.205*	0.209*	0.031*	0.037
SumCyclomaticStrict	**0.207***	**0.223***	0.030*	0.048
MaxCyclomaticStrict	0.203*	0.212*	0.048*	0.057*
CommentDensity	0.120*	0.083*	0.053*	0.009
Dependency network complexity metrics				
InDegree	0.076*	0.066*	0.059*	0.016
InDegree_w	0.085*	0.097*	0.057*	0.033
OutDegree	0.202*	0.212*	**0.082***	0.126*
OutDegree_w	0.201*	0.220*	0.071*	**0.115***
SP_Betweenness	0.129*	0.180*	0.059*	**0.089***
EvCent	0.179*	0.104*	**0.080***	0.050
OO design metrics				
CBO	**0.206***	**0.222***	0.072*	**0.081***
DIT	0.055*	0.007	0.030*	0.008
NOC	0.024	0.01	0.053*	-0.002
LCOM	0.131*	0.199*	0.006	0.049
WMC	0.080*	0.071*	0.037*	-0.008
RFC	0.145*	0.153*	0.040*	0.044

7.5.2. Discriminative Power

Table 28 shows the results of Welch's t-test. In Table 28, 79% and 63% of metrics showed statistically significant difference between vulnerable class and neutral classes for Firefox and Tomcat, respectively. The means of metric values for vulnerable classes are higher than the means of metric values for neutral classes for most metrics in both projects except for CommentDensity, EvCent, DIT, and NOC.

Table 28. Discriminative power at class level

	Firefox		Tomcat	
	Welch's t-test	**Mean Ratio**	**Welch's t-test**	**Mean Ratio**
Code complexity metrics				
LOC	√	5.9	√	5.4
LOCVarDecl	√	5.4	√	4.6
NumLinePreprocessor	√	7.7		
SumEssential	√	5.5	√	4.9
SumCyclomaticStrict	√	6.0	√	5.8
MaxCyclomaticStrict	√	3.5	√	4.2
CommentDensity	X	1.0	***X***	1.0
Dependency network complexity metrics				
InDegree	X	1.9	X	1.3
InDegree_w	X	2.2	X	1.7
OutDegree	√	3.5	√	3.9
OutDegree_w	√	6.3	√	6.7
SP_Betweenness	√	11.1	X	7.6
EvCent	√	2.4	√	0.1
OO design metrics				
CBO	√	3.6	√	3.8
DIT	√	1.3	X	1.0
NOC	X	1.3	X	0.6
LCOM	√	1.5	√	2.0
WMC	√	4.5	X	1.3
RFC	√	2.5	√	2.0

7.5.3. Predictability

Table 29 provides the prediction results using 10x10 cross-validation at class level. Similar to file level analysis, recall is high (0.72 to 0.91), but precision is very low (0.08 to 0.10). PF is less than 0.32.

In all predictions, AUC is higher than 0.5 indicating the prediction results are better than random classification of files. AUC is also higher than the performance benchmark

Table 29. Predictability at class level

	Recall	Precision	PF	AUC	Std(AUC)	AUC ≠ 0.5	AUC ≠ 0.785	FIR	LIR
Firefox									
all	0.73	0.09	0.27	0.81	0.051	+	+	0.59	-0.04
code	0.73	0.09	0.27	0.82	0.047	+	+	0.60	-0.04
dependency	0.77	0.09	0.31	0.79	0.055	+	X	0.56	0.07
design	0.72	0.08	0.32	0.76	0.060	+	-	0.52	-0.01
Tomcat									
all	0.86	0.10	0.20	0.89	0.084	+	+	0.73	0.13
code	0.86	0.10	0.21	0.89	0.076	+	+	0.72	0.09
dependency	0.86	0.09	0.23	0.88	0.090	+	+	0.70	0.16
design	0.91	0.10	0.21	0.89	0.093	+	+	0.73	0.24

(AUC of 0.785) except for the prediction results using network complexity metrics and OO design complexity metrics for Firefox.

Table 30 shows the pairwise t-test comparison results. As explained in Section 7.3.3, "A > B" indicates A metric type provides better prediction performance than B metric type at 0.05 significance level. "A, B" indicates no significant difference in the prediction performance exists between the two metric types A and B. For Firefox, the models using all complexity metrics and code complexity metrics provide significantly better performance than network complexity metrics. The models using OO design complexity metrics provide worst prediction performance. For Tomcat, there is no significant difference in prediction performance between the models using all complexity metrics, code complexity metrics, and network complexity metrics. There is also no significant difference in prediction performance between the models using all complexity metrics, code complexity metrics, and design complexity metrics. However, pairwise t.test shows that the model using OO design

complexity metrics is statistically significantly better than the model using network complexity metrics.

Table 30. Pairwise comparison of AUC between metric types at class level

	Comparision of AUC
Firefox	all, code > dependency > design
Tomcat	all, code, dependency, design

Table 31 presents the top three most frequently selected metrics by the InfoGain variable selection method at class level. LOC was selected in both projects. Among the six OO design complexity metrics, CBO was selected in both projects.

Table 31. Top three most frequently selected metrics at class level

Firefox 3.0	**Tomcat 6.0**
LOC (67)	CBO (63)
SumCyclomaticStrict (58)	LOC(56)
CBO (44)	OutDegree_w (41)

7.6 Discussion

OutDegree and OutDegree_w tend to have higher correlations with vulnerabilities than other metrics for the projects whose execution metrics are not highly correlated with vulnerabilities. For Firefox at file level and Wireshark at component level, execution complexity metrics provided the highest correlations with vulnerabilities even after those metrics are controlled for LOC.

In general, complexity metrics investigated in this study have discriminative power of vulnerable and neutral files and classes, but the evidence of having discriminative power was weak at component level. As shown in Equation (1) in Section 4.1, Welch's t-test depends on the means, standard deviations, and sample sizes of compared two groups. The larger the mean difference, the smaller the standard deviations, and the larger the sample sizes, the difference of mean is significant. The same sizes at component level are much smaller than at file level. Standard deviations of metric values at component level are much larger at file level. For example, standard deviation of LOC at file level is 764, but standard deviation of LOC at component level is 67,226. These facts partly explain why p-values are not small enough at component level. Despite of the lack of significance in discriminative power, means of metric values from vulnerable components were higher than the means of them from non-vulnerable components for most metrics.

In all predictions, the models with each type of complexity metrics and with combined set of complexity metrics provided statistically significantly better prediction performance than random classification except for the model using execution metrics for Firefox. However, 16 of 38 models across entity levels provided significantly better performance than the performance benchmark (AUC of 0.785) and 6 of 38 models provided significantly worse performance than performance benchmark. Other models did not show significant evidence that the performance is different from the performance benchmark.

While execution complexity metrics provided lower prediction performance than other complexity metrics, inspection reduction based on the prediction results was highest for both Firefox and Wireshark at both file and component level. The low recall for execution complexity metrics is natural because not all code has been executed with the use cases that this study used. Executing the programs with additional use cases may improve recall. However, adding more use cases may also decrease precision. Therefore, inspecting the small set of entities executed most heavily (frequently for long duration) first and inspecting the remaining entities based on the prediction results with other static complexity metrics seems to be an efficient approach to detect the most vulnerabilities with less inspection effort.

The analysis at various levels of entity granularity in this study reveals some concerns that researchers and practitioners should care when coarse entity level analysis is performed. First, the projects with high percentages of faults or vulnerabilities provide higher correlations and higher prediction results than the projects with low percentages of vulnerabilities. To see this phenomenon more clearly, Figure 18 shows the plots of four synthetic data sets. The synthetic data sets were made from Firefox 3.0 data with the same percentages of vulnerable entities, but simplified distribution of LOC and vulnerabilities. The data sets in Figure 18 (a) and Figure 18 (b) have the same number of entities and the same number of vulnerable entities as the ones for components and files of Firefox 3.0,

respectively, making the percentages of vulnerable entities in (a) 55% and (b) 4%, respectively. In Figure 18 (a) and Figure 18 (b), LOC increases linearly; entities with small LOC have 0 vulnerability; entities with large LOC have one vulnerability. In Figure 18, coarse granularity entities provide much higher correlation (ρ, $r = 0.86$) than fine granularity entities (ρ, $r = 0.33$) in (a) and (b). This is true even when the number of vulnerabilities increases as LOC increases as in (c) and (d). Hence, the percentage of vulnerable entities almost works as an upper bound of correlations.

Interpretation of discriminative power also requires caution as mentioned earlier. Therefore, when fault and vulnerability prediction studies do not specify the percentage of faults with their prediction results, direct comparison between studies are not valid. This is also true for the studies that use the same projects if the data collection method, entity granularity, and dependent variable (faults or vulnerabilities) are not exactly the same between the two compared studies. Another concern of using coarse entities as a unit of fault or vulnerability prediction is that, although using coarse granularity entities will improve the predictability, coarse level prediction is not as valuable as fine granularity entities because it is difficult for software engineers to guide to the specific code locations that are likely to have vulnerabilities.

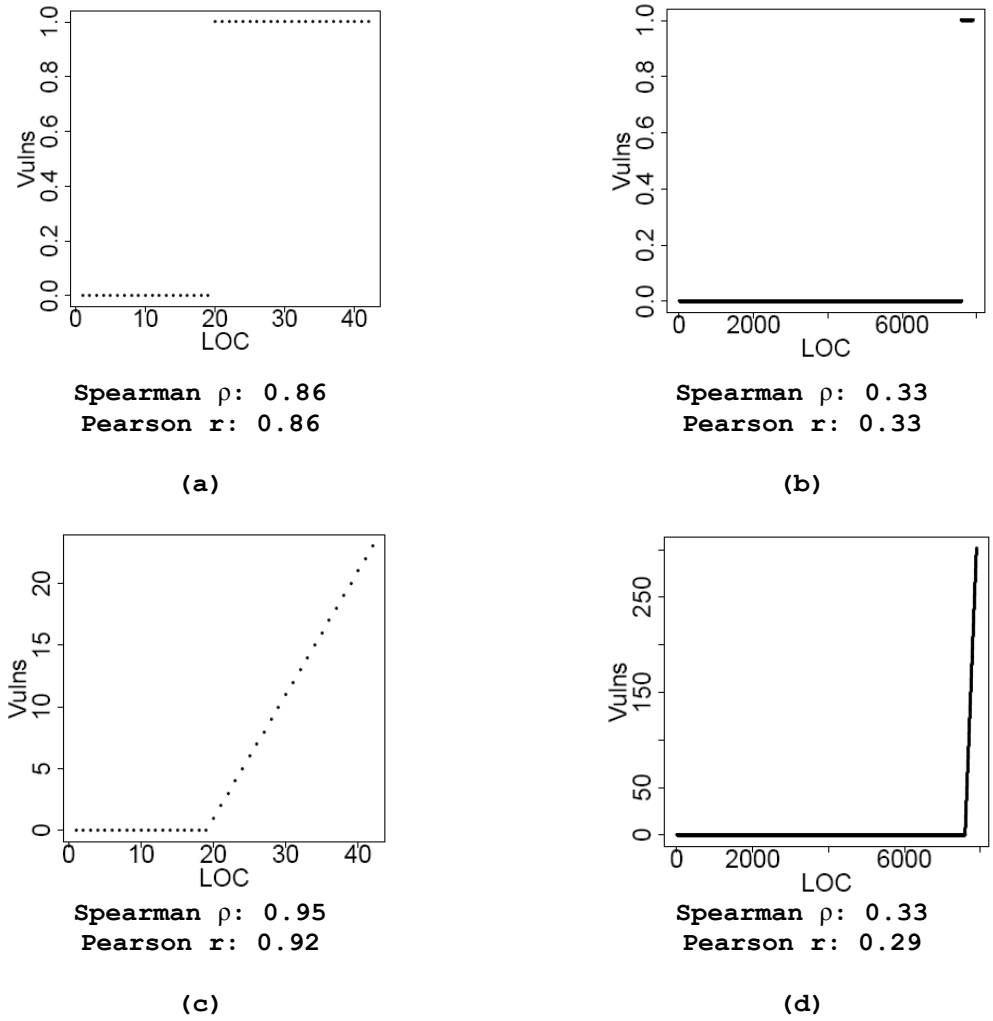

Figure 18. Correlation comparison between different entity granularities

7.7 Summary

This study investigated four types of complexity metrics as indicators of vulnerabilities. In this study, correlations, discriminative power, and predictability were measured for the complexity metrics on four widely used open source projects: the Mozilla Firefox 3.0 web browser, the Red Hat Enterprise Linux 4 kernel, the Wireshark 1.2.0 network protocol analyzer, and the Apache Tomcat 6.0 servlet container. The analysis has been performed at file, component, and class level.

Correlations greatly varied depending on the percentage of vulnerable entities. At file level, the Spearman correlations were up to around 0.3 and at component level, the Spearman correlations were up to around 0.9. However, most correlations were statistically significant showing that the correlations are not by chance.

Over 70% of the metrics in Firefox, RHEL, and Tomcat, and 43% of the metrics in Wireshared showed significant discriminative power at file level. Less than half of the metrics for Firefox and Tomcat and almost none of the metrics for RHEL and Wireshark showed significant discriminative power at component level. However, regardless of the statistical significance, the means of metric values for vulnerable entities were greater than the means of metric values for neutral entities for most of the metrics.

Predictability of the four types of complexity metrics and the combined set of complexity metrics was examined by building predictive models for each type of metrics.

Among the 38 prediction models, 37 models provided better performance than random classification. When tested against performance benchmark (AUC of 0.785), the prediction models using the execution complexity metrics for both Firefix and Wireshark and all the models for Wireshark were less predictive than the performance benchmark at file level. At class level, all the prediction models were superior to the performance benchmark except for the models using network complexity metrics and OO design complexity metrics.

Overall, code complexity, OO design complexity, network complexity, and execution complexity metrics can discriminative vulnerable and neutral code locations at file and class level and can predict vulnerable code locations at file, component, and class level better than a random classification model. However, many of the prediction models did not achieve the benchmarking performance. The best metric type varied depending on projects or the difference in predictability between models using different types of metrics were not significant except that execution complexity metrics showed worst prediction performance at the file level (see Table 21 and Table 30). Although the prediction models using execution complexity metrics provided lower predictability than other models, the percentage of vulnerable entities in commonly executed entities were certainly higher than the percentage of vulnerable entities in the entire system and provided higher efficiency in reducing inspection effort. Therefore, inspecting the files predicted from the models with execution

complexity metrics first and then inspecting the remaining files predicted from the models with static complexity metrics seems to be an efficient approach.

7.8 Threats to Validity

This study has the same limitations of limited accuracy of the collected vulnerability data as was in Chapter 5 and Chapter 6: limited traceability from a vulnerability report to the actual vulnerable code locations and the problem of undiscovered vulnerabilities. However, since this research used the large scale open source projects whose bug report process and source code management process are stable, the vulnerable code locations that were found in this research should reflect the most representative vulnerable code locations in the four subject projects.

The component level analysis was performed based on the directory structure. However, the variance between the numbers of files in directories is large. For example, the largest component of Firefox 2.0 has 251 files and the smallest component has only one file. The largest component of RHEL 4 has 4,702 files and the smallest component has only one file. Depending on the way of grouping the files, the prediction results may be different. For example, some component should be combined together while other component should be split to represent the functional relevance of files in a similar level between components. Although this research performed the analysis at component level for informational purpose,

finer level analyses such as file or class level provide more useful guidance to organizations for security inspection and testing activities.

Execution complexity metrics vary depending on the use cases. This research collects execution complexity metrics from informally chosen common usage patterns by normal users. However, most common usage patterns also may vary depending on the users or organizations that use the software projects. Further analysis on the common usage patterns of software using survey or observation may improve the objectivity and thus repeatability of data collection.

CHAPTER 8

Conclusions

This research investigated whether a set of complexity metrics that were collected from software statically and dynamically can indicate vulnerable code locations using empirical case studies on four widely used, large size open source projects: the Mozilla Firefox 3.0 web browser; the Red Hat Enterprise Linux 4 (RHEL4) operating system kernel; the Wireshark 1.2.0 network protocol analyzer; and the Apache Tomcat 6.0 servlet container.

Since complexity metrics have been effective for fault prediction, this study also investigated whether organizations can get benefits by creating a vulnerability prediction model separately from a fault prediction model. Because there are complex code that are rarely executed, this study additionally investigated whether execution complexity measured by frequency and duration of code execution can indicate vulnerable code locations. For this purpose, this research set up two hypotheses and a research question. Then this research tested the hypotheses and answered the question by performing statistical tests on the data collected from the four projects. Table 32 repeats the two hypotheses and a research question and summarizes the results.

Table 32. Results of hypothesis testing and answer for a research question

H1.	Complexity metrics statically obtained from software artifacts can discriminate vulnerable code locations and neutral code locations and can predict vulnerable code locations.	Supported except for discriminative power at component level
H2.	Vulnerability prediction models trained to predict vulnerabilities can provide better performance than fault prediction models trained to predict faults in the ability to predict vulnerable code locations.	Not supported
Q3'.	Can the execution complexity metrics obtained from usage patterns by normal users indicate vulnerable code locations?	Yes

The results from Hypothesis 1 testing indicate that complexity metrics are effective to discriminate and predict vulnerable code locations. This research also showed that process metrics are better indicators than complexity metrics. However, finding effective complexity metrics is still important because not all projects have process history especially if they are initial projects. Although the results are promising, the prediction models require significant improvement to reduce false positives.

The results from Hypothesis 2 indicates that organizations can utilize fault prediction models to predict vulnerabilities with the similar predictability and the similar code inspection and testing efficiency to the ones provided by vulnerability prediction models.

However, some projects maintain the vulnerability information rigorously and provide links traceable to the vulnerable code locations, while fault information is mixed with faults and functional enhancements and not always traceable to faulty code locations. Therefore, for some projects, only vulnerability prediction models can be constructed unless the organization wants to use text mining to separate out faults from enhancements as was done in this research.

As an answer for Q3', the models with execution complexity metrics resulted in low recall because of low coverage of executed code and also showed widely different results between projects. However, the models with execution complexity metrics were able to predict vulnerable code locations with high inspection reduction compared to other models.

Although the complexity metrics can statistically significantly indicate vulnerable code locations, low precision still matters. For this problem of searching for "needles in a haystack", efforts to find the small set of code locations that are most likely to have vulnerabilities seems more beneficial than the efforts to prioritize the entire system. Metrics, models, and evaluation criteria should focus on the small set of the most vulnerable code areas. The prediction models based on the currently known software metrics including complexity metrics and process metrics can prioritize code locations that software engineers should focus on. However, which vulnerabilities software engineers should find from the predicted entities is at the discretion of software engineers. Therefore, finding a way to guide

software engineers by providing further information on which vulnerabilities might be in the predicted code may improve the utility of vulnerability prediction models.

REFERENCES

[1] *Metrics data program: NASA IV & V facility*, retrieved Nov. 15th 2010, http://mdp.ivv.nasa.gov/repository.html.

[2] 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.

[3] 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. 2008 Conference of the Center for Advanced Studies on Collaborative Research*, Ontario, Canada, Oct. 27-30, 2008.

[4] 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.

[5] 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.

[6] B.W. Boehm, *Software engineering economics*, Englewood Cliffs, NJ: Prentice-Hall Inc., 1981.

[7] P. Bonacich, "Power and centrality: A family of measures," *American Journal of Sociology*, vol. 92, no. 5, 1987, pp. 1170-1182.

[8] U. Brandes, and T. Erlebach, *Network analysis: Methodological foundations*, Lecture notes in computer science, Springer, 2005.

[9] L. Breiman, "Random forests " *Machine Learning*, vol. 45, no. 1, 2001, pp. 5-32.

[10] L.C. Briand, J. Wust, S.V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," *Proc. the 1999 International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, USA, 16-22 May, 1999, pp. 345-354.

[11] B. Cashell, W.D. Jackson, M. Jickling, and B. Web, "CRS report for congress: The economic impact of cyber-attacks," Congressional Research Service, April 1, 2004.

[12] S.R. Chidamber, and C.F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994, pp. 476-493.

[13] complexity, *Dictionary.Com unabridged*, Random House, Inc, retrieved Nov. 13 2010, http://dictionary.reference.com/browse/complexity.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms*, 2nd ed., Cambridge, Massachusets, 2001.

[15] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, vol. SE-5, no. 2, 1979, pp. 96- 104.

[16] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, no., 2006, pp. 1 - 30.

[17] G. Denaro, "Estimating software fault-proneness for tuning testing activities," *Proc. 22nd international conference on Software engineering*, Limerick, Ireland, June 04-11, 2000, pp. 704-706.

[18] A. Drozdek, *Data structures and algorithms in java*, Singapore: Cengage Learning Asia Pte Ltd, 2008.

[19] K.E. Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Software Eng.*, vol. 27, no. 7, 2001, pp. 630 - 650.

[20] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no., 2006, pp. 861-874.

[21] N.E. Fenton, and S.L. Pfleeger, *Software metrics: A rigorous and practical approach*, 1997.

[22] L.C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, 1977, pp. 35-41.

[23] L.C. Freeman, "Centrality in social networks: Conceptual clarification," *Social Networks*, vol. 1, no., 1979, pp. 215-239.

[24] L.C. Freeman, S.P. Borgatti, and D.R. White, "Centrality in valued graphs: A measure of betweenness based on network flow," *Social Networks*, vol. 13, no., 1991, pp. 141-154.

[25] D.E. Geer, "A witness testimony in the hearing, wednesday 25 april 07, entitled addressing the nation's cybersecurity challenges: Reducing vulnerabilities requires strategic investment and immediate action," *submitted to the Subcommittee on Emerging Threats, Cybersecurity, and Science and Technology*, vol., no., 2007.

[26] 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.

[27] 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.

[28] M. Girvan, and M.E.J. Newman, "Community structure in social and biological networks," *the National Academy of Sciences*, vol. 99, no. 12, 2001, pp. 7821-7826.

[29] 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.

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

[31] I. Guyon, and A. Elisseeff, "An introduction to variable and feature selection," *J. Machine Learning Research*, vol., no., 2003, pp. 1157-1182.

[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.

[33] IEEE, "IEEE std 982.1-1988 IEEE standard dictionary of measures to produce reliable software," IEEE Computer Society IEEE Std 982.1-1988, June 9, 1988.

[34] IEEE, "IEEE standard for a software quality metrics methodology," IEEE Computer Society, June 24, 2004.

[35] A. Jaquith, *Security metrics: Replacing fear, uncertainty, and doubt*, Upper Saddle River, NJ: Pearson Education, Inc, 2007.

[36] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," *Proc. the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, Leipzig, Germany, May 12–13, 2008, pp. 11-18.

[37] 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.

[38] T.M. Khoshgoftaar, and J.C. Munson, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, 1990, pp. 253-261.

[39] T.M. Khoshgoftaar, and J.C. Munson, "The lines of code metric as a predictor of program faults: A critical analysis," *Proc. the 14th Annual International Computer Software and Applictions Conference (COMPSAC90)*, Chicago, IL, USA, 31 Oct-2 Nov, 1990, pp. 408-413.

[40] T.M. Khoshgoftaar, and J.**C.** Munson, "Dynamic system complexity," *Proc. First International Software Metrics Symposium*, 1993, pp. 129 - 140.

[41] T.M. Khoshgoftaar, R. Shan, and E.B. Allen, "Using product, process, and execution metrics to predict fault-prone software modules with classification trees," *Proc. Fifth IEEE International Symposim on High Assurance Systems Engineering*, 2000, pp. 301 - 310.

[42] 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.

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

[44] 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.

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

[46] G. McGraw, *Software security: Building security in*, Boston, NY: Addison-Wesley, 2006.

[47] T. Mende, and R. Koschke, "Revisiting the evaluation of defect prediction models," *Proc. Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE'09)*, Vancouver, Canada, 2009.

[48] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, Georgia, Nov. 9-14, 2008, pp. 13-23.

[49] 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. 453-462.

[50] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "Comments on 'data mining static code attributes to learn defect predictors'"," *IEEE Trans. Software Eng.*, vol. 33, no. 9, 2007, pp. 637-640.

[51] 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.

[52] 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.

[53] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," *Proc. the 13th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 10 - 18 May, 2008, pp. 181-190.

[54] J.D. Musa, "Operational profiles in software reliability engineering," *IEEE Software*, vol. 10, no. 2, 1993, pp. 14 - 32.

[55] 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.

[56] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," *Proc. the 17th International Symposium on Software Reliability Engineering*, Raleigh, NC, U.S.A., November 7-10, 2006, pp. 62-74.

[57] 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.

[58] N. Nagappan, and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," *Proc. First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 20-21 Sept., 2007, pp. 364-373.

[59] 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.

[60] S. Neuhaus, and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," *Proc. 2009 USENIX Annual Technical Conference*, San Diego, CA, 2009.

[61] M.E.J. Newman, "A measure of betweenness centrality based on random walks," *Social Networks*, vol. 27, no., 2005, pp. 39-54.

[62] 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.

[63] T.J. Ostrand, E.J. Weyuker, and R.M. Bell:, "Automating algorithms for the identification of fault-prone files," *Proc. the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, London, UK, July 9-12, 2007, pp. 219 - 227.

[64] R.L. Ott, and M. Longnecker, *An introduction to statistical methods and data analysis*, 5th ed., Pacific Grove: Duxbury, 2001.

[65] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?," *Proc. International Symposium on Foundations in Software Engineering*, Atlanta, GA, 9-14 Nov., 2008, pp. 2-12.

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

[67] B. Schneier, *Beyond fear: Thinking sensibly about security in an uncertain world*, New York, NY: Springer-Verlag, 2003.

[68] 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.

[69] 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, Germany, 2008, pp. 315-317.

[70] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, "Does calling structure information improve the accuracy of fault prediction?," *Proc. 6th IEEE International Working Conference on Mining Software Repositories*, Vancouver, BC, Canada, May 16-17, 2009, pp. 61 - 70.

[71] Y. Shin, and L. Williams, "Can fault prediction models and metrics be used for vulnerability prediction?," North Carolina State University TR-2010-6, August 11, 2010.

[72] R. Subramanyam, and M.S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Software Eng.*, vol. 29, no. 4, 2003, pp. 297-310.

[73] B. Turhan, T. Menzies, A.B. Bener, and J.D. Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, 2009, pp. 540-578.

[74] J. Walden, M. Doyle, G.A. Welch, and M. Whelan, "Security of open source web applications," *Proc. 3rd International Symposium on Empirical Software Engineering and Measurement*, Lake Buena Vista, Florida, Oct. 14, 2009, pp. 545-553.

[75] A.H. Watson, and T.J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," NIST Special Publication 500-235, National Institute of Standards and Technology, September, 1996.

[76] L. Weissman, "Psychological complexity of computer programs: An experimental methodology," *ACM SIGPLAN Notices*, vol. 9, no. 6, 1974, pp. 25 - 36.

[77] E.J. Weyuker, T.J. Ostrand, and R.M. Bell, "Using developer information as a factor for fault prediction " *Proc. International Workshop on Predictor Models in Software Engineering (PROMISE '07)*, Minneapolis, MN, 20 May, 2007.

[78] E.J. Weyuker, T.J. Ostrand, and R.M. Bell, "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, 2008, pp. 539-559.

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

[80] Y. Zhou, and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Trans. Software Eng.*, vol. 32, no. 10, 2006, pp. 771 - 789.

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

[82] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," *Proc. 3rd International Conference on Software Testing, Verification and Validation*, Paris, France, Apr. 6-11, 2010, pp. 421 - 428.