8-10-2018

A Software Vulnerability Prediction Model Using Traceable Code Patterns And Software Metrics

Kazi Zakia Sultana

A software vulnerability prediction model using traceable code patterns and software metrics

By

Kazi Zakia Sultana

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2018

A software vulnerability prediction model using traceable code patterns

and software metrics

By

Kazi Zakia Sultana

Approved:

Byron J. Williams
(Major Professor)

Eric Hansen
(Committee Member)

Sarah B. Lee
(Committee Member)

Stefano Iannucci
(Committee Member)

Mike J. Phillips
(Committee Member)

T. J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Kazi Zakia Sultana

Date of Degree: August 10, 2018

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Byron J. Williams

Title of Study: A software vulnerability prediction model using traceable code patterns and software metrics

Pages of Study: 112

Candidate for Degree of Doctor of Philosophy

Context: Software security is an important aspect of ensuring software quality. The goal of this study is to help developers evaluate software security at the early stage of development using traceable patterns and software metrics. The concept of traceable patterns is similar to design patterns, but they can be automatically recognized and extracted from source code. If these patterns can better predict vulnerable code compared to the traditional software metrics, they can be used in developing a vulnerability prediction model to classify code as vulnerable or not. By analyzing and comparing the performance of traceable patterns with metrics, we propose a vulnerability prediction model. *Objective:* This study explores the performance of code patterns in vulnerability prediction and compares them with traditional software metrics. We have used the findings to build an effective vulnerability prediction model. *Method:* We designed and conducted experiments on the security vulnerabilities reported for Apache Tomcat (Releases 6, 7 and 8), Apache CXF and three stand-alone Java web applications of Stanford Securibench. We used machine

learning and statistical techniques for predicting vulnerabilities of the systems using traceable patterns and metrics as features. *Result:* We found that patterns have a lower false negative rate and higher recall in detecting vulnerable code than the traditional software metrics. We also found a set of patterns and metrics that shows higher recall in vulnerability prediction. *Conclusion:* Based on the results of the experiments, we proposed a prediction model using patterns and metrics to better predict vulnerable code with higher recall rate. We evaluated the model for the systems under study. We also evaluated their performance in the cross-dataset validation.

Key words: vulnerability, software security, software quality, software testing, software metrics, nano-patterns, micro patterns

DEDICATION

To my parents Kazi Abul Kashem and Roksana Akther.

ACKNOWLEDGEMENTS

First and foremost I want to thank my advisor Dr. Byron Williams. I am really grateful to my advisor for his enormous dedication towards fulfilling my degree. I appreciate all his contributions in developing research ideas, writing papers, providing support, and funding to complete my Ph.D. He was not only motivational and caring for my academic and research related activities, he was also a great support for me even during tough times in the Ph.D. pursuit. I really believe that without his contribution, I would not be able to achieve this degree. I am also thankful to the members of the ESE group who have given their significant advice and important guidelines to improve my experiments, ideas, and presentation skill. The ESE meeting in every week really helped me to learn about different research potentials in my relevant area. The group has been a source of friendship as well as good direction and collaboration. I would like to acknowledge honorary group member Dr. Edward B. Allen who retired a few years ago. I am also grateful to the members of my PhD committee who have given their valuable advice and reviews of my work since my preliminary examination of PhD.

I would also like to thank specially to Dr. Donna Reese who was the head of the department of Computer Science and Engineering when I first started my PhD in Mississippi State University. It would not be possible for me to start my PhD if she did not provide financial support at that time. She also cooperated me in every step whenever I needed

any support or recommendation from the department for participating in any conference or competing for any award. I am also indebted to my MS advisor Dr. Hasan Jamil who built my background for PhD through giving me opportunities to work with him. I could learn how to develop ideas and write papers during my MS from his enormous research experience. It was possible to publish a number of papers during my MS because of his extraordinary motivation and support at that time.

I would like to thank the department of Plant and Soil Sciences of Mississippi State University which has been supporting my research assistant position for last two years. I would be happy to thank my two lab mates Ajay Deo and Dr. Zadia Codabux for their collaboration in some of my papers. I am really grateful for the pattern extraction tool developed by Dr. Singer that I used to do the experiments of the dissertation.

Finally, I think my achievements would not be possible without the continuous support and inspiration that I got from my family members. I believe that it is my parents without whom I could never get my strength and motivation back to start my PhD. They supported me in all my pursuits since my childhood. My parents, my brothers, and my other family members have made my life indebted to them by giving me mental strength and inspiration throughout this journey. Finally, the person who was always beside me during the long and difficult path of my PhD is my loving husband Muhammad Aminul Islam. I believe that his company and his continuous cooperation made my struggle worth and fruitful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software vulnerability research is crucial for enabling improvements to the state-of-the-art in software engineering focused on delivering functional and secure software projects. Although a software vulnerability may be deemed as a special kind of software bug, this bug violates security policies and results in serious consequences that makes the software exploitable to malicious attack. According to Munaiah et al. in [34], while missing, insufficient, or incorrect functionality can be referred as traditional bugs, vulnerabilities are defined as the misuse of functionality that deviates the software from its intended behavior and allows exposure to malicious attackers. According to [31], vulnerability discovery differs from defect discovery in development the process. Therefore, taking precaution against vulnerabilities is a dominant factor in ensuring security and mitigating risks. Developers are encouraged adopt secure coding practices and to follow security policies during the software implementation phase. There have been a number of security patterns and guidelines for developers to follow [19, 56, 40, 23]. Following these rules and procedures can reduce the probability of vulnerable code, but they do not guarantee a seal to all kinds of security leakages that may be exploitable by attackers. There is a need for methods that can highlight certain code constructs that identify vulnerable code characteristics.

To minimize the likelihood of vulnerabilities developers can investigate their code repository and vulnerability history in order to find code constructs that are susceptible to vulnerability. Vulnerability history in earlier releases can then be used to mine code constructs that are associated with vulnerabilities and then build a vulnerability prediction model [8]. In earlier studies, software metrics were extracted from existing software repositories and then used as features in supervised machine learning to build vulnerability prediction models [45, 42, 43, 46, 44, 6, 7, 8]. In these studies, the authors considered file-level vulnerability prediction and did not distinguish between class-level and method-level software metrics. In addition, although they showed reasonable accuracy at vulnerability prediction, their false negative rates were high resulting in failure to detect many vulnerable files. Moreover, vulnerability prediction at a lower granular level assists developers to precisely identify the location of the vulnerable code. Developers may not be able to locate or pinpoint the source of the vulnerability in a file if it consists of a number of classes and/or methods. A developer needs a significant amount of time to examine all methods in order to locate a particular bug in a large file [17]. Prediction at source file level also reduces accuracy [31]. Binary-level predictions are also not desirable as no specific action can be taken due to the size of the binaries that contain hundreds of source files [31]. Therefore, software developers are still in need of a mechanism to classify vulnerable code at varying levels of granularity. Our goal is to build a vulnerability prediction model using extracted code constructs. We focused on the implementation phase using class-level and method-level code patterns to highlight potentially vulnerable areas in the code. These patterns capture the object-oriented features of code and have not previously been extensively ana-

lyzed. The identified code areas that are prone to vulnerability based on the vulnerability history can then be marked for targeted testing or more rigorous reviews.

This research uses traceable software patterns that can be mechanically identified in the source code and determines their relationships with vulnerabilities. Gil et al. [18] developed the concept of traceable patterns that can be automatically (mechanically) recognized. These patterns are related to a specific programming language and have different levels of abstraction. Class-level traceable patterns are called *micro patterns* whereas method-level traceable patterns are called *nano-patterns*. Gil et al. [18] defined 27 micro patterns organized into eight categories with respect to the formal conditions on the structure of Java classes. They capture class properties whereas nano-patterns are method-level patterns and capture properties of methods within a class. Nano-patterns were first introduced by Batarseh in [3]. Singer et al. presented 17 fundamental nano-patterns organized into four groups: calling, object-oriented, control flow and data flow in [47]. Nano-patterns are extracted at the Java method-level and they are defined based on the properties of methods within a class. After the emergence of these traceable patterns, researchers studied their relation with software defects and vulnerabilities [12, 11, 50, 51, 53]. These studies detected bug and vulnerability-prone traceable patterns.

Kim et al. in [24] identified how micro patterns change as software evolves as vulnerabilities are fixed and they identified certain evolution patterns as more bug-prone than others. In our study, we analyzed how micro patterns are changed from vulnerable classes to neutral classes (i.e., where no vulnerability has been found). Other researchers investigated the associations between micro patterns and code smells which is a key indicator

of degrading software quality [15]. Sultana et al. identified associations among the micro patterns and nano-patterns that are more frequent in vulnerable code [50, 53]. In these studies, they computed association rules and the phi-coefficient between each pair of micro and nano-patterns to find their associations both in vulnerable and neutral code. This analysis helps developers to focus on the use of certain patterns together that may negatively affect quality. Moreover, this analysis will assist developers to predict the existence of vulnerabilities. When a developer uses one pattern from the connected pair, he/she should try to avoid the use of its peer pattern (that is closely associated with vulnerable code) in order to reduce the risk of a security violation. If this scenario is infeasible, then these patterns should be targeted for more thorough review or testing.

This research focuses on using traceable patterns to detect the likelihood of security defects. Our research goal was to build a vulnerability prediction model using the relationships between these patterns, software metrics, and vulnerable code. We determined pattern distribution in both vulnerable and neutral code and then identified the patterns that frequently exist in vulnerable code compared to the neutral code (i.e., code where no known vulnerability exists). We also analyzed the association of micro and nano-patterns with vulnerabilities from different angles. We showed how the pair of patterns work for vulnerabilities, how they evolve from one version to another version, and how the patterns perform in vulnerability prediction compared to metrics. In this research, we aimed to build a vulnerability prediction model at the method level that would be able to classify a method as vulnerable or neutral. The motivation of this study is to detect vulnerable code early in the development process so developers can reduce the number of vulnerabilities

released to production. The existence of vulnerability-prone patterns highlights potentially vulnerable code. Being aware of these problematic code constructs, developers can be more cautious when using them during development. When no alternatives exist, developers can program using the constructs but from a defensive standpoint by recognizing that their approach may be troublesome. This strategy will reduce the testing workload by minimizing their search space and suggesting them a lower number of methods that need security related testing. Developers and testers will focus more effort and employ more resources and time on these vulnerable areas executing specific tests that target the types of vulnerabilities observed when using the patterns [49].

The major contributions of this research are as follows:

- **Secure Development:** We analyzed the distribution of micro and nano-patterns in vulnerable code (classes or methods) and code where no vulnerabilities have been reported. Our comparative analysis on the distribution of traceable patterns will discriminate among the patterns regarding their relationship to known vulnerabilities in the source code. This technique will assist developers to be more restrictive in their usage of patterns as well as guide them to inspect their code for potential vulnerabilities.

- **Cost-effective Testing:** The findings of this study will help testers become more focused on the potentially vulnerable areas of code instead of the entire code base. It will save time and effort of the testers and ensure efficient testing for the suspected code.

- **Lower Granularity:** Our vulnerability study is targeted at the class and method level. Doing so will help developers to concentrate on a lower granularity level of code and pinpoint the vulnerable components more easily. The results of this analysis will develop a good understanding of what makes secure code. Developers will be able to ensure reliable system evolution by re-engineering later versions with the proper usage of these code constructs.

- **Comparative Study:** We trained a vulnerability prediction model using the micro and nano-patterns extracted from the vulnerable and neutral code (classes and methods where no known vulnerability exists, we will use the term "neutral" to refer them) data extracted from three different software systems. We executed three machine learning techniques to classify vulnerable code. We trained the model using traditional class and method-level software metrics extracted from the vulnerable and neutral classes and methods respectively from three different systems using the same techniques. We have presented a comparison of performance measures including precision and recall for two types of features (nano-patterns vs method-level metrics and micro patterns vs class-level metrics) in vulnerability prediction. This strategy will help developers to decide features to use in prediction model based on their performance for their respective projects.

- **Improved Vulnerability Prediction:** We developed a vulnerability prediction model using the significant set of nano-patterns and method-level metrics extracted from vulnerable and neutral methods of different software systems. We executed two

machine-learning techniques to classify vulnerable code. This study will help the developers to better asses how vulnerability-prone the code it. It will also assist testers to test code to understand how different nano-patterns and metrics contribute to code that is prone to vulnerabilities. Moreover, the proposed set of metrics and patterns can be used for testing vulnerabilities in any system in general as we have presented their performance results in cross-dataset validation.

- **Generic and Robust Prediction Model:** We have presented experimental results for both within dataset and cross-dataset validation. This study will help developers to evaluate their system in three different ways. First, they can identify significant patterns for their own systems by analyzing vulnerability history and then use these patterns to train a model for vulnerability prediction in later releases. Second, they can extract only the proposed set of metrics and patterns from their vulnerability history and train a machine for future vulnerability prediction. Third, they can test their codebase using a machine that is trained by the proposed set of metrics and patterns from any other system. Our cross-dataset validation shows good performance while using the proposed metrics and patterns as features.

We describe terms used in this study in chapter 2. Chapter 3 presents the existing literature and their findings. In chapter 4, we focus on the research questions covered in the entire study. Chapter 5 describes the procedure followed to answer all the questions. Chapter 6 presents the vulnerability prediction model proposed by this study. We present all the experimental results and their discussion in chapter 7 and 8 respectively. Chap-

ter 9 discusses the limitation of our work and chapter 10 concludes the study with future direction.

CHAPTER 2

BACKGROUND

This chapter describes terms used in this study.

- **Vulnerability:** "A vulnerability is a security exposure that results from a product weakness that the product developer did not intend to introduce and should fix once it is discovered.[1]" Another definition is "An information security 'vulnerability' is a mistake in software that can be directly used by a hacker to gain access to a system or network.[2]" Examples of vulnerabilities are Cross-site scripting (XSS) which allows remote authenticated users to inject arbitrary web scripts or HTML, denial of service that refuses users' requests for a service through the network, injection flaws which occur when untrustworthy data is sent to an interpreter as part of a command or query, and Broken Authentication and Session Management which allows attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities[3].

- **Micro Pattern:** Micro patterns are mechanically recognizable patterns that are captured using formal conditions of the structure of a Java class such as use of inheri-

[1] https://msdn.microsoft.com/en-us/library/cc751383.aspx
[2] https://cve.mitre.org/about/terminology.html
[3] https://www.owasp.org/index.php/Top_10_2013-Top_10

tance, immutability, data management and wrapping, restricted creation, and so on. They are similar to design patterns except that they are defined at a lower level of abstraction [18]. The detailed description of 27 micro patterns is presented in Table 2.1. This table presents the categories of micro patterns and their description.

- **Nano-pattern:** Nano-patterns are method-level traceable patterns. They represent coding actions at the method level frequently used in Java software development [3]. Table 2.2 defines 17 fundamental nano-patterns [47]. Singer et al. supplemented the fundamental nano-patterns by incorporating additional patterns.

- **Micro Pattern Evolution:** Examining the development history of a system explores facts about the software and enables a better understanding of its qualities [24]. In this study, we examined the change history of micro patterns in different versions of the same system. For example, a vulnerability has been detected in foo.java of version 6.0.1, and its pattern type is A, later the class is changed to fix the vulnerability in version 6.0.2, and now its pattern type is B after fixing the vulnerability in that class. This change has been termed as a $A \rightarrow B$ evolution.

- **Phi-Coefficient:** Phi-Coefficient measures the degree of association between two binary variables. It is the linear correlation between postulated underlying discrete univariate distributions of variables X and Y [13]. The strength of the association is determined by following criteria [41]: a small association ($.10 \leq \phi < .30$), a medium association ($.30 \geq \phi < .50$) and a high association ($\phi \geq .50$).

- **Vulnerable code vs Neutral code:** If a vulnerability is fixed in foo.java of version 7.0.2, that class of version 7.0.2 and its later versions are termed as a "neutral" class. On the other hand, the source code of foo.java in the previous versions of 7.0.2 will be termed as a "vulnerable" class. The same definition applies in case of a Java method. If no vulnerability is found in a class or method, that class / method is deemed neutral.

- **Itemset and Support Count:** In our study, we used the concept of *Market Basket Transactions* that relates to a set of data where each row represents a transaction and each column corresponds to an item. An entry in the table will be '1' if that item is purchased in the transaction and '0' otherwise. Each transaction can be represented as a series of 1's and 0's. In our case, each method is represented as a series of 1's and 0's, where '1' indicates a pattern exists in that method and '0' indicates that pattern is absent in the method.

  LET $I = i_1, i_2, \ldots, i_d$ be the set of all items and $T = t_1, t_2, \ldots, t_N$ be the set of all transactions. A subset of items is purchased in each transaction. We define an itemset as a collection of zero or more items. If an itemset contains k items, it will be termed as a $k-$itemset. $Bread, Milk$ is an instance of a $2-$itemset [54], [20]. An important property of an itemset is support count which represents the number of transactions containing a specific itemset [54]. This can be defined as follows:

$$\sigma(X) = |\{t_i, \text{ where } X \subset t_i \text{ and } t_i \subset T\}|$$

  Here, X refers to an itemset which is a subset of t_i.

- **Association Rule Mining:** An association rule is an expression of the form

$$X \rightarrow Y | X \cap Y = \phi$$

Each association rule is measured by its support and confidence. Support indicates the frequency of the rule for a given dataset. Confidence means the frequency of the occurrences of items in Y in the samples that also contain the items in X [54]. They can be mathematically represented as follows:

$$Support, s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N}$$
$$Confidence, c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

Association rule mining can be defined as finding all rules fulfilling the conditions $support \geq minsup$ and $confidence \geq minconf$, where $minsup$ and $minconf$ are the predefined support and confidence thresholds respectively [54]. An exponential number of rules can be generated out of a given dataset. The general procedure for mining association rules can be decomposed into two subtasks: Frequent Itemset Generation and Rule Generation.

- **Frequent Itemset Generation:** The itemsets that satisfy $minsup$ thresholds are known as frequent itemsets. As there is an exponential number of frequent itemsets ($2^k - 1$ frequent itemsets from k items of dataset), the generation of all these itemsets are computationally expensive for large values of k. To solve this issue, the $Apriori$ principal has been developed to reduce the number of candidate itemsets [54].

- **Rule Generation:** In this step, all rules having a high confidence are generated from the frequent itemsets. Frequent itemset having k items can produce up to $2^k - 2$ rules [54]. The general way to produce an association rule out of a set I is to partition it into two non-empty subsets X and Y, where $Y = I - X$ and the rule $X \rightarrow I - X$ satisfies the confidence threshold. For example, $I = \{M_1, M_2, M_3\}$ is a frequent itemset. The relevant candidate rules are: $\{M_1, M_2\} \rightarrow \{M_3\}, \{M_2, M_3\} \rightarrow \{M_1\}, \{M_1, M_3\} \rightarrow \{M_2\}, \{M_1\} \rightarrow \{M_2, M_3\}, \{M_2\} \rightarrow \{M_1, M_3\}, \{M_3\} \rightarrow \{M_1, M_2\}$. In this example, if $\{M_1, M_2\} \rightarrow \{M_3\}$ rule is a low confidence rule, all other rules containing M_3 in their consequent part such as $\{M_1\} \rightarrow \{M_2, M_3\}, \{M_2\} \rightarrow \{M_1, M_3\}$ will be considered low confidence rules and will be ignored.

- **Class-level Software Metrics:** Chowdhury et al. in [8] evaluated a set of software metrics to predict vulnerable files. We selected the set of metrics that are related to classes as listed in Table 2.3. The metrics were extracted using the SciTools Understand code analysis tool. These metrics capture the properties of classes including their structural complexity, dependency on other classes, inheritance properties, cohesiveness among their methods, and so on. The detailed descriptions of these metrics and their purposes are described in Table 2.3. The selected software metrics are related to program complexity and defect-prone code structure. For example, according to Chidamber & Kemerer [4]: 1) Excessive coupling between object classes is detrimental to modular design and prevents reuse. 2) Inter-object class couples should be kept to a minimum. 3) The higher the inter-object class coupling, the more rigorous testing needs to be.

- **Method-level Software Metrics:** Chowdhury et al. in [8] evaluated a set of software metrics to predict vulnerable files. We selected the set of metrics that are related to methods as listed in Table 2.4. These metrics capture the properties of methods including their structural complexity, dependency on other methods, properties of parameters, return conditions, and so on. The detailed descriptions of these metrics and their purposes are described in Table 2.4. The selected software metrics are related to program complexity and defect-prone code structure.

- **Supervised Machine Learning Techniques:** In our problem, we identified a set of features from the program component (i.e., class or method). We collected the feature values from vulnerable classes or methods and neutral classes or methods. In other words, there are two groups: vulnerable and neutral. We collected labeled data, marked as vulnerable or neutral data, and then trained the machine so that it can classify any class or method as vulnerable or neutral based on its learning. In this way, supervised machine learning can help us to predict vulnerable code based on its feature values.

- **Naive Bayes:** Naive Bayes is a simple classification technique based on Bayes' rule of conditional probability that assumes that the value of one feature is independent of the value of another feature. Naive Bayes classifier classifies an instance assuming that all its features independently contribute to the probability and the correlations among the features do not play any role in the classification. Moreover, it requires a small set of training data for classifying properly [30].

- **Logistic Regression:** Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function. In our study, LR calculates the probability of a class or a method being vulnerable or neutral for the given values of the used features. We included this technique as this has been used in earlier studies of vulnerability prediction using CCC metrics [8] and wanted to be able to compare results while building from existing studies.

- **Support Vector Machine:** SVM is a supervised learning model where the training points are separated by the widest possible gap. Then a new point is classified based on its side of the gap. SVM can perform both the linear and non-linear classification mapping the input into higher dimensional space.

Table 2.1

Catalog of Micro patterns [18]

Category	Patterns	Description
Degenerate Class	Designator	An interface with absolutely no members.
	Taxonomy	An empty interface extending another interface.
	Joiner	An empty interface joining two or more superinterfaces.
	Pool	A class which declares only static final fields, but no methods.
	Function Pointer	A class with a single public instance method, but with no fields.
	Function Object	A class with a single public instance method, and at least one instance field.
	Cobol Like	A class with a single static method, but no instance members
	Stateless	A class with no fields, other than static final ones.
	Common State	A class in which all fields are static.
	Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.
	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class
	Sampler	A class with one or more public constructors, and at least one static field of the same type as the class
Containment	Box	A class which has exactly one, mutable, instance field.
	Compound Box	A class with exactly one non primitive instance field.
	Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.
	Record	A class in which all fields are public, no declared methods.
	Data Manager	A class where all methods are either setters or getters.
	Sink	A class whose methods do not propagate calls to any other class.
Inheritance	Outline	A class where at least two methods invoke an abstract method on“this”
	Trait	An abstract class which has no state.
	State Machine	An interface whose methods accept no parameters.
	Pure Type	A class with only abstract methods, and no static members, and no fields
	Augmented Type	Only abstract methods and three or more static final fields of the same type
	Pseudo Class	A class which can be rewritten as an interface: no concrete methods, only static fields
	Implementor	A concrete class, where all the methods override inherited abstract methods.
	Overrider	A class in which all methods override inherited, non-abstract methods.
	Extender	A class which extends the inherited protocol, without overriding any methods.

Table 2.2

Catalog of Fundamental Nano-Patterns [47]

Category	Nano-Patterns
Calling	*noParams*—takes no arguments *noReturn* — returns void *recursive* — calls itself recursively *sameName* — calls another method with the same name *leaf* — does not issue any method calls
Object-Oriented	*objCreator* — creates new objects *fieldReader* — reads (static or instance) field values from an object *fieldWriter* — writes values to (static or instance) field of an object *typeManipulator* — uses type casts or instanceof operations
Control Flow	*straightLine* — no branches in method body *looper* — one or more control flow loops in method body *exceptions* — may throw an unhandled exception
Data Flow	*localReader* — reads values of local variables on stack frame *localWriter* — writes values of local variables on stack frame *arrayCreator* — creates a new array *arrayReader* — reads values from an array *arrayWriter* — writes values to an array

Table 2.3

Class-level Software Metrics

Metrics	Description
AvgCyclomatic [29]	McCabes cyclomatic complexity counts the number of independent paths through a program unit (i.e., number of decision statements plus one). AvgCyclomatic takes the average of this metric for all nested functions or methods in a class.
AvgCyclomaticModified [29]	It is same as cyclomatic complexity except that each decision in a multi-decision structure (switch in C/Java) statement is counted as 1. Average modified cyclomatic complexity is the average of this metric for all nested functions or methods in a class.
AvgCyclomaticStrict [29]	The cyclomatic complexity that adds 1 for every occurrence of logical and (&&) and logical or in conditional expressions.
AvgEssential [29]	It is the cyclomatic complexity after iteratively replacing all well structured control structures (if-then-else and while loops) with a single statement.
CountClassBase	The number of immediate base classes.
CountClassCoupled [4] (Chidamber & Kemerer metric)	The number of other classes to which a class is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B.
CountClassDerived [4] (Chidamber & Kemerer metric)	The number of immediate subclasses. (i.e. the number of classes one level down the inheritance tree from this class).
CountDeclMethodAll [27] (Lorenz & Kidd metric)	The number of methods, including inherited ones.
CountLineCode [14]	The number of lines that contain source code. A line can contain source and a comment. For Classes this is the sum of the CountLineCode for the member functions of the class.
MaxInheritanceTree [4] (Chidamber & Kemerer metric, also known as Depth of inheritance tree (DIT))	DIT is the maximum number of nodes from the class node to the root of the inheritance tree. The root node has a DIT of 0. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity.
PercentLackOfCohesion [4] (Chidamber & Kemerer metric, also known as Lack of Cohesion of Methods (LCOM))	It calculates what percentage of class methods use a given class instance variable. It is computed by taking each instance variable and then divide the number of functions that use it by the total number of functions. Then the average of this value for all instance variables is subtracted from 1. A lower percentage means higher cohesion between class data and methods.
SumCyclomatic [4] (Chidamber & Kemerer, also known as Weighted Methods per Class (WCM))	It is the sum of cyclomatic complexity of all nested functions or methods.

Table 2.4

Method-level Software Metrics

Metrics	Description
CountInput [22]	The number of inputs a function uses plus the number of unique subprograms calling the function. Inputs include parameters and global variables that are used in the function, so Functions calledby + Parameters read + Global Variables read.
CountOutput [22]	The number of outputs that are SET. This can be parameters or global variables. So Functions calls + Parameters set/modify + Global Varibales set/modify.
CountLineCode [14]	The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple metrics.
CountPath [14]	It is the number of unique paths though a body of code, not counting abnormal exits or gotos.
Cyclomatic [29]	McCabes cyclomatic complexity counts the number of independent paths through a program unit (i.e., number of decision statements plus one).
CyclomaticModified [29]	The Cyclomatic Complexity except that each decision in a multi-decision structure (switch in C/Java) statement is not counted and instead the entire multi-way decision structure counts as 1.
CyclomaticStrict [29]	The Cyclomatic Complexity with logical conjunction and logical and in conditional expressions also adding 1 to the complexity for each of their occurrences.
Essential [29]	It is the cyclomatic complexity after iteratively replacing all well structured control structures (if-then-else and while loops) with a single statement.
MaxNesting [21]	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.

CHAPTER 3

RELATED WORK

This chapter presents relevant research on software security, traceable patterns and metrics used for defect and vulnerability prediction.

3.1 Micro Patterns

A traceable pattern can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components [18]. These patterns are based on modules including code fragments, routines, classes and packages. The class-level traceable patterns are micro patterns while the method-level patterns are called nano-patterns. Gil and Maman described the catalog of 27 micro patterns on Java classes and interfaces [18]. Table 2.1 presents the categories of micro patterns and their descriptions. Arcelli and Maggioni suggested a different approach for interpreting micro patterns based on the number of attributes (NOA) and the number of methods (NOM) of a type [28]. Their identified types are not completely aligned to the constraints and definitions of micro patterns [28]. Destefanis et al. identified micro patterns that were more error-prone than others and detected correlations among the patterns [12, 11]. They also found that the classes having no micro pattern are supposed to be more fault-prone than the classes with micro patterns. Kim et al. detected the micro patterns evolution along with the pro-

gram's evolution throughout the development process [24]. Their analysis of micro pattern evolution from one revision to another reported some types of micro pattern evolution to be more bug-prone than others. They performed their micro pattern evolution analysis on three open-source projects: ArgoUML, Columba, and jEdit. The kinds of evolution are almost identical across all the projects. In another study, Singer et al. extracted the association between micro patterns with class name suffix which might be useful for run-time bug detection by the developers [48]. In [53], we previously analyzed the correlation of micro patterns with vulnerable classes and identified certain micro patterns as more prone to vulnerability than others. They also found that some micro patterns are highly associated within vulnerable classes, much more than they are in other classes [53]. We also in [52] presented the comparative study between micro patterns and class-level software metrics using machine learning techniques for vulnerability prediction.

3.2 Nano-patterns

Nano-patterns represent coding actions at the method level frequently used in Java software development [3]. Table 2.2 defines the 17 fundamental nano-patterns [47]. Singer et al. supplemented the fundamental nano-patterns by incorporating additional patterns and classified Java methods using data mining concepts: frequent itemset generation and association rule mining [47]. They applied their work to clustering and categorizing Java methods based on the associated nano-patterns. Sultana et al. used a similar approach to explore the relationship between nano-patterns and vulnerabilities [50, 51]. Deo et al. found certain nano-patterns to be more fault-prone than others [10]. This study was limited

to finding nano-patterns associated with software defects and did not consider security defects.

3.3 Software Security using Metrics

This research contributes new knowledge in the field of software security assessment. Researchers developed a number of software security metrics to assess the security of a software system. These metrics are used to predict vulnerabilities in code based on some predefined threshold values at a certain confidence level. Most of the existing literature focuses on complexity metrics for software quality assessment. Researchers have found that complexity is related with software faults and other problematic issues [42]. Researchers also determined that the fault prediction models based on complexity metrics might be useful for vulnerability prediction and some metrics, such as nesting complexity metrics, are more effective for locating vulnerable code than to locate faulty code [45]. Shin et al. [44, 46] conducted an empirical study to analyze the impacts of complexity metrics on vulnerable and non-vulnerable files. They also determined that vulnerable functions have distinctive characteristics from non-vulnerable functions and from faulty but non-vulnerable functions in terms of code complexity. They showed that fault prediction models based on traditional metrics such as code churn, complexity and fault history provide similar performance in vulnerability prediction across a wide classification threshold [46]. In another study, the authors analyzed how different complexity metrics, code churn, and developer activity metrics can be used to predict vulnerabilities [43]. They showed that these metrics are positively correlated to vulnerabilities. These studies attempted to im-

prove on existing code-level analysis studies for vulnerability prediction. Although the techniques resulted in adequate vulnerability prediction, the methods suffered from high false negative rates [43, 44, 42]. Their study identified development history metrics as stronger indicators of vulnerabilities than code complexity metrics as complexity metrics do not have statistically significant power to predict vulnerabilities [16]. Chowdhury et al. in [7, 8] defined how different code structures cause vulnerable source code. The authors empirically showed that complexity, coupling, and cohesion (CCC) can be effectively used as metrics to detect vulnerability in the early stage of development. They used four alternative data mining and statistical techniques such as C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive-Bayes and compared their performance on vulnerability prediction [8]. Another study on complexity metrics showed that these metrics are good predictors of vulnerabilities between different releases of a project and also between different projects in various fields [33]. According to [32], complexity metrics perform better than coupling metrics in vulnerability prediction. Another study concluded that software metrics can discriminate between vulnerable and non vulnerable functions, but they have no strong correlations with the number of vulnerabilities in the analyzed functions [2].

Structural complexity metrics can exist at the code or design level. Alshammari et al. in [1] introduced seven metrics for assessing the security of an object-oriented class. Chowdhury et al. also proposed three code level metrics: stall ratio, coupling corruption propagation, and critical element ratio to assess the security in a software [6]. The stall ratio measures the ratio of stall statements in loop structures which can be targeted by the attacker to cause a denial of service attack. Coupling corruption propagation is a measure

of propagation of a potential damage across the components of the software. The critical element ratio is a metric for measuring the ways a program or class can be infected by malicious inputs [6]. In another study, authors showed that relative code churns can be used as a predictor of defect density in software systems [35]. Zimmermann et al. [60] demonstrated a large-scale empirical study to evaluate the efficacy of classical metrics such as complexity, churn, coverage, dependency measures, and organizational structure of the company to predict vulnerabilities. In their study, they showed that classical metrics predict vulnerabilities with higher precision but lower recall, and they concluded that vulnerability prediction is not as simple as defect prediction. For the accurate vulnerability prediction domain, usage of the program components are needed to be captured. Younis et al. [58] also determined the performance of software metrics for vulnerability prediction. In their work, they trained their model with exploit data, and considered the vulnerabilities which have been exposed by some attacks.

Neuhaus et al. [36] introduced a new metric called import and function-call to construct the prediction model for vulnerability. The imports in programming languages (e.g. #include in C++, import in Java) allow developers to reuse services provided by other libraries. They successfully leveraged machine learning techniques to predict vulnerabilities with an average precision of 70% and recall of 45%. Walden et al. in [55] employed different code and design based software metrics as vulnerability predictors and used text mining techniques to build vulnerability prediction models for web applications written in PHP. Zhang et al. in [59] proposed a composite algorithm VULPREDICTOR to predict vulnerable files by analyzing software metrics and text features together which is built on

an ensemble of many classifiers. They claimed that VULPREDICTOR performs better than Walden et al. approaches for a wide range of percentages of files to be inspected [59]. In [39], the authors used bag-of-words representation considering a Java source file as a series of terms with associated frequencies. However, the reason why bag-of-approach method or the frequency of several terms is relevant to vulnerability has not been defined clearly. Gopalakrishna et al. [38] measured vulnerability likelihood based on four artifacts of computer programming, including privileged lines of code, error-prone constructs, programming mistakes, and program style. They suggested to weight these metrics based on the extent of their correlations with vulnerabilities, but they did not identify how they could be weighted and how their threshold values would be determined. Nguyen et al. [37] introduced a new prediction model using dependency graphs of a software system based on the relationship among software elements such as components, classes, functions, and variables which can be obtained from static code analyzers. Their proposed model was tested on JavaScript Engine of Firefox.

These studies focused on identifying metrics that quantify software security. Our work is motivated by our goal of improving the performance of existing metrics-based vulnerability prediction models. In addition, we aim to assess security by targeting a specific granularity level. Earlier works did not distinguish between class and method-level metrics. They considered the metrics at the file-level which has no specific granularity. Because a file can be of any size, it cannot represent a program unit that can be compared with another unit. This analysis will lead us to develop effective security measures to assess software security. In this research, we separated class-level and method-level metrics and trained

our model with those metrics separately. Then we compared their prediction accuracy with the accuracy of the model trained with micro patterns and nano-patterns respectively. To reach our target, we investigated the relationship between Java traceable patterns and security vulnerabilities. This relationship has been studied from different viewpoints such as the patterns' distribution, associations among the micro and nano-patterns and patterns' evolution types in vulnerable versus neutral code. For example, if we can identify micro patterns and their associations or evolution types that relate to vulnerable code, we will be able to identify potentially vulnerable areas in newly-implemented code. It may not be a direct causation of vulnerability, but the resulting data can be used to assess security or predict the likelihood of a vulnerability in the source code. Finally, based on the experiments we performed on patterns and metrics, we proposed a vulnerability prediction model using a combined set of patterns and metrics (nano-metrics) that can be used for any system and that performs better in terms of recall.

CHAPTER 4

RESEARCH GOAL

This chapter presents the research goal, research questions that need to be answered in order to reach the goal, and the approach we followed to reach it.

The goal of this research is to develop a vulnerability prediction framework that limits false negative rates using traceable patterns and code-level software metrics. Our motivation is to reduce the number of vulnerabilities in code during implementation and maintenance by using traceable patterns and software metrics. In doing so, developers can be more conscious about vulnerable areas detected by the model and can program defensively as certain patterns arise while coding or some defined thresholds are violated. Testers will concentrate more on the defective classes or methods and spend more time and resources testing them. Both developers and testers can further target problematic areas for testing and thus can save resources allocated for security testing.

4.1 Research Questions

These section presents research questions that this work addresses. We present each question and describe the implications of each and our approach to address the question. We have also refined several questions to further focus the research.

4.1.1 RQ1: What is the relationship between traceable patterns and software vulnerabilities?

Is there any significant difference between traceable pattern distribution in vulnerable and neutral code? This question determines the correlation between traceable patterns and vulnerabilities. We have answered this question by exploring the distribution of patterns in both vulnerable and neutral code. The objective is find interesting patterns that show a significant difference in their frequencies in vulnerable versus neutral code. For example, if the frequency of pattern A is significantly high in vulnerable code compared to neutral, the pattern may exhibit interesting characteristics that may represent vulnerable code by its presence. To find significant presence or absence, we have not only compared the pattern's frequencies in vulnerable versus neutral code, but also conducted hypothesis testing to validate the findings statistically. We formulate our Hypothesis H0 as follows:

H0: *Software vulnerabilities are independent of patterns contained in their source code.*

If we can reject this hypothesis for a particular pattern, we can say that the particular pattern and software vulnerability are related. We do not claim that the presence of the pattern is the cause for the vulnerability rather, we suggest the code to be considered for further testing and review.

How are the traceable patterns associated with each other in vulnerable and neutral code? To answer this research question, we computed the phi-coefficient for each pair of patterns in vulnerable and neutral code using a statistical tool. The connected pairs found in code indicate that they frequently exist together. Developers can avoid using them together

if they understand that their co-existence can indicate vulnerable code. This analysis helps us to determine the joint effect of patterns on vulnerability generation. For example, if we see that $CompoundBox - Immutable$ is a highly associated pair in vulnerable classes whereas they have no association in neutral classes, we can conclude that their joint occurrence has some relationship to the vulnerable code. As the phi-coefficient is a statistical measure, we did not conduct any further statistical test for these findings.

Are the association rules useful in identifying vulnerabilities? Association analysis is a well-known concept in data mining for discovering hidden relationships among large data-sets [20]. The association rules among the nano-patterns were generated for the vulnerable and neutral methods using a data mining tool. These rules explore if there is a dependency among the nano-patterns that may be attributed to vulnerable methods. Knowing that the co-existence of two nano-patterns possesses a high likelihood for vulnerability, they will be prompted to either avoid the co-occurrence of the patterns or rigorously test the code where the patterns exist. For example, if developers know that $jdkClient \rightarrow tailCaller$ is frequent in vulnerable methods, they will carefully use them together or try to avoid their coexistence by replacing them with other types. This methodology for identifying nano-patterns could be integrated into a developer's IDE and used to alert the developer when code contains the risky patterns.

The vulnerable method can be represented when compared to changes made to a method's structure to address vulnerabilities as a series of 1's and 0's as shown in Table 4.1. Each row represents a method that has been changed to fix a particular vulnerability. Each column p_i corresponds to a nano-pattern. An entry in the table will be '1' if that pattern exists

in the method and '0' otherwise. In our problem, we can assume $P = p_1, p_2, \ldots, p_d$ to be the set of all items (i.e., patterns) and $S = s_1, s_2, \ldots, s_N$ be the set of all vulnerable methods. Each method s_i contains a subset of patterns chosen from P. We will define an itemset as a collection of zero or more patterns. Table 4.1 shows that the support count of p_1, p_2 is 2 as they exist together in s_1 and s_4.

Table 4.1

Binary Representation of Vulnerable Methods

	p_1	p_2	p_3	p_4	p_5	p_6
s_1	1	1	0	0	0	0
s_2	1	0	1	1	1	0
s_3	0	1	1	1	0	1
s_4	1	1	1	1	0	0

In our problem, an association rule satisfying a pre-defined threshold of the support count can be generated using a data mining tool (e.g., Weka). If we can generate a set of association rules with support and confidence thresholds, we can correlate the nano patterns with the vulnerability by analyzing those rules. For example, if we see $\{p_1, p_2\} \rightarrow \{p_3\}$ is a rule with high confidence, we can interpret two things. First, we can say that $\{p_1, p_2, p_3\}$ is a frequent itemset indicating the nano-patterns co-exist in many known vulnerable methods. Second, we can confidently determine that the probability of the occurrence of p_3 is high in a vulnerable method if it also contains the patterns p_1 and p_2. As a result, a developer notified of such relationships, will be more conscious when generating any code that contains the three patterns together.

How do traceable patterns evolve from vulnerable code to neutral code? This question serves to strengthen the relationships between patterns and vulnerabilities. For example, if we see that pattern A is frequent in vulnerable code and pattern B is frequent in neutral code, and we get the evolution type $A \rightarrow B$ (i.e. vulnerable code is fixed by removing pattern A and including pattern B), this result strengthens the claim that pattern A is more vulnerability-prone than pattern B.

4.1.2 RQ2: Can traceable patterns better predict vulnerable code than software metrics?

What are the performance measures of traceable patterns in vulnerability prediction? We extracted a total of 24 nano-patterns (including 17 fundamental patterns) for every method in our codebase using the nano-patterns extraction tool [47]. After that, we computed Welch's t-test between the nano-patterns and security vulnerabilities. Welch's Test[1] for unequal variances (aka Welch's t-test) is a modification of a Student's t-test to see if two sample means are significantly different. Using this test, we determined the nano-patterns that have different means between the two groups of methods (i.e., vulnerable and neutral for each system under study). If a nano-pattern has significantly different mean in vulnerable and neutral methods according to Welch's test, we assume that pattern is distributed differently in two groups and consider it as significant for that system. By doing so, we collect the significant patterns out of 24 nano-patterns and then train the machine using the nano-patterns as features. The nano-patterns show a significantly different distribution in two groups, they can better distinguish the vulnerable and neutral methods,

[1] http://www.statisticshowto.com/welchs-test-for-unequal-variances/

and we can ignore unnecessary patterns in the training model. The results of the related experiments evaluate traceable patterns as predictors for software vulnerabilities. We used machine learning techniques to evaluate their performance in prediction. For micro patterns, we directly used all of them for machine learning. We present performance measures including False Negative rate, Recall, Precision, F-measure while using traceable patterns as features for classifying code as vulnerable or neutral.

What are the performance measures of software metrics in vulnerability prediction? The results of the related experiments evaluate software metrics as predictors for software vulnerabilities. We used machine learning techniques to evaluate their performance in prediction and then presented performance measures including False Negative rate, Recall, Precision, F-measure while using software metrics as features for classifying a code as vulnerable or neutral.

4.1.3 RQ3: How do we determine the most significant set of patterns and metrics to build a framework for vulnerability prediction?

What are the significant nano-patterns that can predict vulnerable and neutral methods? This question determines the association of nano-patterns with vulnerable and neutral code. Based on the project and codebase, the set of significant patterns varies and we consider the common set of patterns that show significant relationship with vulnerability for all projects under study. Those patterns have been used as features for vulnerability prediction.

For this question, we used chi-square test of independence. We have developed the following null hypothesis:

H_0 : There is no association between nano-patterns and vulnerabilities.

The alternative hypothesis is:

H_a : There is an association between nano-patterns and vulnerabilities.

What are the significant software metrics that can predict vulnerable and neutral methods? This question determines the association of method-level metrics with vulnerable and neutral code. Based on the project and codebase, the set of significant metrics varies and we considered the common set of metrics that show significant relationship with vulnerability for all projects under study. Those metrics have been used as features for vulnerability prediction.

Then, we combined the set of patterns and metrics (we have defined the combination of a traceable softare pattern and traditional metric as a *nano-metric*) and used them as features for building the proposed vulnerability prediction framework.

4.1.4 RQ4: Is the framework effective at predicting vulnerabilities?

Once we have developed our framework, we determined its effectiveness in predicting code-level vulnerabilities. We used vulnerability data extracted from versions of open source software to train our predictive model. We validated our framework by using the remaining software versions and additional projects under study. First, we used the same system dataset for both training and testing. This experiment shows how these nano-metrics work for vulnerability prediction in later releases of the same system. Second, we do cross-dataset validation where we first train the machine with one system and then test the vulnerabilities for another system. This strategy ensures the effectiveness of the proposed

features whether the machine trained by these features from any system can universally be used for various systems.

The four research questions were developed targeting our motivations that led to the overall research goal. The first research question helps to analyze the relationship between vulnerabilities and patterns from different perspectives. The results of this question provides guidelines to developers and testers about using these patterns in code. The second research question enables a comparative study between patterns and metrics so that developers can choose appropriate metrics for predicting vulnerabilities in their own systems. The third and fourth research questions result in proposing and evaluating a new set of metrics, combining both the nano-patterns and metrics, that can perform better than the existing metrics for any system alone. Although the motivations of the research questions are different from each other, their results have guided us to fulfilling our research goal. For example, in the first research question, we found that some patterns are significantly present in vulnerable and neutral code which motivated us to assume that they can be used to build a predictor which may work better than current predictors (using metrics). In the second research question, we built separate predictors using patterns and metrics and compared their performance. We found that micro patterns do not perform better than metrics, whereas nano-patterns perform better than metrics (when considering certain criteria [e.g., false negatives]). So, in our third research question, we proposed nano-metrics consisting of nano-patterns and software metrics and evaluated their performance. Chapter 7 presents the results of the research questions.

CHAPTER 5

EXPERIMENTAL SETUP

This chapter presents the experimental setup including systems and tools information, the approach of vulnerability collection, data extraction to get the answers of the research questions.

5.1 Vulnerability Collection and Tools

We used three different projects: Apache Tomcat[1], Apache CXF[2]; and the Stanford Securibench dataset[3] to evaluate our research questions. The statistics of the systems used for RQ1 are presented in Table 5.1. Table 5.2 shows the systems used for the other research questions. Apache projects were used for two major reasons. First, all vulnerability reports including pointers to the classes affected by a vulnerability for every release are documented on the Apache website. Second, we needed Java based systems because micro patterns and nano-patterns are defined for Java classes and methods respectively. Apache Tomcat consists of about half a million lines of code and more than 3000 classes. The vulnerabilities reported on the site are listed in Table 5.3. This table shows the reported vulnerability types across two major releases of Apache Tomcat. We considered 14 ver-

[1]https://tomcat.apache.org/
[2]http://cxf.apache.org/
[3]https://suif.stanford.edu/ livshits/securibench/stats.html

sions of Tomcat-6, 18 versions of Tomcat-7, 7 versions of Tomcat-8, and 12 versions of Apache CXF. The reported vulnerabilities are detected across these versions as shown in Table 5.4 and Table 5.5. In most cases, the versions of Apache Tomcat 6 and 7 are identical in terms of the source code; we were mainly interested in the classes and methods where the vulnerabilities are identified and fixed in a subsequent release. The source code of Tomcat is contained in the Apache Tomcat Archives[4], and the source code of CXF is in Apache CXF Archives[5]. These repositories were used to download the source files across the versions specified. Stanford SecuriBench is a set of open source programs used to test static and dynamic security tools [25, 26]. We conducted the experiment for traceable patterns on the J2EE web applications known as Pebble 1.6-beta1, Personalblog 1.2.6 and Roller 0.9.9 in the Stanford SecuriBench dataset.

Table 5.1

Systems in Study for RQ1

	Systems	**Versions**	**Files/Classes**	**Methods**
Apache	Tomcat (Releases 6/7/8)	Any Version	2800 (approx.)	10296 (approx.)
Stanford	Personalblog	1.2.6	39	33
	Roller	0.9.9	276	2857
	Pebble	1.6-beta1	333	N/A

5.1.1 Tools

This section describes tools used to collect and analyze data for our studies.

[4]http://archive.apache.org/dist/tomcat/
[5]http://archive.apache.org/dist/cxf/

Table 5.2

Systems in Study for RQ2-RQ4

	Systems	Versions	Files/Classes	Methods
Apache	Tomcat (Releases 6/7)	Any version	2800 (approx.)	10296 (approx.)
	CXF	3.1.10	737	26366
Stanford	Personalblog	1.2.6	39	33
	Roller	0.9.9	276	2857
	Pebble	1.6-beta1	333	N/A
Other	Evaluation Dataset	Apache Tomcat 9.0.0.M1, 9.0.0.M9, 9.0.0.M11, 9.0.0.M15, 9.0.0.M17, 9.0.0.M18, 9.0.0.M20, Apache Struts 2.2.3, Apache Struts 2.3.4, and Apache Commons-Compress-1.4	N/A	2468

Table 5.3

Vulnerabilities

Tomcat-6	Tomcat-7	Tomcat-8	Apache CXF	Stanford Securibench
Information disclosure	Security Manager bypass	Security Manager bypass	Denial of Service (DoS)	Cross-Site Scripting
Security Manager bypass	Request Smuggling	Request Smuggling	Authentication bypass	HTTP Response Splitting
Request Smuggling	Denial of Service	Denial of Service	SOAP Action spoofing attacks	SQL Injections
Information disclosure	Information Disclosure	Information Disclosure	Wrapping attack	Path Traversal
Frame injection in documentation Javadoc	Remote Code Execution		XML External Entity (XXE) Injection	Log Forging.
Session fixation	Session fixation		POODLE attack	
DIGEST authentication weakness	Bypass of CSRF prevention filter		XML Encryption flaw	
Denial of Service	DIGEST authentication weakness		DTD based XML attacks	
Bypass of security constraints	Bypass of security constraints			
Bypass of CSRF prevention filter	Privilege Escalation			
Authentication bypass and information disclosure	Multiple weaknesses in HTTP DIGEST authentication			
Multiple weaknesses in HTTP DIGEST authentication	Security constraint bypass			
Cross-site scripting	Remote Denial Of Service			
SecurityManager file permission bypass	Cross-site scripting			
Remote Denial Of Service and Information Disclosure	SecurityManager file permission bypass			
Information disclosure in authentication headers				
Arbitrary file deletion and or alteration on deploy				
Insecure partial deploy after failed undeploy				
Unexpected file deletion in work directory				

Table 5.4

Affected Versions for RQ1

Systems	Affected Versions	Vulnerable Classes	Vulnerable Methods
Tomcat-6	6.0.16, 6.0.18, 6.0.26, 6.0.27, 6.0.29, 6.0.30, 6.0.32, 6.0.33, 6.0.35, 6.0.36, 6.0.37, 6.0.39, 6.0.41, 6.0.43	104	124
Tomcat-7	7.0.6, 7.0.10, 7.0.11, 7.0.16, 7.0.20, 7.0.21, 7.0.22, 7.0.27, 7.0.29, 7.0.32, 7.0.39, 7.0.42, 7.0.47, 7.0.50, 7.0.52, 7.0.53, 7.0.54, 7.0.57	108	106
Tomcat-8	8.0.0-RC1, 8.0.0-RC5, 8.0.1, 8.0.3, 8.0.5, 8.0.8, 8.0.15	55	21
Stanford Securibench	Blueblog 1.0, Pebble 1.6-beta1, Personalblog 1.2.6 and Roller 0.9.9	91	151

Table 5.5

Affected Versions for RQ2-RQ4

Systems	Affected Versions	Vulnerable Classes	Vulnerable Methods
Tomcat-6	6.0.16, 6.0.18, 6.0.26, 6.0.27, 6.0.29, 6.0.30, 6.0.32, 6.0.33, 6.0.35, 6.0.36, 6.0.37, 6.0.39, 6.0.41, 6.0.43	104	124
Tomcat-7	7.0.6, 7.0.10, 7.0.11, 7.0.16, 7.0.20, 7.0.21, 7.0.22, 7.0.27, 7.0.29, 7.0.32, 7.0.39, 7.0.42, 7.0.47, 7.0.50, 7.0.52, 7.0.53, 7.0.54, 7.0.57	63	106
Apache CXF	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.2, 2.7.7, 2.7.9, 2.7.10, fediz-core-1.2.0, 3.0.2, 3.0.6, 3.1.8	33	45
Stanford Securibench	Blueblog 1.0, Pebble 1.6-beta1, Personalblog 1.2.6 and Roller 0.9.9	91	151
Evaluation Dataset	Apache Tomcat 9.0.0.M1, 9.0.0.M9, 9.0.0.M11, 9.0.0.M15, 9.0.0.M17, 9.0.0.M18, 9.0.0.M20, Apache Struts 2.2.3, Apache Struts 2.3.4, and Apache Commons-Compress-1.4	N/A	67

Vulnerability Extraction Tool: We used a static analysis tool called Early Security Vulnerability Detector - ESVD[6] to identify vulnerable classes and methods for the projects in Stanford Securibench. The reason behind using ESVD instead of other tools is that it was shown to have fewer false positives in its results with higher precision and recall for the projects within Stanford Securibench in existing studies[7]. The tool has an Eclipse plugin that was used to analyze the projects.

Traceable Pattern Extraction Tool: The micro patterns were extracted using a pattern extraction tool developed by Gil and Maman [18]. This command line tool is available at Maman's webpage[8]. Singer et al. in [47] developed a tool to detect nano-patterns in Java byte code. This tool provides the list of all methods and their associated nano-patterns. We used another tool, JiraExtractor [10] developed at Mississippi State University, that extracts the methods modified for each revision number involved in fixing a vulnerability. This tool then pulls nano-patterns of those methods from the database that contains the

[6]https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/

[7]http://docplayer.net/1619013-Early-vulnerability-detection-for-supporting-secure-programming.html

[8]http://www.cs.technion.ac.il/ imaman/mp/download.html

nano-pattern information for the software versions that is part of the analysis. JiraExtractor actually incorporates the nano-pattern tool developed in [47] to extract the nano-pattern information.

Metrics Extraction Tool: We used SciTools Understand 4.0[9] to extract class-level and method-level software metrics in our research.

Other Tools: The phi-coefficients were measured using SPSS [10]. We used WEKA 3.8[11] to extract association rules and build the vulnerability prediction model.

5.1.2 Vulnerability Collection

This section presents our approach to data collection by presenting the systems that were used in the study.

5.1.2.1 Apache Tomcat

We collected Apache Tomcat vulnerability reports that provide the information about the vulnerability type, its CVE (Common Vulnerabilities and Exposures)[12] id, affected versions, revision number, fixed version, and the severity level of the vulnerabilities fixed in the identified versions. For example, if a vulnerability affects the versions 6.1, 6.2, 6.3 and is fixed in 6.4, we consider 6.3 as the last affected version. By doing so, we collected the last affected code versions for the listed vulnerabilities as shown in Table 5.4 and Table 5.5. Table 5.4 shows the collected versions for research question 1 and Table 5.5 shows the collected version for studying the other research questions. We considered 6.0.45 as the

[9]http://www.scitools.com
[10]http://www-01.ibm.com/software/analytics/spss/products/statistics/index.html
[11]http://www.cs.waikato.ac.nz/ml/weka
[12]https://cve.mitre.org/

last non-affected version in release 6, 7.0.69 as the last non-affected version in release 7, and 8.0.33 as the last non-affected version in release 8 at the time of study for research question 1. But for the remaining research questions, we considered 6.0.48 as the last non-affected version in release 6, 7.0.75 as the last non-affected version in release 7, and 8.0.69 as the last non-affected version in release 8. The reason behind it is that the vulnerabilities that we considered for each major release are not available in the last version of that release as they were already fixed. Although some other vulnerabilities may still exist in this last version, we were only interested in the vulnerabilities that were reported in that release. Then, we downloaded the code for all *affected* and *non-affected* versions listed in Table 5.4 for the first research question and Table 5.5 for the other research questions. For example, in Figure 5.1, the Denial of Service vulnerability (CVE-2014-0075) was fixed in revision 1578341 of version 7.0.53. If we follow the link to revision number[13], we will get the list of classes modified to fix the vulnerability as shown in Figure 5.2. So our collected data is as follows:

- CVE id
- Vulnerability Type
- Last Affected Version
- Revision No
- List of Classes modified to fix

[13]http://svn.apache.org/viewvc?view=revision&revision=1578341

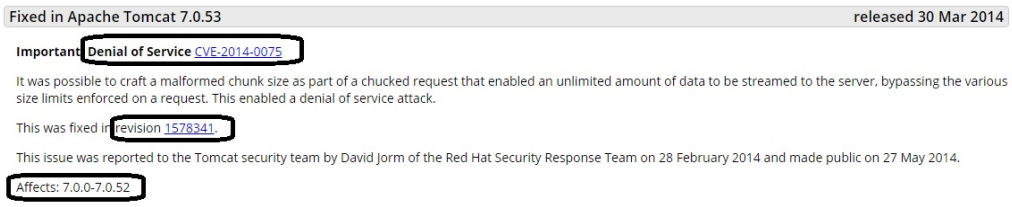


Figure 5.1

Apache Tomcat Security Page

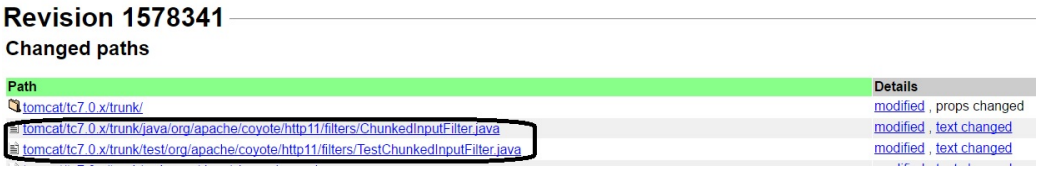


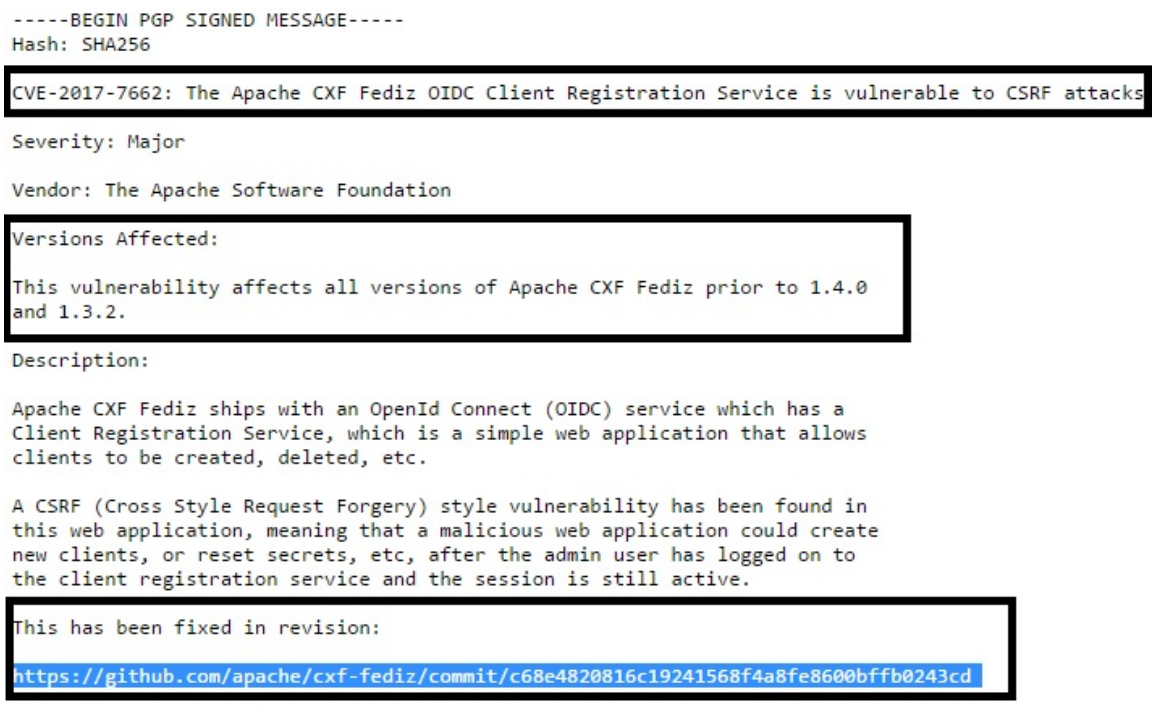
Figure 5.2

Affected class names for a vulnerability in Apache Tomcat

5.1.2.2 Apache CXF

The vulnerability reports of Apache CXF provide the information about the vulnerability type, its CVE id, affected versions, revision number and fixed version as shown in Figure 5.3. The reports also provide the list of classes that were modified for fixing the respective vulnerability as in Figure 5.4. According to Figure 5.3, CSRF attacks (CVE-2017-7662) affected the versions of Apache CXF Fediz prior to 1.4.0 and 1.3.2. If we follow the link[14], we get the list of class files for fixing the vulnerability as shown in Figure 5.4. On the other hand, we considered 3.1.10 as the neutral version for Apache CXF.

[14]https://github.com/apache/cxf-fediz/commit/c68e4820816c19241568f4a8fe8600bffb0243cd



```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

CVE-2017-7662: The Apache CXF Fediz OIDC Client Registration Service is vulnerable to CSRF attacks

Severity: Major

Vendor: The Apache Software Foundation

Versions Affected:

This vulnerability affects all versions of Apache CXF Fediz prior to 1.4.0
and 1.3.2.

Description:

Apache CXF Fediz ships with an OpenId Connect (OIDC) service which has a
Client Registration Service, which is a simple web application that allows
clients to be created, deleted, etc.

A CSRF (Cross Style Request Forgery) style vulnerability has been found in
this web application, meaning that a malicious web application could create
new clients, or reset secrets, etc, after the admin user has logged on to
the client registration service and the session is still active.

This has been fixed in revision:

https://github.com/apache/cxf-fediz/commit/c68e4820816c19241568f4a8fe8600bffb0243cd
```

Figure 5.3

Apache CXF Security Page

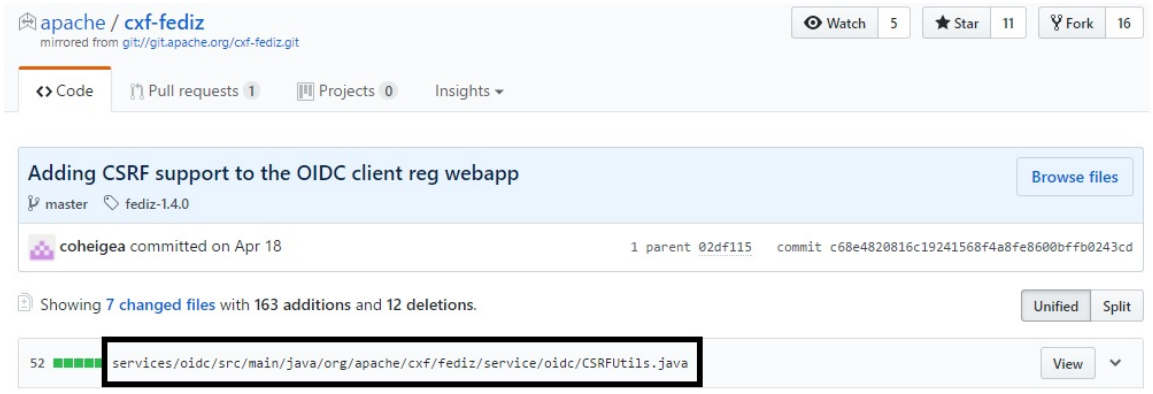


Figure 5.4

Affected class names for a vulnerability in Apache CXF

5.1.2.3 Stanford Securibench

We used a static analyzer tool called Early Security Vulnerability Detector - ESVD[15] to identify the vulnerable classes and methods of the projects: pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9. The tool has an Eclipse plugin that we installed in the Eclipse IDE to analyze the projects. We obtained 23 vulnerabilities in pebble 1.6-beta1, 34 vulnerabilities in personalblog 1.2.6, and 207 vulnerabilities in roller 0.9.9. The vulnerability types were cross-site scripting, http response splitting, sql injections, path traversal, and log forging. We then explored the classes and methods that are associated to these types of vulnerabilities. To that end, ESVD detected in total, 12, seven, and 72 vulnerable classes in pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9 respectively. There are 12 and 139 vulnerable methods in personalblog 1.2.6 and roller 0.9.9 respectively as detected by ESVD.

5.2 Data Extraction

This section presents our approach to data extraction.

5.2.1 Micro pattern Extraction

We downloaded the affected versions listed in Table 5.5 from the Apache projects. The micro pattern tool is activated via the command line and accepts the class name or a jar file as input and extracts all micro patterns identified in that class or classes in the jar file. If a particular micro pattern exists in that class, the entry is '1'; otherwise, it is '0'. We collected the micro pattern data for 104 vulnerable classes in Tomcat-6, 63

[15]https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/

vulnerable classes in Tomcat-7, 38 vulnerable classes in Tomcat-8, 33 classes in CXF, and 91 classes in Stanford Securibench. We then analyzed micro patterns from the source code of the neutral version for each project. For RQ1, there were 2211 classes from version 6.0.45, 2789 classes from version 7.0.69, 2972 classes from version 8.0.33, and 551 from Stanford. For RQ2 and RQ3, we collected micro patterns data for 717 classes of Tomcat-6, 827 classes of Tomcat-7, 737 of CXF, and 364 of Stanford.

5.2.2 Software Metrics Extraction

We used a commercial tool from SciTools called Understand 4.0 to compute the metrics from the source code. We created a separate project in Understand for every version of every project and ran a scheduler after specifying the metrics we needed. We executed the scheduler for each project and generated separate csv files containing the class-level and method-level metrics as listed in Table 2.3 and in Table 2.4 for that project. Then we selected the classes and methods that were recorded as vulnerable and stored their metrics in separate files.

5.2.3 Nano-pattern Extraction

We used the JiraExtractor tool to get the list of methods changed for a revision [10]. The tool fetches the nano-patterns of those methods using Singer's nano-patterns extraction tool. For example, we found the list of vulnerabilities which affected a specific version of Apache Tomcat listed in **Affected Versions** column of Table 5.5 and then used the nano-patterns tool to dump all methods and their nano-patterns of that version in the database. After that, the JiraExtractor tool gets all the revision numbers for that version and fetches

the list of all methods in that version that have been revised to fix the vulnerability. Finally, the tool extracts the nano-patterns of those methods from the database table. For RQ1, we found 124, 106, 21, 151 vulnerable methods in Tomcat-6, Tomcat-7, Tomcat-8, and Stanford respectively as presented in Table 5.4. For RQ2 and RQ3, we found 124, 106, 45, 151, 67 vulnerable methods in Tomcat-6, Tomcat-7, Apache CXF, Stanford, and Evaluation dataset respectively as presented in Table 5.5. Then, we collected nano-patterns from the source code of the neutral versions by using the nano-patterns tool by Singer. So for the neutral versions, we had nano-patterns of 8645 methods in Tomcat-6, 10296 methods in Tomcat-7, 13165 methods in Tomcat-8, 26366 methods in CXF, 2734 methods in Stanford, and 2401 methods in Evaluation dataset.

5.3 Vulnerability Prediction

We developed vulnerability predictors by using three machine learning techniques. Waikato Environment for Knowledge Analysis (WEKA) is a popular, open source toolkit implemented in Java for machine learning and data mining. We used WEKA 3.8 for our study. The parameters for each of the techniques were initialized with the default settings for WEKA.

As our dataset was not balanced, we needed to create a balanced dataset consisting of the same number of vulnerable and neutral classes or methods. Earlier studies either considered under-sampling of the majority category or over-sampling of the minority category. The problem with under-sampling is that information may be lost. In the case of over-sampling the minority category, i.e., duplicating the instances representing vulnerable

files can make the system biased to the vulnerable class when the size of vulnerable classes is too small. In this research, we applied *ClassBalancer* filter in WEKA 3.8[16]. This filter re-weights the instances in the data so that each category has the same total weight. This filter changes only the weight of the first batch of the data.

We ran three machine learning algorithms separately for the nano-patterns and the method-level metrics. We also ran them separately for the micro patterns and the class-level metrics feature set. For each algorithm, we used 10-fold cross-validation to ensure that the trained model will work accurately for an unknown dataset in practice [57]. In 10-fold validation, the sample dataset is randomly partitioned into 10 subsamples of equal size. Of the 10 subsamples, a single subsample is retained as the validation data or test data, and the remaining $10 - 1$ subsamples are used as training data. The process is then repeated 10 times considering each of the 10 subsamples exactly once as the test data. After 10 runs, the average is taken on 10 folds to produce a single estimation. We trained the model with each project separately and predicted the test data of the respective project. The result of this experiment shows how accurately a model trained with historical data can predict vulnerable classes or methods of later releases. For the RQ3, we also performed cross-dataset validation, where we trained the model with one dataset and tested it for another dataset.

5.3.1 Performance Measures

The following performance measures are used to evaluate the model.

[16]http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

- **Precision:** Precision for a vulnerable method or class prediction can be defined as the ratio of the number of predicted vulnerable methods or classes that are actually vulnerable to the total number of methods or classes retrieved as vulnerable [57]. It is also known as correctness of the prediction model. We used this measure in our experiment in order to show how much irrelevant data are drawn by the model while classifying a vulnerable or a neutral method or a class. The formula used for computing precision is as follows:

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

- **Recall:** Recall of a vulnerable method or class prediction is defined as the ratio of the number of predicted vulnerable methods or classes that are actually vulnerable to the total number of vulnerable methods or classes in the system. It can be defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

Recall is more significant than precision in the case of vulnerability prediction. Higher recall for vulnerable methods or class prediction ensures that it has detected most of the vulnerable methods or classes and very few of them remain undetected. There is a trade-off between recall and precision. For example, if all the methods or classes predicted as vulnerable by the model are actually vulnerable, its precision will be 100 percent although it does not guarantee that no vulnerable method

or class remains as undetected or wrongly predicted as neutral. Similarly, precision will degrade in case a model wrongly predicts many neutral methods or classes as vulnerable although it can successfully detect all the vulnerable methods or classes and make 100 percent recall. It will make the predictor inefficient and useless.

- **False Negative Rate (Miss Rate):** The False Negative rate indicates the percentage of vulnerable methods or classes that are wrongly predicted as neutral. It is more significant in our case as predicting a vulnerable method or a class as neutral may cause serious security failure compared to the wrongly predicted neutral methods or classes. In fact, recall is $1 - FNRate$. With the increase of FN rate, recall will decrease.

$$FNRate = \frac{FN}{TP + FN} \tag{5.3}$$

- **False Positive Rate (Fall-out):** FP Rate indicates the proportion of neutral methods or classes wrongly predicted as vulnerable compared to the total number of actual neutral methods or classes in the system. The formula is as follows:

$$FPRate = \frac{FP}{FP + TN} \tag{5.4}$$

This measure is important in case of cross-dataset validation. As we applied classbalancer only for the training set and not for the testing set. In the case of cross-dataset validation, precision becomes extremely poor. In those cases, we focus on the FP

Rate to see the rate of wrongly predicted neutral methods compared to their actual number.

- **F-measure:** F-measure can be interpreted as a weighted average of precision and recall [57]. It gives equal importance to both precision and recall by considering their harmonic mean [57]. The formula for F-measure is defined as follows [5]:

$$F_\beta = \frac{(\beta^2 + 1) Precision \times Recall}{\beta^2 \times Precision + Recall} \tag{5.5}$$

  In the above equation, β controls a balance between Precision and Recall. $\beta = 1$ makes F_1-measure equivalent to the harmonic mean of Precision and Recall. $\beta > 1$ makes F-measure more recall-oriented and $\beta < 1$ makes it more precision oriented.

- **ROC:** ROC (Receiver Operating Characteristic) curve shows the trade-off between the True Positive Rate and the False Positive Rate. The area under ROC curve is a measure for evaluating the performance of the binary classifier in terms of TP and FP rate. Area close to 1 shows high-performance model and area about 0.5 shows low-performance model [32]. Increase in Recall (TP Rate) also results in increase in FP Rate. Therefore, ROC measure helps to track the optimum point where the metric performs well with respect to the TP Rate and FP Rate.

CHAPTER 6

VULNERABILITY PREDICTION MODEL

This chapter presents the research plan developed to evaluate first two research questions; RQ1: What is the relationship between traceable patterns and software vulnerabilities? and RQ2: Can traceable patterns better predict vulnerable code than software metrics? Figure 6.2 presents the workflow for accomplishing the research goal. After completing the experiments related to RQ1 and RQ2, we developed an architecture for a vulnerability prediction model using nano-patterns and method-level metrics. Based on the results of first two research questions, we realized that micro patterns are not a good predictor of vulnerabilities and class-level granularity does not work better than method-level granularity. Therefore, we decided to consider method-level granularity and proposed a model based on nano-patterns and method-level software metrics. We evaluated the effectiveness of the model using vulnerability data of real-world applications.

6.1 Overview of the Prediction Model

We have devised a model to predict vulnerabilities using nano-patterns and method-level metrics. The model comprises several phases. The dataflow diagram of the model is shown in Figure 6.1.

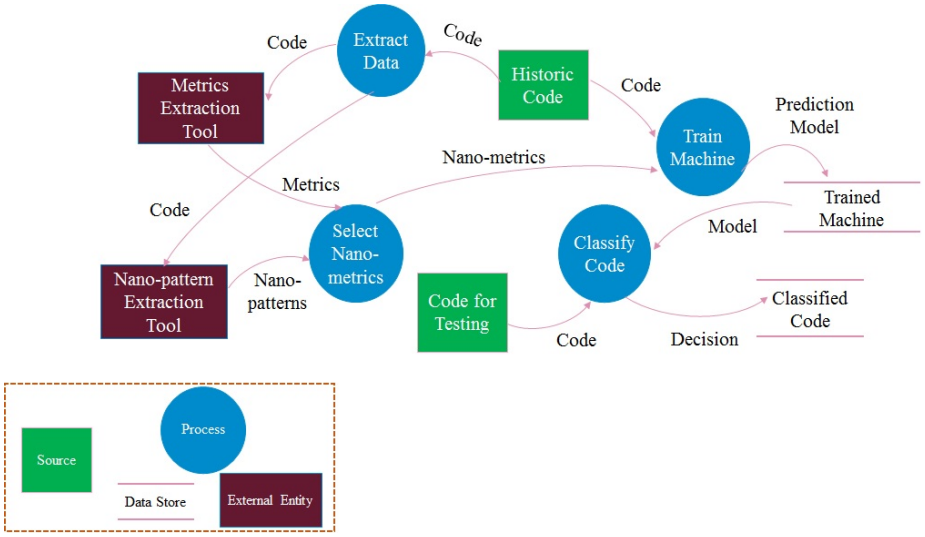


Figure 6.1

Data Flow Diagram of the Research Plan

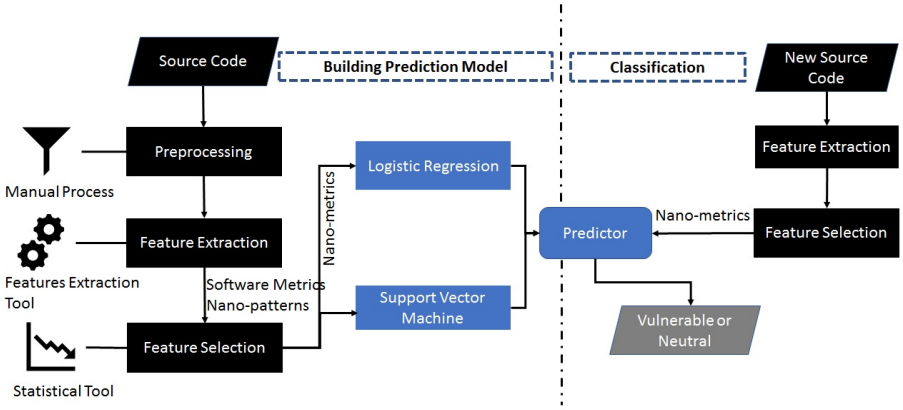


Figure 6.2

Overall Framework of the Research Plan

6.1.1 Building the Prediction Model

6.1.1.1 Preprocessing

We collected a set of vulnerable and neutral methods for the systems under study. We also cleaned the set of methods by removing duplicate instances.

6.1.1.2 Feature Extraction

We extract nano-patterns from the vulnerable and neutral methods using the nano-patterns extraction tool [47]. The tool extracts all 24 nano-patterns from the given methods. After that, we collect method-level metrics as specified in Table 2.4 for those methods using the Understand tool.

6.1.1.3 Feature Selection

We extract data consisting of nano-patterns and method-level software metrics. To initiate creation of the model, we need to perform statistical tests to get correlations of nano-patterns and metrics with vulnerabilities. Are goal was to identify significant patterns and metrics that show significantly different distributions in vulnerable and neutral code in all projects under study. We consider the common set of patterns and metrics in all projects and then used them to build a vulnerability prediction model.

6.1.1.4 Build Predictor

We build a predictor using machine learning techniques to classify vulnerable and neutral code. We combined selected features (nano-metrics) to build this predictor.

6.1.2 Classification

6.1.2.1 Feature Extraction and Selection

We extract nano-patterns and metrics using the tools described previously. After extracting the patterns and metrics, we chose the selected features (significant set of patterns and metrics or nano-metrics) for the test dataset.

6.1.2.2 Model Evaluation

These features are fed to the predictor which has already been trained using machine learning techniques. The predictor then classifies the new piece of code as vulnerable or neutral. We evaluate the effectiveness of the model using the measures including Recall, Precision, F-measure, and AUC (Area under ROC curve).

6.2 Methodology

6.2.1 System Selection

We experimented our proposed model on four different projects: Apache Tomcat (Release 6 and 7)[1], Apache CXF[2], the Stanford Securibench dataset[3] and the evaluation dataset. The evaluation dataset consisted of a set of vulnerable and neutral methods from Apache Tomcat 9.0.0.M1, 9.0.0.M9, 9.0.0.M11, 9.0.0.M15, 9.0.0.M17, 9.0.0.M18, 9.0.0.M20, Apache Struts 2.2.3, Apache Struts 2.3.4, and Apache Commons-Compress-1.4.

[1] https://tomcat.apache.org/
[2] http://cxf.apache.org/
[3] https://suif.stanford.edu/ livshits/securibench/stats.html

6.2.2 Features Extraction

In this section, we describe how we collected features (nano-patterns and method-level metrics) from the vulnerable and neutral methods of the systems under study.

6.2.2.1 Nano-Patterns Extraction

We identified the affected versions of Apache Tomcat as listed in the **Affected Versions** column of Table 5.5. After that, we used the nano-patterns tool developed by Singer [47] to dump all methods and their respective nano-patterns in those versions in a database. Then we collected the name of the affected methods inside every class that had been revised to remove a vulnerability and finally stored those methods and their nano-patterns in a separate file. We found 124, 106, 45, 151, and 67 vulnerable methods in Tomcat-6, Tomcat-7, Apache CXF, Stanford, and Evaluation dataset respectively (presented in Table 5.5). Then we collected nano-patterns from the neutral methods of the neutral versions using the nano-patterns extraction tool. We obtained nano-patterns for 8645 methods in Tomcat-6, 10296 methods in Tomcat-7, 26366 methods in CXF, 2734 methods in the Stanford applications, and 2401 methods in the evaluation set.

6.2.2.2 Software Metrics Extraction

SciTools Understand 4.0[4] was used to compute the source code metrics. We first created a project in the Understand tool for every version of every system and ran a scheduler to extract the specific metrics needed. This process took almost 30 seconds for every version of a project. We generated separate csv files for all different versions containing the

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

method-level metrics of that version. Finally, we separated our vulnerable methods from that file and stored their metrics in a separate file. We followed the same procedure for collecting metrics of the neutral methods of the systems under study.

6.2.3 Feature Selection

6.2.3.1 Finding Significant Nano-patterns

We extracted 24 nano-patterns (including 17 fundamental patterns) for every method in our codebase using the nano-patterns extraction tool [47]. After that, we computed the chi-square test of significance between the nano-patterns and security vulnerabilities to get the set of patterns that show a significant relationship with vulnerable methods.

6.2.3.2 Finding Significant Metrics

Welch's Test[5] for unequal variances (aka Welch's t-test) is a modification of a Student's t-test used to determine if two sample means are significantly different. Using this test, we determined the metrics that have different means between the two groups of methods (i.e., vulnerable and neutral for each system under study). If a metric has significantly different mean in vulnerable vs neutral methods according to Welch's test, we assumed that metric is distributed differently in two groups and considered it significant for that system. By showing a different distribution between the two groups, they better distinguish the vulnerable and neutral methods and we can ignore unnecessary metrics in the training model.

[5]http://www.statisticshowto.com/welchs-test-for-unequal-variances/

6.2.3.3 Select Common Nano-metrics

After getting the list of patterns and metrics for every system under study, we considered the patterns and metrics that are common in all systems. We discarded other patterns and metrics and did not consider them as features for the prediction model. The common set of nano-patterns and metrics selected for the study has been termed as "Nano-metrics".

6.2.4 Building Prediction Model

We developed a prediction model using the nano-metrics. In this section, we describe the steps followed to build the model.

We employed two machine-learning algorithms (Logistic Regression and Support Vector Machine) to build the prediction model and classify the vulnerable and neutral methods in all the systems under study. We followed two types of experiments for predicting vulnerabilities. In the first type, we trained the model with each dataset separately and used 10-fold cross-validation to ensure that the trained model would work accurately for the later releases of the dataset in practice [57]. We trained the model with each project separately and predicted the test data of the respective project. The result of this experiment showed how accurately a model trained with historical data can predict vulnerable methods of later releases. In the second type, we trained the model with one system and then used another system as test dataset (cross-dataset validation). The goal of this approach was to determine if the trained model would work for different systems. For example, if we trained the model with Tomcat 6, we tested the model using the vulnerability data from Tomcat 7, Apache CXF, Stanford, and the evaluation dataset.

CHAPTER 7

EXPERIMENTAL RESULTS

This chapter presents the experimental results.

7.1 RQ1: What is the relation between traceable patterns and software vulnerabilities?

7.1.1 Is there any significant difference between traceable pattern distribution in vulnerable and neutral code?

We found that some micro patterns exist frequently in vulnerable classes whereas others are completely absent. In the same way, some micro patterns are frequent in neutral classes and others are generally not present in them (for the target systems we analyzed). Our significant observations are as follows:

- Vulnerable classes in all versions of releases 6, 7, and 8 do not contain the patterns Box, $Canopy$, $RestrictedCreation$, $PureType$, $Extender$, $DataManager$, $Trait$, and $StateMachine$ whereas they are present in neutral classes as shown in Figure 7.1, Figure 7.2, and Figure 7.3. Therefore, we can recognize them as safe patterns to be used in code.

- There are some micro patterns that have a statistically significant presence in neutral classes compared to their presence in vulnerable classes as shown in Figure 7.1,

Figure 7.2, and Figure 7.3. They are $CompoundBox$, $Immutable$, $Implementor$, $Overrider$, $Sink$, $Stateless$, $FunctionObject$, and $LimitedSelf$.

- There are certain micro patterns which are significantly present in vulnerable classes but are almost absent in neutral classes. They are $Outline$ and $AugmentedType$. The percentage of $Outline$ pattern in vulnerable versions of release 7 is 10.2 percent and in neutral version of release 7 is 0.97 percent. $Outline$ pattern is 9.09 percent in vulnerable versions of release 8, and 1.11 percent in neutral version of release 8. $AugmentedType$ pattern is 2.78 percent in vulnerable versions of release 7, and 0.68 percent in neutral version of release 7. $AugmentedType$ pattern is 3.64 percent in vulnerable versions of release 8, and 0.64 percent in neutral version of release 8. $CommonState$ pattern is available in version 6, but its uses have been reduced in the later versions. The percentage of $CobolLike$ pattern is higher in vulnerable versions than in neutral versions.

- Other micro patterns such as $Useless$, $Sampler$, $PseudoClass$, $Joiner$, $Designator$, $Record$, $Taxonomy$, and $Recursive$ are almost absent in both the vulnerable and neutral classes in all the versions of the target systems.

- In the Stanford applications, we found $LimitedSelf$, $Implementor$, $Overrider$, $Sink$, $Stateless$, $FunctionObject$ as the most frequent patterns in the neutral classes compared to the vulnerable classes as in Figure 7.4.

We performed a chi-square test to measure the statistical significance of our findings. We formulated our Hypothesis as $H0$: *Software vulnerabilities are independent of micro*

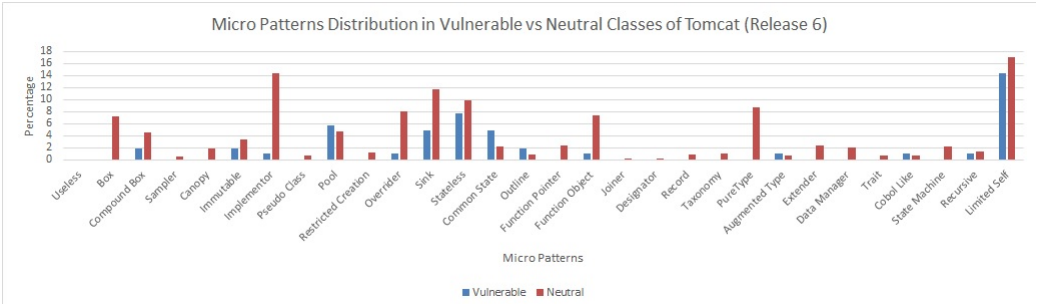


Figure 7.1

Micro Patterns Distribution in Tomcat (Release 6)

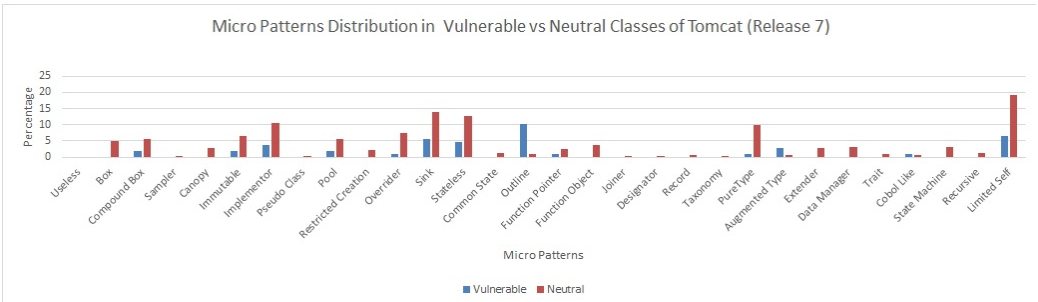


Figure 7.2

Micro Patterns Distribution in Tomcat (Release 7)

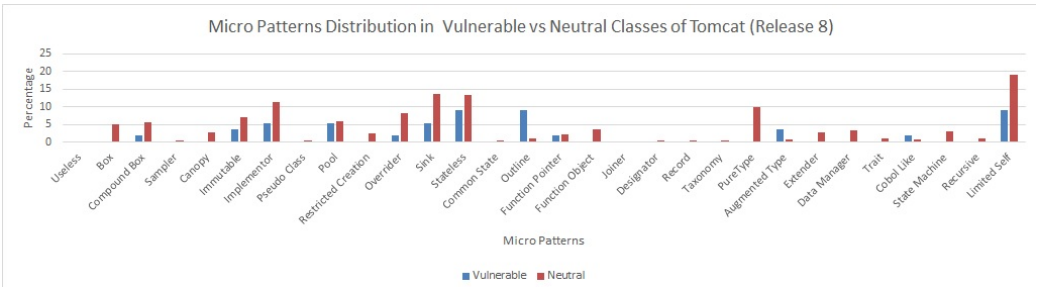


Figure 7.3

Micro Patterns Distribution in Tomcat (Release 8)

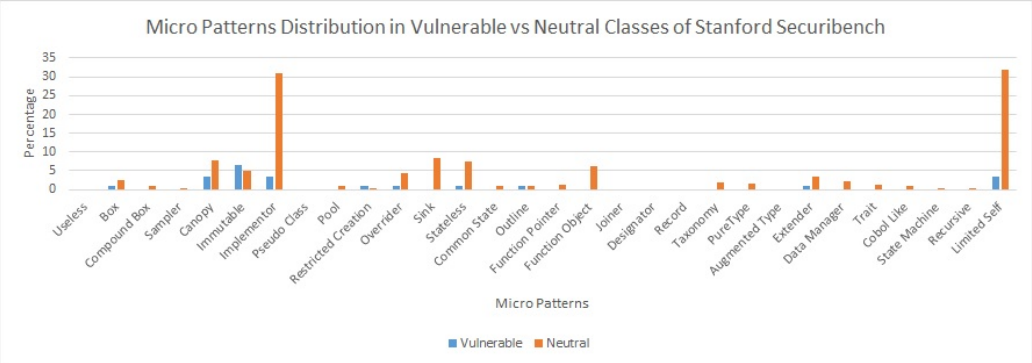


Figure 7.4

Micro Patterns Distribution in Stanford Securibench

patterns contained in their source code and assumed $\alpha = 0.05$. We have tested this hypothesis for all the affected and non-affected versions in release 6. We have reported the chi-square values for all of them that we found using the SPSS tool in Table 7.1. In our case, degrees of freedom is 1. For degrees of freedom=1 and at 5% level of significance, the appropriate critical value is 3.84, and the decision rule is: *Reject* $H0$ *if* $\chi^2 \geq 3.84$. Therefore, we reject $H0$ for the micro patterns whose chi-square values are greater than 3.84 according to Table 7.1. In other words, we find statistically significant associations between Tomcat vulnerabilities and these micro patterns at $\alpha = 0.05$ ($p \leq 0.005$). On the other hand, there are some micro patterns such as $Sampler$, $PseudoClass$, $Pool$, $CommonState$, $FunctionPointer$, $Joiner$, $Designator$, $Record$, $Taxonomy$, $Trait$, $CobolLike$, and $Recursive$ for which we cannot reject the null hypothesis (for these patterns $\chi^2 \leq 3.84$). This result shows there was not enough evidence at $\alpha = 0.05$ to claim that Tomcat vulnerabilities and these micro patterns are dependent where $p \leq 0.005$. To obtain the statistical significance of our findings in Stanford, we performed a chi-square

test and had statistically significant evidence at $\alpha = 0.05$ to show that vulnerabilities present in the web applications and these micro patterns are not independent (i.e. they are dependent or related in some way) where $p \leq 0.005$. We have reported the chi-square values that are greater than 3.84 in Table 7.2. In our case, degrees of freedom is 1. For $LimitedSelf$, $Implementor$, $Sink$, $Stateless$, $FunctionObject$ micro patterns, we can say that their presence in neutral classes is significantly different than their presence in vulnerable classes.

Table 7.1

Chi-square values for micro patterns in Tomcat (Release 6)

Micro Pattern	**Chi-Square**	**Micro Pattern**	**Chi-Square**
Outline	75.645	Compound Box	6.101
Pure Type	25.776	Immutable	5.947
Implementor	19.557	Restricted Creation	5.438
Overrider	16.378	Function Pointer	2.997
Box	15.801	Trait	2.304
Sink	14.095	Record	1.8
Limited Self	11.764	Taxonomy	1.599
Function Object	11.223	Recursive	1.557
Augmented Type	9.596	Sampler	1.331
Data Manager	7.73	Pseudo Class	1.231
State Machine	7.695	Designator	1.03
Extender	7.101	Common State	0.83
Canopy	7.032	Pool	0.816
Stateless	6.874	Cobol Like	0.783
		Joiner	0.232

The comparative study between nano-pattern distribution in the vulnerable methods and that in the neutral methods show significant differences between them. This result leads to the conclusion that some nano-patterns are more frequent in vulnerable methods than they are in neutral methods and vice versa. Our significant observations are as follows:

Table 7.2

Chi-square values for micro patterns in Stanford Securibench

Micro Pattern	**Chi-Square**
LimitedSelf	30.899
Implementor	29.213
Sink	8.053
Function Object	6.074
Stateless	5.052

- There are some nano-patterns that are more widely present in vulnerable methods of Tomcat-6, 7, and 8 compared to the neutral methods as shown in Figure 7.5, Figure 7.6, and Figure 7.7 respectively. They are $objCreator$, $thisInstanceFieldReader$, $thisInstanceFieldWriter$, $otherInstanceFieldReader$, $otherInstanceFieldWriter$, $looper$, $exceptions$, $localWriter$, $arrReader$.

- There are some nano-patterns that are more widely present in neutral methods of Tomcat-6, 7, and 8 compared to their presence in vulnerable methods as shown in Figure 7.5, Figure 7.6, and Figure 7.7 respectively. They are $void$, $samename$, $leaf$, $tailCaller$, and $straightLine$.

- According to Figure 7.8, the most prominent nano-patterns in vulnerable methods of Stanford applications are $void$, $objCreator$, $thisInstanceFieldReader$, $staticFieldReader$, $typeManipulator$, $looper$, $exceptions$, $localReader$, $localWriter$, $jdkClient$, and $tailCaller$. On the other hand, $samename$, $leaf$, and $straightLine$ were identified as significantly less-prominent in vulnerable methods compared to their existence in neutral methods.

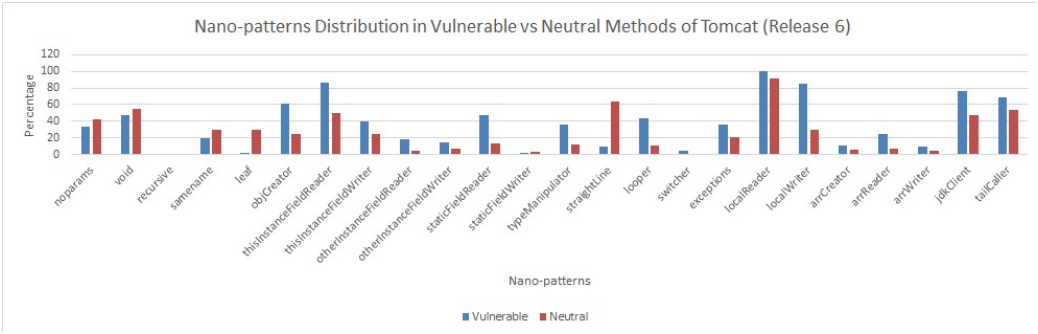


Figure 7.5

Nano-patterns Distribution in Tomcat (Release 6)

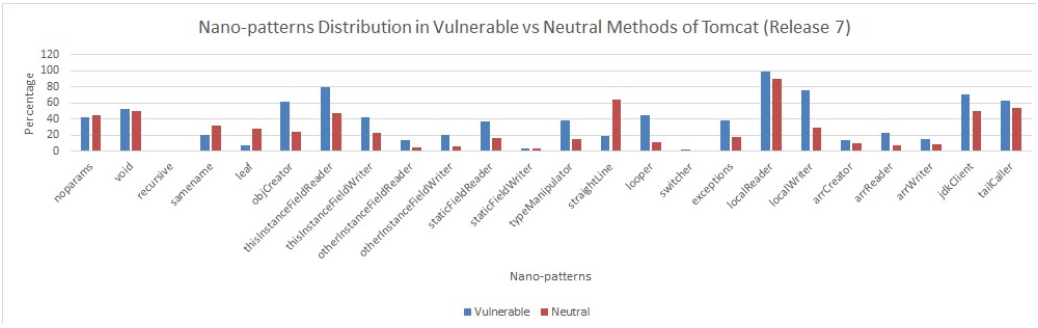


Figure 7.6

Nano-patterns Distribution in Tomcat (Release 7)

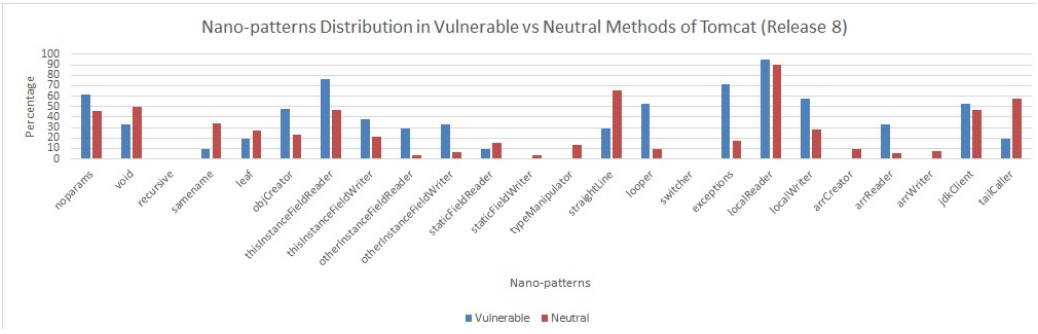


Figure 7.7

Nano-patterns Distribution in Tomcat (Release 8)

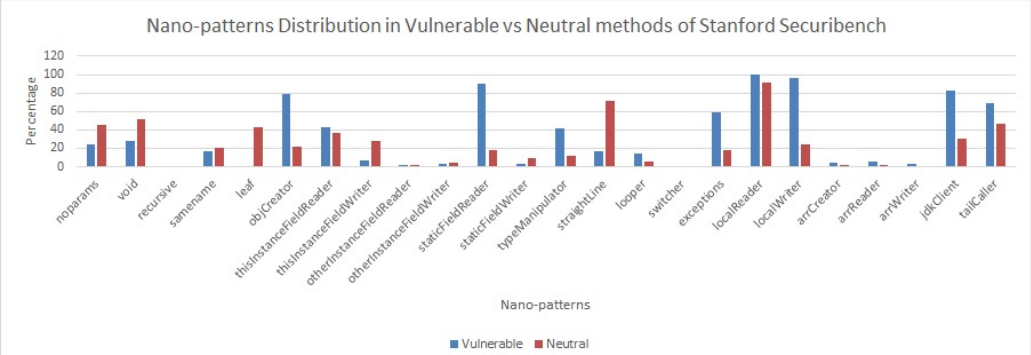


Figure 7.8

Nano-patterns Distribution in Stanford Securibench

To obtain the statistical significance of our findings, we performed a chi-square test. We formulated our Hypothesis $H0$ as *Software vulnerabilities are independent of the nano-patterns contained in their source code*, and we assumed $\alpha = 0.05$. We tested this hypothesis for all the affected and non-affected versions in release 8. We have reported the chi-square values for all of them that we found in Table 7.3. In our case, degrees of freedom is 1. For degrees of freedom=1 and at 5% significance level, the appropriate critical value is 3.84, and the decision rule is: *Reject* $H0$ *if* $\chi^2 \geq 3.84$. Therefore, we reject $H0$ for the nano-patterns whose chi-square values are greater than 3.84 according to Table 7.3. In other words, we can say that we have statistically significant evidence at $\alpha = 0.05$ to show that vulnerabilities present in Tomcat and these nano-patterns are not independent (i.e., they are dependent or related in some way) where $p \leq 0.005$. On the other hand, there are certain nano-patterns for which we cannot reject the null hypothesis where $\chi^2 \leq 3.84$ as in Table 7.3. In other words, we can say that we do not have statistically significant evidence

at $\alpha = 0.05$ to show that vulnerabilities present in Tomcat and these nano-patterns are not independent where $p \leq 0.005$.

Table 7.3

Chi-square values for nano-patterns in Tomcat (Release 8)

Nano-pattern	Chi-Square	Nano-pattern	Chi-Square
exceptions	44.122	typeManipulator	3.289
looper	43.334	void	2.350
otherInstanceFieldReader	36.981	noparams	2.269
arrReader	31.597	arrCreator	2.088
otherInstanceFieldWriter	26.879	arrWriter	1.734
straightLine	13.052	leaf	.658
tailCaller	12.896	staticFieldWriter	.655
localWriter	9.083	localReader	.566
objCreator	7.426	staticFieldReader	.489
thisInstanceFieldReader	7.066	jdkClient	.296
samename	5.549	client	.184
thisInstanceFieldWriter	3.723	switcher	.122
		recursive	.069

7.1.2 How are the traceable patterns associated with each other in vulnerable and neutral code?

We have computed the phi-coefficient for each pair of micro patterns and nano-patterns in both the vulnerable and neutral versions of Tomcat 6, 7, 8, and Stanford Securibench applications [9]. We present the connected pairs of micro patterns of vulnerable and neutral versions of Tomcat 6, 7, and 8 and Stanford Securibench in Table 7.4, Table 7.5, Table 7.6, and Table 7.7 respectively. We also present the connected pairs of nano-patterns of vulnerable and neutral versions of three releases of Apache Tomcat and Stanford Securibench in Table 7.8 and Table 7.9 respectively. Here we present the associations that are different in vulnerable versus neutral versions in respective releases.

Table 7.4

Micro patterns association types in Tomcat (Release 6)

Vulnerable		Neutral	
High Association	**Medium Association**	**High Association**	**Medium Association**
Compound Box-Immutable (1)	Common State-Function Object (0.438)	Implementor-Function Object (0.598)	Stateless-Function Pointer (0.467)
Function Object-Cobol Like (1)	Common State-Cobol Like (0.438)		Box-Implementor (0.384)
Pool-Sink (0.908)			Pure Type-State Machine (0.374)
Sink-Stateless (0.778)			Function Pointer-Cobol Like (0.364)
Compound Box-Implementor (0.704)			Box-Function Object (0.361)
Immutable-Implementor (0.704)			Sink-Stateless (0.357)
Pool-Stateless (0.702)			Pool-Stateless (0.357)
			Outline-Trait (0.349)
			Function Pointer-Limited Self (0.343)
			Immutable-Restricted Creation (0.332)

Table 7.5

Micro patterns association types in Tomcat (Release 7)

Vulnerable		Neutral	
High Association	**Medium Association**	**High Association**	**Medium Association**
Compound Box-Immutable (1)	Stateless-Cobol Like (0.439)		Implementor-Function Object (0.468)
Function Pointer-Cobol Like (1)	Function Pointer-Limited Self (0.367)		Pure Type-State Machine (0.457)
Pool-Sink (0.566)	Cobol Like - Limited Self (0.367)		Compound Box-Restricted Creation (0.418)
Pool-Limited Self (0.522)			Immutable-Restricted Creation (0.417)
			Function Pointer-Cobol Like (0.416)
			Sink-Data Manager (0.409)
			Sink-Limited Self (0.392)
			Pool-Restricted Creation (0.343)
			Implementor-Function Pointer (0.341)
			Joiner-Taxonomy (0.332)
			Compound Box-Immutable (0.331)
			Function Pointer-Limited Self (0.33)

Table 7.6

Micro patterns association types in Tomcat (Release 8)

Vulnerable		Neutral	
High Association	**Medium Association**	**High Association**	**Medium Association**
Pool-Sink (1)	Stateless-Cobol Like (0.43)	Function Object-Cobol Like (0.534)	Pure Type-State Machine (0.474)
Function Pointer-Cobol Like (1)	Function Pointer-Limited Self (0.43)		Immutable-Restricted Creation (0.472)
Pool-Stateless (0.759)			Compound Box-Restricted Creation (0.466)
Sink-Stateless (0.759)			Sink-Data Manager (0.435)
Pool-Limited Self (0.759)			Sink-Limited Self (0.393)
Sink-Limited Self (0.759)			Pool-Restricted Creation (0.392)
Compound Box-Immutable (0.7)			Compound Box-Immutable (0.361)
			Pool-Stateless (0.345)
			Sink-Stateless (0.329)
			Function Object-Limited Self (0.317)

Table 7.7

Micro patterns association types in Stanford Securibench

Vulnerable		Neutral	
High Association	**Medium Association**	**High Association**	**Medium Association**
Outline-Extender (1)	Immutable-RestrictedCreation (.397)	Canopy-FunctionObject (.772)	Sink-Taxonomy (.467)
	Canopy-Implementor (.311)	FunctionPointer-CobolLike (.667)	Stateless-FunctionPointer (.417)
		Implementor-LimitedSelf (.591)	PureType-StateMachine (.405)
		Pool-CommonState (.573)	CompoundBox-DataManager (.344)
		Sink-DataManager (.5)	Implementor-FunctionObject (.316)

Table 7.8

Nano-patterns association types in Apache Tomcat (Release 6, 7, 8)

High Association	**Medium Association**
leaf-straightLine (.606)	localWriter-jdkClient (.487)
objCreator-jdkClient (.591)	objCreator-localWriter (.468)
looper-arrReader (.514)	otherInstanceFieldReader-otherInstanceFieldWriter (.441)
	otherInstanceFieldReader-arrReader (.418)
	otherInstanceFieldWriter-exceptions (.398)
	typeManipulator-jdkClient (.375)
	looper-exceptions (.368)
	objCreator-typeManipulator (.359)
	arrReader-arrWriter (.343)
	typeManipulator-tailCaller (.333)
	looper-localWriter (.324)
	staticFieldReader-localWriter (.312)
	typeManipulator-localWriter(.303)

Table 7.9

Nano-patterns association types in Stanford Securibench

High Association	**Medium Association**
arrCreator-arrWriter (.782)	otherInstanceFieldReader-arrCreator (.49)
objCreator-jdkClient (.708)	thisInstanceFieldWriter-otherInstanceFieldWriter (.43)
otherInstanceFieldReader-arrWriter (.626)	noparams-thisInstanceFieldReader (.419)
staticFieldReader-localWriter (.557)	exceptions-jdkClient (.405)
objCreator-exceptions (.524)	void-thisInstanceFieldWriter (.402)
	samename-straightLine (.399)
	typeManipulator-exceptions (.389)
	objCreator-localWriter (.357)
	typeManipulator-jdkClient (.341)
	objCreator-typeManipulator (.33)
	typeManipulator-looper (.322)
	looper-arrCreator (.312)
	localWriter-jdkClient (0.308)

7.1.3 Are the association rules useful in identifying vulnerabilities?

As we see from Table 7.10 we can conclude the following statements:

- $objCreator$ and $typeManipulator$ either exist together in vulnerable methods or both of them are absent (i.e., if both patterns exist, the method contains a vulnerability).

- In vulnerable methods, both $jdkClient$ and $tailCaller$ exist together, but in neutral methods, there is no specific pattern between them.

The correlation measure among the two patterns $jdkClient$ and $tailCaller$ in neutral methods is 0.21716, which is not strong. The correlation among these two patterns is 100 percent within vulnerable methods. Hypothesis $H0$ states that *Vulnerability generation is independent of dependencies among the nano-patterns*. We set $\alpha = 0.05$. The value of χ^2 was 2.74 for Tomcat-6 and 0.61 for Tomcat-7. For degrees of freedom $= 1$ and a 5%

Table 7.10

Best Association Rules (in Tomcat-6 and Tomcat-7)

Vulnerable Methods	Neutral Methods
$typeManipulator = 0 \rightarrow objCreator = 0$ conf:(1)	$objCreator = 0 \rightarrow typeManipulator = 0$ conf:(0.91)
$objCreator = 0 \rightarrow typeManipulator = 0$ conf:(1)	
$typeManipulator = 1 \rightarrow objCreator = 1$ conf:(1)	
$objCreator = 1 \rightarrow typeManipulator = 1$ conf:(1)	
$looper = 0 \rightarrow straightLine = 0$ conf:(1)	$straightLine = 1 \rightarrow looper = 0$ conf:(1)
$straightLine = 0 \rightarrow looper = 0$ conf:(1)	
$tailCaller = 1 \rightarrow jdkClient = 1$ conf:(1)	
$jdkClient = 1 \rightarrow tailCaller = 1$ conf:(1)	

level of significance, the appropriate critical value is 3.84 and the decision rule is: *Reject* $H0$ *if* $\chi^2 \geq 3.84$. Therefore, we could not reject $H0$ as chi-square value was less than critical value for both versions. In other words, we can say that we do not have statistically significant evidence at $\alpha = 0.05$ to show that vulnerability generation and a dependency among the patterns are not independent. We found associations among a small number of nano-patterns in vulnerable methods. We assume that the association among the patterns may or may not have significant impact on the vulnerabilities generated.

7.1.4 How do traceable patterns evolve from vulnerable code to neutral code?

To answer this question, we analyzed all the vulnerable classes of a particular release and then analyzed those classes in the neutral version of the same release where the vulnerability had been fixed. We have also determined how the micro patterns changed across the releases and identified several evolution types. For example, in Figure 7.9, we see that Test.java has been evolving across versions 6.0.1 through 6.0.5. The vulnerability was active in versions 6.0.2 and 6.0.3. So we consider 6.0.3 as the last-affected version. The vulnerability has been fixed in version 6.0.4, and after that version, all the released versions

have been considered as neutral versions for that vulnerability. So, in this example, version 6.0.3 is the last-affected version and any version after 6.0.4 can be considered neutral for that vulnerability.

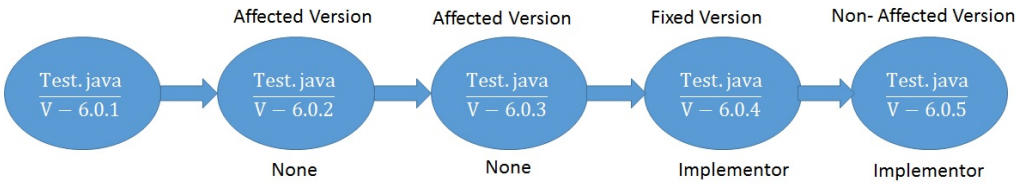


Figure 7.9

Example of micro pattern evolution across vulnerable to neutral class

If we focus on the evolution of the micro pattern in this example, we see that micro pattern $None$ has been changed to $Implementor$ as soon as the vulnerability has been fixed. So the micro pattern type of Test.java has been evolved from $None$ to $Implementor$ ($None \rightarrow Implementor$) once its vulnerability has been fixed and it has transitioned from vulnerable to a neutral class. In other words, we can say that micro pattern $Implementor$ has been introduced once the vulnerability is removed. The micro patterns evolution types found in releases 6, 7, and 8 are listed in Table 7.11. We also notice that in many cases, the micro patterns do not change across the vulnerable to the neutral class. When a class is generally modified to fix a vulnerability, the pattern type remains same. In those cases, the vulnerability is fixed without changing the pattern. Therefore, the findings of our study do not encourage avoiding using some micro patterns in code, but it helps the developers to use them carefully so that they do not make the program vulnerable. We have also detected

how the micro patterns evolve across the releases with time. The types of micro patterns evolution across the releases is listed in Table 7.12. The frequency is computed with respect to the total number of evolution types found in Apache Tomcat. From this table, we see that some evolution types such as $None \rightarrow Implementor$, $Implementor \rightarrow None$, $CommonState \rightarrow Stateless$ are more frequent compared to other types of evolution.

Table 7.11

Micro pattern evolution types from vulnerable to neutral classes in Tomcat

Micro Pattern Evolution
$Recursive \rightarrow None$
$AugmentedType \rightarrow None$
$None \rightarrow Sink$

7.2 RQ2: Can traceable patterns better predict vulnerable code than software metrics?

7.2.1 What are the performance measures of traceable patterns in vulnerability prediction?

The False Negative (FN) rate, Precision, Recall, and F-Measure of the micro and nano-patterns based prediction model for Tomcat-6, Tomcat-7, Stanford Securibench, and Apache CXF are presented in Table 7.14, Table 7.15, Table 7.16, and Table 7.17 respectively. The list of nano-patterns that we found as significant for the systems is presented in Table 7.13.

Table 7.12

Micro pattern evolution types across the releases of Tomcat

Micro Pattern Evolution	**Frequency**
$None \rightarrow None$	66%
$None \rightarrow Implementor$	5.3%
$CommonState \rightarrow Stateless$	5.3%
$Implementor \rightarrow None$	2.67%
$None \rightarrow Sink$	1.3%
$None \rightarrow LimitedSelf$	1.3%
$None \rightarrow FunctionPointer$	1.3%
$AugmentedType \rightarrow None$	1.3%
$Recursive \rightarrow None$	1.3%

Table 7.13

Results of Welch's Test for nano-patterns (*Effect sizes are mentioned within brackets and for all correlations $p < .05$.)

Tomcat-6	**Tomcat-7**	**Apache CXF**	**Stanford Securibench**
noparams (-.183)	samename (.291)	noparams (-.416)	noparams (-.504)
samename (.333)	leaf (-.483)	objCreator (.613)	void (-.842)
leaf (-.633)	objCreator (.826)	staticFieldReader (1.016)	objCreator (1.527)
objCreator (.801)	thisInstanceFieldReader (.447)	typeManipulator (1.631)	staticFieldReader (1.745)
thisInstanceFieldReader (.57)	thisInstanceFieldWriter (.515)	straightLine (-.968)	typeManipulator (1.181)
thisInstanceFieldWriter (.43)	otherInstanceFieldReader (-.175)	looper (1.11)	straightLine (-1.042)
staticFieldReader (.846)	otherInstanceFieldWriter (.274)	exceptions (.430)	exceptions (1.183)
typeManipulator (.539)	staticFieldReader (.530)	localWriter (1.238)	jdkClient (1.092)
straightLine (-.824)	typeManipulator (.523)	jdkClient (.994)	tailCaller (.461)
looper (.711)	straightLine (-.601)		
exceptions (.633)	looper (.736)		
localWriter (.829)	exceptions (.761)		
arrReader (.218)	localReader (.156)		
arrWriter (-.17)	localWriter (.645)		
jdkClient (.736)	jdkClient (.109)		
tailCaller (.420)	tailCaller (.260)		

Table 7.14

Machine Learning results for traceable patterns in Tomcat (Release 6)

	Method	FN Rate	Precision	Recall	F-Measure
Micro Patterns	Naive Bayes	0.144	0.613	0.856	0.793
	Logistic Regression	0.183	0.604	0.817	0.763
	SVM	0.135	0.603	0.865	0.796

	Method	FN Rate	Precision	Recall	F-Measure
Nano-patterns	Naive Bayes	0.177	0.720	0.823	0.800
	Logistic Regression	0.210	0.766	0.790	0.785
	SVM	0.258	0.763	0.742	0.746

Table 7.15

Machine Learning results for traceable patterns in Tomcat (Release 7)

	Method	FN Rate	Precision	Recall	F-Measure
Micro Patterns	Naive Bayes	0.160	0.631	0.840	0.788
	Logistic Regression	0.113	0.640	0.887	0.823
	SVM	0.104	0.644	0.896	0.831
Nano-patterns	Naive Bayes	0.311	0.676	0.689	0.686
	Logistic Regression	0.292	0.744	0.708	0.715
	SVM	0.302	0.730	0.698	0.704

Table 7.16

Machine Learning results for traceable patterns in Stanford Securibench

	Method	FN Rate	Precision	Recall	F-Measure
Micro Patterns	Naive Bayes	0.088	0.604	0.912	0.828
	Logistic Regression	0.099	0.618	0.901	0.829
	SVM	0.077	0.610	0.923	0.837
Nano-patterns	Naive Bayes	0.173	0.812	0.827	0.824
	Logistic Regression	0.135	0.837	0.865	0.824
	SVM	0.103	0.833	0.897	0.883

Table 7.17

Machine Learning results for traceable patterns in Apache CXF

	Method	FN Rate	Precision	Recall	F-Measure
Micro Patterns	Naive Bayes	0.394	0.630	0.606	0.612
	Logistic Regression	0.364	0.765	0.636	0.658
	SVM	0.424	0.716	0.576	0.599
Nano-patterns	Naive Bayes	0.178	0.771	0.822	0.771
	Logistic Regression	0.156	0.796	0.844	0.859
	SVM	0.200	0.788	0.800	0.798

7.2.2 What are the performance measures of software metrics in vulnerability prediction?

The False Negative (FN) rate, Precision, Recall, and F-Measure of class-level and method-level metrics based prediction model for Tomcat-6, Tomcat-7, Stanford Securibench, and Apache CXF have been presented in Table 7.18, Table 7.19, Table 7.20, and Table 7.21 respectively.

Table 7.18

Machine Learning results for software metrics in Tomcat (Release 6)

	Method	FN Rate	Precision	Recall	F-Measure
Class metrics	Naive Bayes	0.204	0.865	0.796	0.809
	Logistic Regression	0.185	0.859	0.815	0.823
	SVM	0.185	0.864	0.815	0.824

	Method	FN Rate	Precision	Recall	F-Measure
Method metrics	Naive Bayes	0.500	0.847	0.500	0.545
	Logistic Regression	0.347	0.794	0.653	0.677
	SVM	0.331	0.775	0.669	0.688

Table 7.19

Machine Learning results for software metrics in Tomcat (Release 7)

	Method	FN Rate	Precision	Recall	F-Measure
Class metrics	Naive Bayes	0.268	0.890	0.732	0.759
	Logistic Regression	0.125	0.888	0.875	0.878
	SVM	0.143	0.884	0.857	0.862

	Method	FN Rate	Precision	Recall	F-Measure
Method metrics	Naive Bayes	0.475	0.856	0.525	0.569
	Logistic Regression	0.366	0.800	0.634	0.661
	SVM	0.356	0.794	0.644	0.669

Table 7.20

Machine Learning results for software metrics in Stanford Securibench

	Method	FN Rate	Precision	Recall	F-Measure
Class metrics	Naive Bayes	0.506	0.729	0.494	0.528
	Logistic Regression	0.329	0.759	0.671	0.687
	SVM	0.291	0.764	0.709	0.719
Method metrics	Naive Bayes	0.408	0.832	0.592	0.628
	Logistic Regression	0.224	0.830	0.776	0.628
	SVM	0.099	0.809	0.901	0.881

Table 7.21

Machine Learning results for software metrics in Apache CXF

	Method	FN Rate	Precision	Recall	F-Measure
Class metrics	Naive Bayes	0.194	0.907	0.806	0.854
	Logistic Regression	0.194	0.895	0.806	0.822
	SVM	0.194	0.885	0.806	0.824
Method metrics	Naive Bayes	0.311	0.891	0.689	0.722
	Logistic Regression	0.289	0.866	0.711	0.786
	SVM	0.356	0.871	0.644	0.679

7.3 RQ3: How do we determine the most significant set of patterns and metrics to build a framework for vulnerability prediction?

7.3.1 What are the significant nano-patterns that can predict vulnerable and neutral methods?

Table 7.22 presents the chi-square values of the nano-patterns for all the systems under study. Here we present only the nano-patterns for which we got chi value greater than 3.84 and we can reject the null hypothesis *H0:There is no association between nano-patterns and vulnerabilities.* As we know that, for degrees of freedom=1 and at 5% level of significance, the appropriate critical value is 3.84, and the decision rule is: *Reject* $H0$ *if* $\chi^2 \geq 3.84$. Table 7.23 lists all the nano-patterns chosen as our final list of features. For this set of nano-patterns, we can reject the null hypothesis and they are common in all the systems under study.

Table 7.22

Chi-square values of nano-patterns

Tomcat-6		Tomcat-7		CXF		Stanford	
Nano-patterns	**Chi-sq values**	**Nano-patterns**	**Chi-sq values**	**Nano-patterns**	**Chi-sq values**	**Nano-patterns**	**Chi-sq values**
noparams	6.88	void	3.91	straightLine	30.22	void	8.75
straightLine	105.35	samename	16.56	leaf	16.92	noparams	15.93
samename	30.12	leaf	31.54	objCreator	8.12	samename	4.83
leaf	60.06	objCreator	57.1	thisInstanceFieldWriter	9.65	leaf	116.55
objCreator	73.48	thisInstanceFieldReader	13.12	staticFieldReader	28.41	objCreator	164.72
thisInstanceFieldReader	32.26	thisInstanceFieldWriter	9.1	typeManipulator	59.93	thisInstanceFieldWriter	22.02
thisInstanceFieldWriter	9.11	otherInstanceFieldReader	6.38	looper	38.75	staticFieldReader	422.12
otherInstanceFieldReader	29.88	otherInstanceFieldWriter	24.58	exceptions	11.46	typeManipulator	59.73
otherInstanceFieldWriter	11.16	staticFieldReader	30.09	localWriter	41.57	straightLine	147.69
staticFieldReader	71.12	staticFieldWriter	13.2	jdkClient	28.39	looper	11.62
staticFieldWriter	6.11	typeManipulator	42.81			exceptions	90.89
typeManipulator	54.04	straightLine	57.17			localReader	7.28
looper	56.93	looper	48.37			localWriter	225.67
exceptions	38.04	exceptions	24.20			arrCreator	4.19
localWriter	109.19	localWriter	61.48			jdkClient	136.15
arrCreator	7.04	arrCreator	6.48			tailCaller	38.33
arrReader	16.52	arrReader	14.54				
arrWriter	4.6	arrWriter	11.20				
jdkClient	50.42	jdkClient	29.38				
tailCaller	29.20	tailCaller	16.25				

** For all values, $p < .05$.

Table 7.23

Selected Set of nano-patterns

straightLine
leaf
objCreator
thisInstanceFieldWriter
staticFieldReader
typeManipulator
looper
exceptions
localWriter
jdkClient

7.3.2 What are the significant software metrics that can predict vulnerable and neutral methods?

We performed the Welch's t-test to select a significant set of method-level software metrics in the systems under study. Table 7.24 shows that all the significant metrics are common for each system. Therefore, we selected these metrics to be combined with selected nano-patterns to develop the nano-metrics.

Table 7.24

Results of Welch's Test

Tomcat-6	**Tomcat-7**	**Apache CXF**	**Stanford Securibench**
CountInput (1.156)	CountInput (.688)	CountInput (.356)	CountInput (.502)
CountLineCode (1.658)	CountLineCode (1.162)	CountLineCode (1.888)	CountLineCode (1.646)
CountOutput (1.752)	CountOutput (1.453)	CountOutput (1.674)	CountOutput (1.567)
Cyclomatic (1.468)	Cyclomatic (1.039)	Cyclomatic (1.801)	Cyclomatic (1.184)
CyclomaticModified (1.579)	CyclomaticModified (1.101)	CyclomaticModified (2.198)	CyclomaticModified (1.185)
CyclomaticStrict (1.503)	CyclomaticStrict (1.007)	CyclomaticStrict (1.918)	CyclomaticStrict (.974)
Essential (1.086)	Essential (.692)	Essential (1.783)	Essential (.260)
MaxNesting (1.118)	MaxNesting (.943)	MaxNesting (1.499)	MaxNesting (1.659)

*Effect sizes are mentioned within brackets and for all correlations $p < .05$.

7.4 RQ4: Is the framework effective at predicting vulnerabilities?

In this section, we present the performance measures including FN Rate, FP Rate, Precision, Recall, F_2-measure, and ROC while using nano-metrics as features for the vulnerability prediction model. The nano-metrics that were developed combining the selected nano-patterns and method-level metrics are listed in Table 7.25. Table 7.26 and Table 7.27 present the measures in logistic regression and support vector machine techniques respectively. Table 7.28 and Table 7.29 show the results while using Tomcat-6 and Tomcat-7 as

training datasets in the logistic regression model. Table 7.30 and Table 7.31 show the results while using Tomcat-6 and Tomcat-7 as training datasets in the support vector machine model.

Table 7.25

Nano-metrics

straightLine
leaf
objCreator
thisInstanceFieldWriter
staticFieldReader
typeManipulator
looper
exceptions
localWriter
jdkClient
CountInput
CountLineCode
CountOutput
Cyclomatic
CyclomaticModified
CyclomaticStrict
Essential
MaxNesting

Table 7.26

Performance Measures in Logistic Regression

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-6	0.202	0.255	0.758	0.798	0.777	0.843
Tomcat-7	0.314	0.237	0.743	0.686	0.714	0.761
Apache CXF	0.275	0.170	0.810	0.725	0.765	0.868
Stanford	0.095	0.145	0.862	0.905	0.883	0.938

Table 7.27

Performance Measures in Support Vector Machine

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-6	0.211	0.320	0.711	0.789	0.748	0.734
Tomcat-7	0.279	0.344	0.677	0.721	0.698	0.688
Apache CXF	0.250	0.202	0.788	0.750	0.768	0.774
Stanford	0.095	0.157	0.852	0.905	0.878	0.874

Table 7.28

Performance Measures in Logistic Regression (using Tomcat-6 as training data)

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-7	0.279	0.266	0.037	0.721	0.070	0.805
Apache CXF	0.175	0.337	0.010	0.825	0.020	0.827
Stanford	0.197	0.271	0.197	0.803	0.316	0.844
Evaluation dataset	0.269	0.249	0.076	0.731	0.137	0.827

Table 7.29

Performance Measures in Logistic Regression (using Tomcat-7 as training data)

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-6	0.229	0.242	0.065	0.771	0.119	0.853
Apache CXF	0.275	0.303	0.010	0.725	0.020	0.792
Stanford	0.252	0.258	0.194	0.748	0.308	0.831
Evaluation dataset	0.299	0.225	0.080	0.701	0.144	0.809

Table 7.30

Performance Measures in Support Vector Machine (using Tomcat-6 as training data)

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-7	0.233	0.291	0.036	0.767	0.069	0.738
Apache CXF	0.100	0.379	0.010	0.900	0.019	0.761
Stanford	0.136	0.281	0.203	0.864	0.329	0.792
Evaluation dataset	0.194	0.268	0.077	0.806	0.141	0.769

Table 7.31

Performance Measures in Support Vector Machine (using Tomcat-7 as training data)

System	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC
Tomcat-6	0.147	0.363	0.049	0.853	0.092	0.745
Apache CXF	0.025	0.464	0.009	0.975	0.017	0.755
Stanford	0.034	0.339	0.191	0.966	0.319	0.813
Evaluation dataset	0.209	0.324	0.064	0.791	0.118	0.733

CHAPTER 8

DISCUSSION

This chapter discusses the experimental results.

8.1 RQ1: What is the relation between traceable patterns and software vulnerabilities?

Is there any significant difference between traceable pattern distribution in vulnerable and neutral code?

We found $Outline$ and $AugmentedType$ micro patterns to be more frequent in vulnerable classes than they are in neutral classes of Apache Tomcat. Gil et al. [18] defined an $Outline$ pattern as an abstract class where two or more declared methods invoke at least one abstract method of the current ("this") object. On the other hand, a class having only abstract methods and three or more static final fields of the same type is known as $AugmentedType$ [18]. As an abstract class can not be instantiated, and it is only for other classes to extend, the abstract classes with abstract methods need to be used carefully such that other classes can ensure their secured use. Kim et al. in [24] also observed the $Outline$ micro pattern as a defective pattern as they found high bug rates in classes having this pattern. According to Destefanis [12], "The Cobol Like anti pattern is a class having a single static method, one or more static variables, and no instance methods or fields. Cobol Like

classes do not declare any method or instance field. Classes of this kind are very far from the object-oriented programming paradigm and should be very rare." The statement supports our result as $CobolLike$ pattern is also not declared as a safe pattern in our analysis. Destefanis in [12] declared five micro patterns $Pool$, $CobolLike$, $Record$, $PseudoClass$, and $FunctionPointer$ as anti-patterns as they are associated with bad programming practices. In our study, they are also not present in the list of safer micro patterns that have been shown to be significantly frequent in neutral classes compared to vulnerable classes.

Another observation is the $Containment$ category patterns such as Box, $CompundBox$, $Canopy$, $DataManager$ are almost absent in vulnerable classes of Tomcat. Box, $CompundBox$, $Canopy$ patterns are classes that wrap a central instance field with their methods. The main purpose of $DataManager$ and $Sink$ patterns is related to the management of data stored in a set of instance variables. The $DataManager$ pattern is a set of setter and getter methods which encapsulates all its fields and controls the access to these fields [18]. This is an example of a good object-oriented programming practice that ensures encapsulation. The presence of the patterns in the $inheritors$ category such as $Implementor$, $Overrider$, and $Extender$ in neutral classes is more significant than their presence in the vulnerable classes of Tomcat. Kim et al. in [24] found the evolution type $Implementor \rightarrow None$ as bug-prone. It indicates that the removal of $Implementor$ pattern makes a class defective, which supports our claim that the $Implementor$ micro pattern is more frequent in neutral classes.

On the other hand, the results of analyzing the three web applications in Stanford Securibench indicate that several micro patterns such as $LimitedSelf$, $Implementor$,

$Overrider$, $Sink$, $Stateless$, $FunctionObject$ frequently exist in neutral classes. This finding is similar to the findings from Apache Tomcat as they are also significantly frequent in Tomcat's neutral classes. The hypothesis testing validated our findings that several micro patterns are significantly associated with vulnerable code. Some patterns make the code more reliable whereas some of them reduce code robustness resulting in less reliability.

Deo et al. in [10] found that some nano-patterns such as $localReader$, $localWriter$, $fieldReader$, and $objCreator$ have a high presence in defective methods (i.e., they are more error-prone than other patterns). In our experiment, we have also discovered that a set of patterns including these patterns have a high correlation with our examined vulnerabilities. Deo et al. [10] focused on defect-prone nano-patterns while we have analyzed patterns of their potential involvement and their pair-wise associations in security vulnerabilities. We found that some patterns are more frequent in vulnerable methods compared to their presence in neutral methods. If we see the following code of the $importOldData$ method in $PersonalBlogService.java$ file of PersonalBlog as in Figure 8.1, it contains the $FieldReader$ pattern as it reads field values from an object. The line $xyz.setTitle(rs.getString("title"))$ of the method that reads field values of the ResultSet object has been identified as having a vulnerability because $rs.getString("title")$ was not sanitized before its use. Therefore, we can generalize that the methods that read values from objects must be checked for proper sanitization in order to keep the code secure. In this case, our target is not to detect the cause of any vulnerability, rather it is to reduce the risk of vulnerability by making developers aware of code weaknesses that can easily be exploited by an attacker. For this, we have taken advantage of the nano-patterns

that represent the properties of a Java method. On the other hand, $samename$, $leaf$, and $straightLine$ are comparatively more frequent in neutral methods. Their relationship with vulnerabilities have also been statistically verified using chi-square tests. Table 7.3 presents the chi-square values in decreasing order. Moreover, the distributions of these nano-patterns in three vulnerable web applications and in Apache Tomcat are identical which strengthen the claim relating to the proper use of nano-patterns.

```
Post xyz = new Post();
Timestamp created = rs.getTimestamp("created");
xyz.setContent(rs.getString("content"));
xyz.setCategory(rs.getString("category"));
xyz.setCreated(rs.getDate("created"));
xyz.setModified(rs.getDate("modified"));
xyz.setTitle(rs.getString("title"));
```

Figure 8.1

A code snippet from a method in $PersonalBlogService.java$ of PersonalBlog

How are the traceable patterns associated with each other in vulnerable and neutral code?

If we analyze the associated pairs in affected versions of Tomcat release 6, 7, and 8, we see that $CompundBox - Immutable - Implementor$, $Pool - Sink - Stateless$, and $Pool - Sink - LimitedSelf$ have a strong triangular relationship. Developers can be more careful about the existence of these micro patterns' triangles in code. If any two of the three micro patterns in each group are present, the class needs to be tested rigor-

ously. On the other hand, some associations such as $Sink - DataManager$, $Stateless - FunctionPointer$, and $Implementor - FunctionObject$ are found in the neutral classes of both the Apache Tomcat and Stanford Securibench web applications. This result indicates that these associations among the micro patterns are safe and testers can run the standard tests if they see them together in code.

In our study, the results indicate a phi-coefficient of 0.514 for the pair $looper - arrReader$ in Apache Tomcat which means $looper$ and $arrReader$ are highly associated with each other in vulnerable methods. According to the study in [47], $arrReader \rightarrow looper$ is an interesting rule that is more frequent in Java methods due to them iterating over an entire array, reading each element per iteration. We see that our study also supports their finding by adding new knowledge that this association may make a Java method insecure and need attention in vulnerability testing.

Some of the associations such as $objCreator - jdkClient$, $localWriter - jdkClient$, $objCreator - localWriter$, $typeManipulator - jdkClient$, $objCreator - typeManipulator$, and $objCreator - straightLine$ are common in all types of applications studied here. If we see the following code of the $getUserInfo$ method in $BloggerAPIHandler.java$ file of Roller as in Figure 8.2, it contains an $objCreator$ pattern as it creates a new $XmlRpcException$ object here. It also contains $localWriter$ pattern because it assigns local variable msg a value. This line of the method has been identified as having a vulnerability because e may contain sensitive information and can be exploited by an attacker resulting in information leakage. Figure 8.3 shows another code snippet from $ConsistencyCheck.java$ that contains $localWriter$, $objCreator$ and $jdkClient$ together.

This code was marked as having the SQL Injection vulnerability as string concatenation is not allowed in queries. Therefore, methods where a local variable is assigned a string value or a newly-created object should be rigorously tested.

```
String msg = "ERROR in BlooggerAPIHander.getInfo";
        mLogger.error(msg,e);
        throw new XmlRpcException(UNKNOWN_EXCEPTION,msg);
```

Figure 8.2

A code snippet from a method in $BakeWeblogAction.java$ of Roller

```
Statement st = con.createStatement();
    String query =
       "select c.id from weblogcategory as c, weblogcategoryassoc as a "
       +"where a.categoryid=c.id and a.ancestorid is null "
       +"and c.websiteid ='"+websiteid+"'";
    //System.out.println(query);
    ResultSet rs = st.executeQuery(query);
```

Figure 8.3

A code snippet from a method in $ConsistencyCheck.java$ of Roller

Are the association rules useful in identifying vulnerabilities?

We discovered that nano-patterns such as $localWriter$, $tailCaller$ and $jdkClient$ co-occur with the $localReader$ nano-pattern at 100 percent confidence. One goal for this question was to determine the co-occurrences of different patterns in a vulnerable code snippet. The rules that reveal co-existence of some nano-patterns in vulnerable code can

lead developers and testers to predict vulnerabilities due to their existence. The existence of the rule will give the developer an indication that the code is vulnerable. A tester will use the result to apply additional, targeted test to potentially vulnerable areas in the code. As some association rules among the nano-patterns have been explored to be frequent and distinct in vulnerable methods, they can help the developers to identify potentially vulnerable methods. Although we could not show that there is a correlation between vulnerability and dependencies among the patterns, we could at least indicate that some dependencies are frequent in vulnerable methods compared to others.

How do traceable patterns evolve from vulnerable code to neutral code?

Two evolution types $None \rightarrow Implementor$ and $CommonState \rightarrow Stateless$ show that $Implementor$ and $Stateless$ micro patterns are more frequent in later releases. So using these micro patterns in classes may improve the code quality by as isssues are fixed. We observe that the $CommonState$ pattern is frequent in Apache Tomcat release 6 and its uses have been abruptly reduced in later versions. We can describe this scenario in a way that many $CommonState$ patterns have been converted to $Stateless$ patterns in later releases resulting in their frequent presence in later releases. $None \rightarrow Sink$ type of evolution also supports that the pattern $Sink$ is more prevalent in later releases.

We have found that $AugmentedType$ patterns are more frequent in vulnerable classes than they are in neutral classes. According to Table 7.11, the evolution type $AugmentedType \rightarrow None$ from vulnerable to neutral version declares that $AugmentedType$ patterns have been removed from classes to make them neutral. Therefore, the vulnerable classes having $AugmentedType$ pattern in releases 6, 7 and 8 do not contain this pattern in their neutral

versions. Similarly, $None \rightarrow Sink$ type of evolution also supports that the pattern $Sink$ is more prevalent in neutral classes than it is in vulnerable classes. The evolution of the micro patterns across different releases from vulnerable to neutral classes strengthens our claim about micro pattern involvement in making a class more vulnerable.

8.2 RQ2: Can traceable patterns better predict vulnerable code than software metrics?

False Negative rate and Recall: The False Negative rates for vulnerable methods using nano-patterns are lower than method-level metrics for all the systems under study. This result shows that nano-patterns based models can retrieve more vulnerable methods than method-level metrics. The lower the False Negative rate, the higher the recall will be, resulting in a greater number of vulnerable methods with correct predictions. The comparative study of False Negative rate and Recall between nano-patterns and method-level metrics in SVM has been presented in Figure 8.5 and Figure 8.9 respectively.

Similarly, the False Negative rates for vulnerable classes using micro patterns are lower than class-level metrics for all the systems (except Apache CXF) under study. The comparative study of False Negative rate and Recall between micro patterns and class-level metrics in SVM has been presented in Figure 8.4 and Figure 8.8 respectively. The maximum False Negative rate is 31 percent in nano-patterns, which is 50 percent in method-level metrics based model. The lowest Recall rate is 69 percent in nano-patterns, which is 50 percent in the method-level metrics-based model. In our research, false negative rate and recall are the most significant measures because they determine the percentage of vulnerable code that has remained undetected. If neutral code is wrongly determined as vulnerable, it can

increase the workload of testers, but if a vulnerable code is wrongly determined as neutral, it will make the model useless for the software team. Traditional metrics are not directly related to the code constructs. Shin et al. in [45] empirically showed that complexity metrics can predict vulnerabilities at a low false positive rate but at a high false negative rate. It means the vulnerable files as detected by metrics were actual vulnerable files, but many other vulnerable files remained as undetected. Our results also support the findings of Shin et al. in [45].

Precision: Precision indicates the percentage of actual vulnerable code (method or class) in the total vulnerable codebase predicted as vulnerable by the model. It measures the correctness or the efficiency of the model. The comparative study of Precision between nano-patterns and method-level metrics using SVMs has been presented in Figure 8.7. The comparative study of Precision between micro patterns and class-level metrics has been presented in Figure 8.6. According to Figure 8.6 and Figure 8.7, class-level metrics and method-level metrics outperform micro patterns and nano-patterns respectively in terms of precision. The maximum precision is 84 percent in nano-patterns, which is 89 percent in the method-level metrics based model. The maximum precision is 77 percent in micro patterns, which is 91 percent in class-level metrics-based model. This result indicates that software metrics show more correct results than traceable patterns although they may not capture all the vulnerable code of the system.

The metrics relate to the complexity, cohesiveness or coupling among the functions. Other factors may be responsible for the likelihood that a method is vulnerable, and these factors may not be captured by the metrics. On the other hand, a neutral method is usually

less complex, highly cohesive and less coupled and can be better detected by the metrics. Therefore, metrics do not wrongly predict neutral methods as vulnerable resulting in high precision.

F-measure: F-measure is a weighted average of precision and recall. We can assign more weight to precision or recall based on our expected result. In our research, we consider F_2-measure, because it weights recall twice as much as precision. According to the Figure 8.11, F_2-measure in SVM is higher in the nano-patterns based model compared to the method-level metrics based model. On the other hand, micro patterns show lower performance compared to the class-level metrics as shown in Figure 8.10.

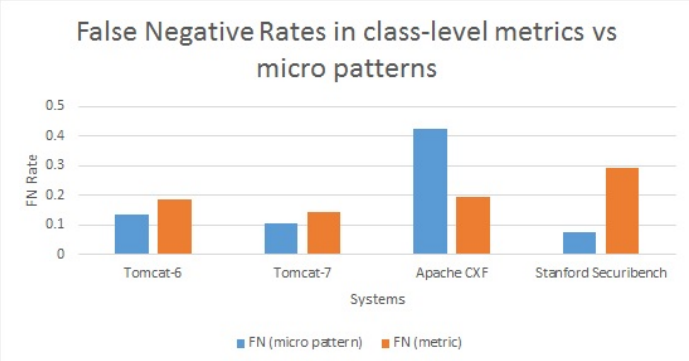


Figure 8.4

False Negative rates in class-level metrics vs micro patterns (SVM)

8.3 RQ4: Is the framework effective at predicting vulnerabilities?

8.3.1 Comparative Study among Nano-patterns, Metrics and Nano-metrics

False Negative Rate and Recall: The False Negative rate for vulnerable methods using nano-metrics is lower than method-level metrics and in some cases, lower than nano-

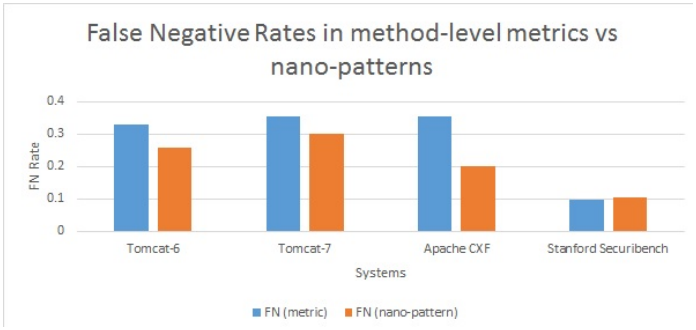


Figure 8.5

False Negative rates in method-level metrics vs nano-patterns (SVM)

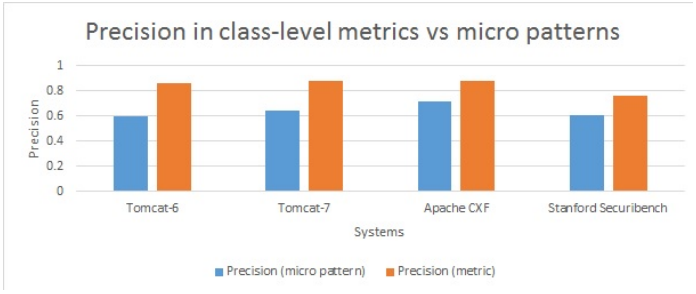


Figure 8.6

Precision in class-level metrics vs micro patterns (SVM)

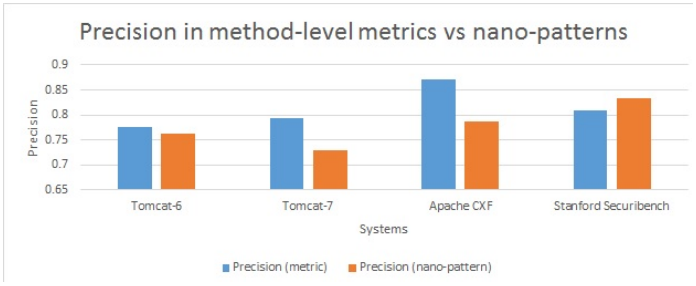


Figure 8.7

Precision in method-level metrics vs nano-patterns (SVM)

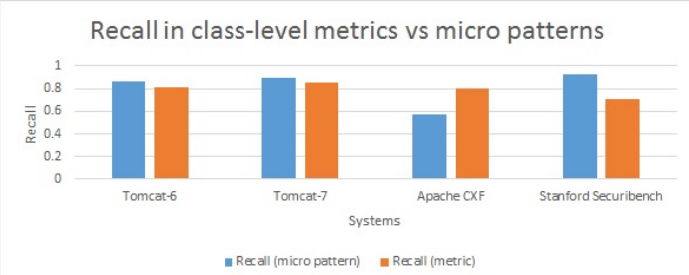


Figure 8.8

Recall in class-level metrics vs micro patterns (SVM)

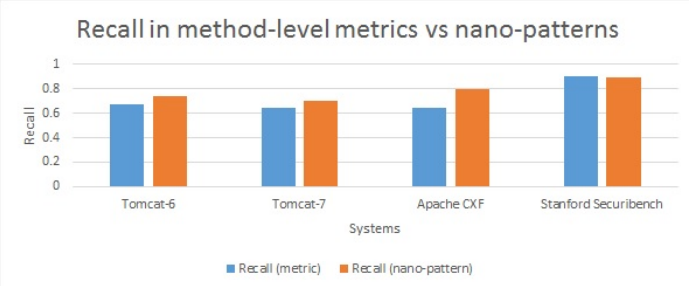


Figure 8.9

Recall in method-level metrics vs nano-patterns (SVM)

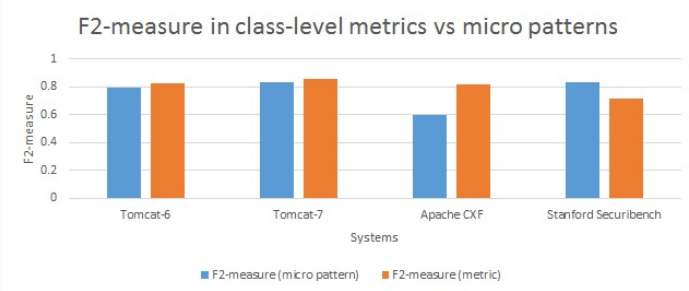


Figure 8.10

F2-measure in class-level metrics vs micro patterns (SVM)

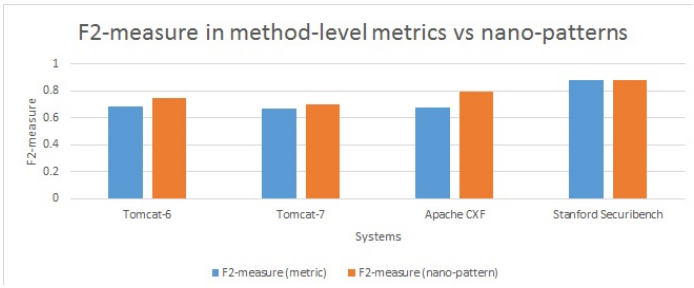


Figure 8.11

F2-measure in method-level metrics vs nano-patterns (SVM)

patterns for the systems under study as shown in Figure 8.12. This result shows that the nano-metrics based model retrieves a higher number of vulnerable methods than the method-level metrics. The lower the FN rate, the higher the recall will be resulting in a greater number of accurately predicted vulnerable methods as in Figure 8.13.

The maximum FN rate is 31 percent (using LR) in nano-metrics, which is 36 percent (using LR) in method-level metrics based model. The lowest Recall is 69 percent (using LR) in nano-metrics, which is 63 percent (using LR) in method-level metrics based model. Sultana et al. in [51] found that some nano-patterns are frequent in vulnerable methods compared to their presence in non-vulnerable methods. They concluded that the methods that read values from objects must be checked for proper sanitization in order to keep the code secure. They statistically determined the significant difference in the presence of nano-patterns in vulnerable and neutral methods. When we combine these patterns with software metrics in order to enable a more gereric assessment for vulnerability prediction, they perform better for certain systems.

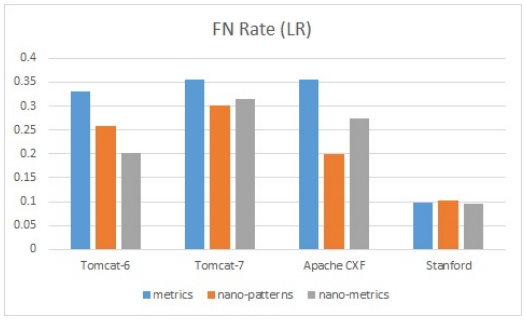


Figure 8.12

Comparative Study on FN Rates (LR)

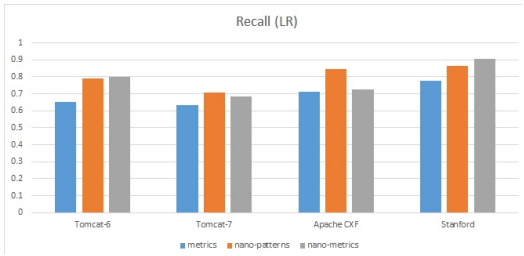


Figure 8.13

Comparative Study on Recall (LR)

In our study, the FN rate and recall are the most significant measures because they determine the percentage of vulnerable code that remains undetected. If neutral code is wrongly determined as vulnerable, it can increase the workload of testers, but if vulnerable code is wrongly determined as neutral, then the model's usefulness is limited for the software team. After analyzing the results, we see that nano-metrics perform better than metrics in retrieving more vulnerable code for all systems. Traditional metrics are not directly related to the code constructs. Shin et al. in [45] empirically showed that complexity metrics can predict vulnerabilities at a low false positive rate but at a high false negative rate. This result indicates that the vulnerable files as detected by metrics were actual vulnerable files, but many other vulnerable files remained undetected. Therefore, we have combined metrics with nano-patterns to improve their recall rate and reduce FN rate.

Precision: Precision indicates the percentage of actual vulnerable code (method or class) in the total vulnerable code predicted as vulnerable by the model. This statistic measures the correctness or the efficiency of the model. An analysis comparing precision among nano-metrics, nano-patterns and method-level metrics using Logistic Regression is presented in Figure 8.14. The figure shows that method-level metrics outperform nano-patterns and nano-metrics. The maximum precision is 86 percent (using LR) in nano-metrics, which is 87 percent (using LR) in method-level metric based model. Nano-metrics show better precision than nano-patterns. The metrics are related to the complexity, cohesiveness or coupling among the functions. Although they can increase the probability of a method being vulnerable, vulnerabilities may not be directly caused by them. Some other factors may be responsible for making a method vulnerable which may not be captured by

these metrics. On the other hand, a neutral method is usually less complex, highly cohesive and less coupled and can be better detected by the metrics. Therefore, metrics do not wrongly predict neutral methods as vulnerable resulting in high precision.

The results show that using metrics as features in vulnerability prediction can be more precise than nano-patterns and nano-metrics. The percentage of actual vulnerable code in the total predicted vulnerable code is higher in the metric based prediction model. This result indicates that software metrics contribute to a higher number of correct results than traceable patterns. We have combined these metrics with nano-patterns so that nano-metrics can perform better in terms of precision by utilizing the properties of software metrics.

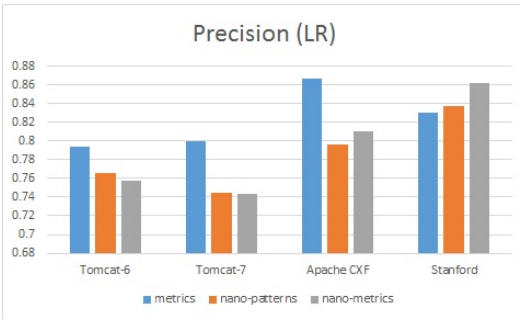


Figure 8.14

Comparative Study on Precision (LR)

F_2-measure: In our study, we assigned more weight to recall as recall is more important in this scenario than precision. We selected F_2-measure as it weighs recall twice as much as precision. If the recall rate is lower, more vulnerable code will remain undetected

or wrongly identified as neutral which is not desirable. Alternatively, if neutral code is detected as vulnerable, precision will be lower. This case may increase the workload of testers, but it is not as harmful as missing a vulnearbility. According to the Figure 8.15, F_2-measure is higher in the nano-patterns and nano-metrics based model compared to the metrics-based model.

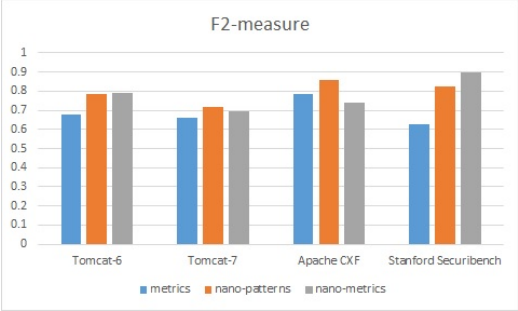


Figure 8.15

Comparative Study on F_2-measure (LR)

8.3.2 Trade-off between Recall and FP Rate

We expect a higher recall with a lower FP rate in predicting vulnerable code. But the FP rate usually increases with the increase of recall. We plot a graph of recall vs. FP rate in order to explore the trade-off between recall and the FP rate. Such plots are known as Receiver Operating Curve (ROC) which are used to visualize the performance of a predictor in detecting the true class (in our case vulnerable methods). Figure 8.16 and Figure 8.17 present the ROC curves using nano-metrics in Logistic Regression and Support Vector Machine respectively. According to the figures, a nano-metrics based model can

correctly identify about 50 percent of the vulnerable methods while keeping the FP rate below 10 percent and about 65 percent (for systems, it goes above 75 percent) of the vulnerable methods when the FP rate is below 20 percent using LR as in Figure 8.16. In SVM, although recall is below 40 percent at FP rate 10 percent for two systems, the measure grows above 75 percent at FP rate 30 percent for them as in Figure 8.17. In the case of the method-level metrics, 70 percent of the vulnerable methods are detected at the FP rate 20 percent but it is below 50 percent at FP rate 10 percent as shown in Figure 8.18.

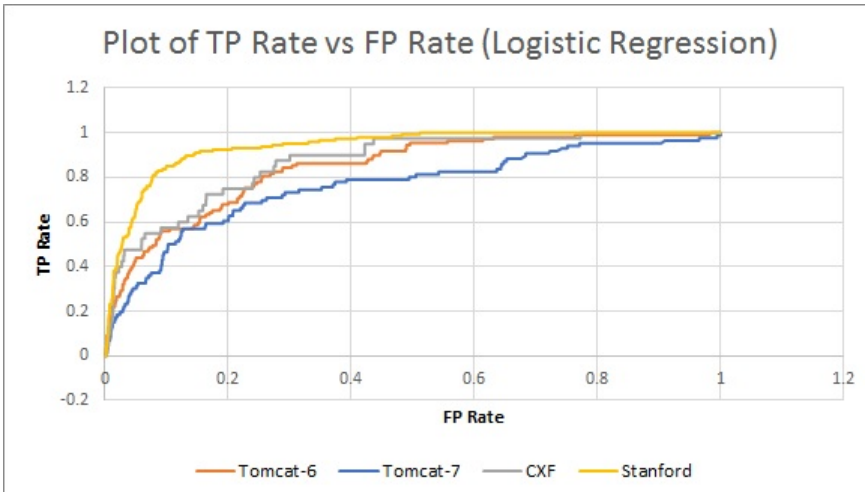


Figure 8.16

Plot of ROC for Nano-metrics in Logistic Regression.

8.3.3 Performance of nano-metrics in Cross-dataset Validation

In order to evaluate the nano-metrics as generic and robust metrics for vulnerability prediction, we trained the machine using the dataset from one system and tested it using the dataset from other systems. In this experiment, we see that the recall rate is more

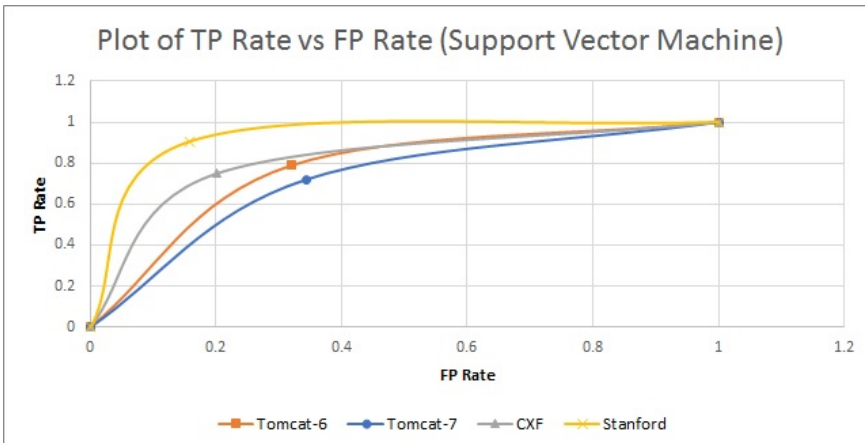


Figure 8.17

Plot of ROC for Nano-metrics in Support Vector Machine.

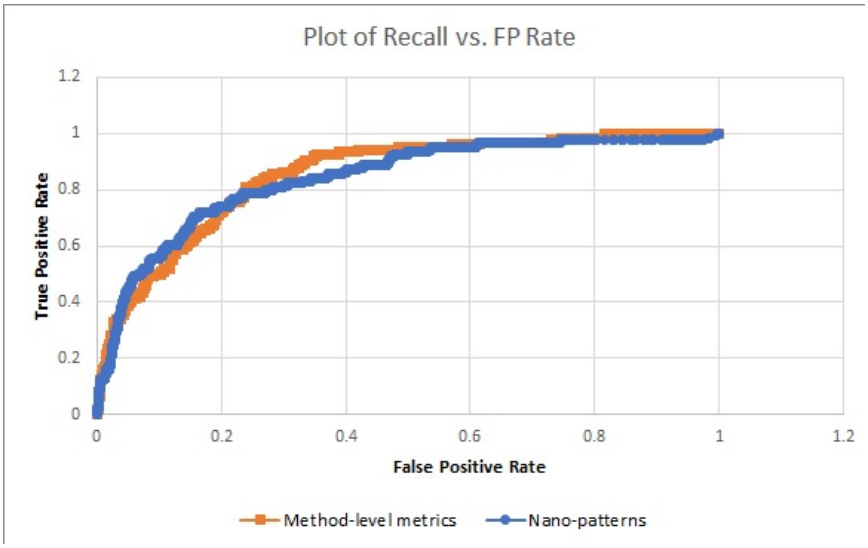


Figure 8.18

Plot of ROC for Nano-patterns and Metrics in Logistic Regression.

than 70 percent using logistic regression as shown in Table 7.28 and Table 7.29. False positive rate is also less than 30 percent in most of the systems under study. Figure 8.19 and Figure 8.21 present ROC curves in cross-dataset validation while using Tomcat-6 and Tomcat-7 as training dataset for logistic regression model respectively. According to these figures, a nano-metrics based model can correctly identify about 60 percent of the vulnerable methods while keeping the FP rate below 15 percent and about 65 percent of the vulnerable methods when the FP rate is below 20 percent (except for Apache CXF) in logistic regression model. On the other hand, in cross-dataset validation, precision is extremely low as in this case, we could not balance the number of vulnerable and neutral methods in test dataset.

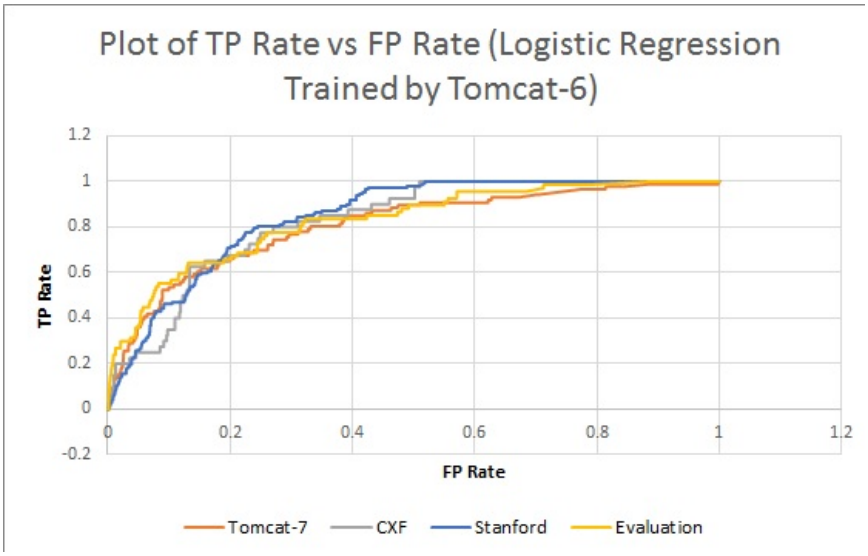


Figure 8.19

Plot of ROC for Nano-metrics in Logistic Regression (Trained by Tomcat-6).

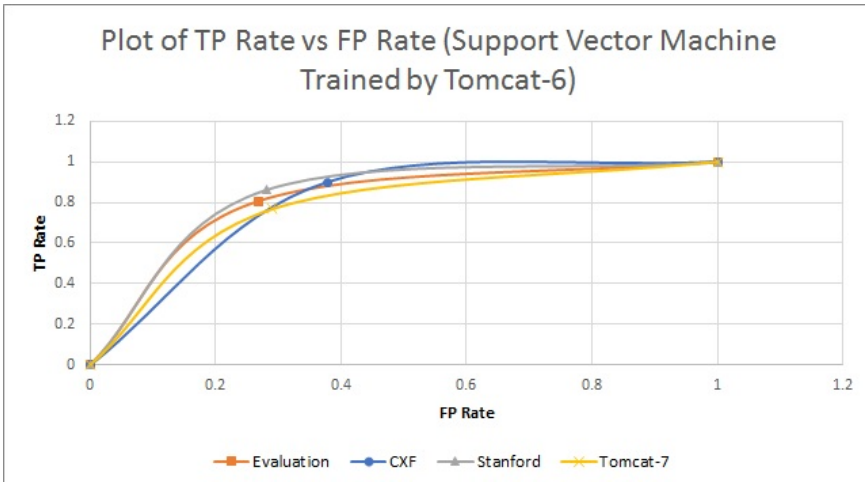


Figure 8.20

Plot of ROC for Nano-metrics in Support Vector Machine (Trained by Tomcat-6).

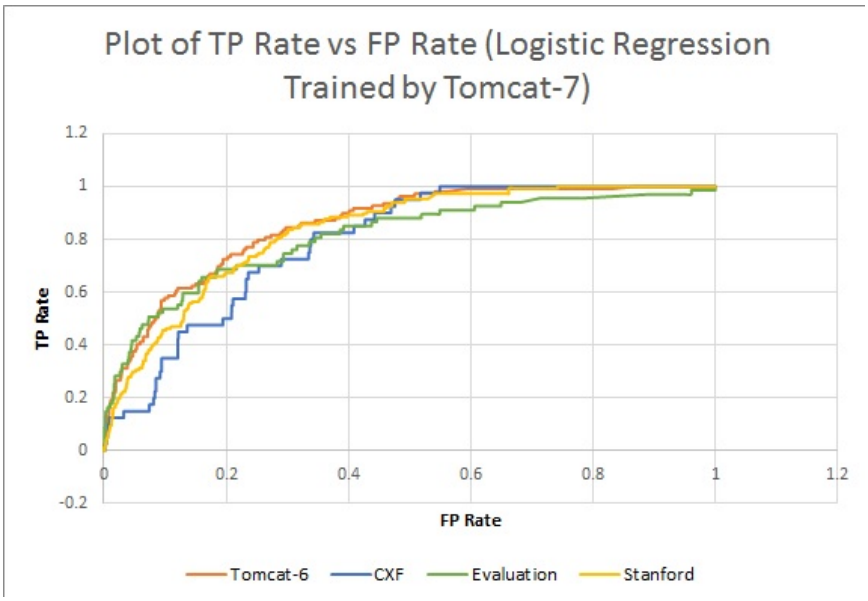


Figure 8.21

Plot of ROC for Nano-metrics in Logistic Regression (Trained by Tomcat-7).

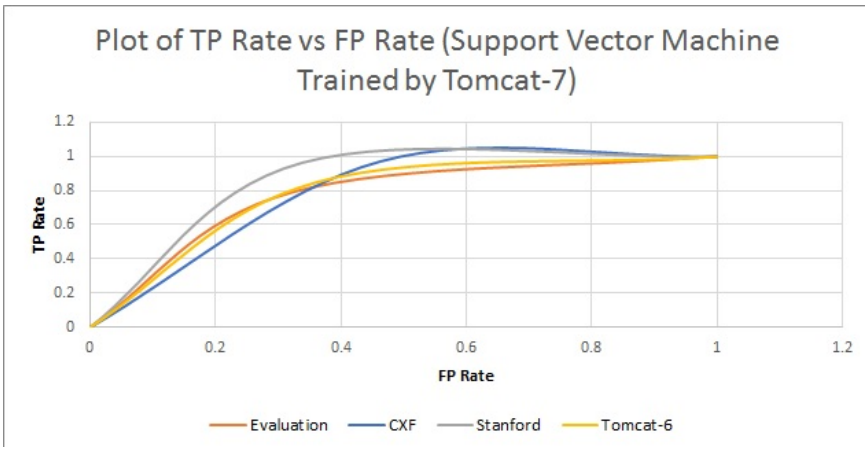


Figure 8.22

Plot of ROC for Nano-metrics in Support Vector Machine (Trained by Tomcat-7).

CHAPTER 9

THREATS TO VALIDITY

In this chapter, we discuss the limitations and threats to the validity of our findings.

9.1 Construct Validity

Construct validity refers to the degree to which a test measures what it claims or purports to be measuring. In this study, micro patterns and nano-patterns are defined based on the formal conditions of the structure of a Java class or a method. These patterns may not be enough to consider all types of characteristics a class or a method may have. The vulnerable classes and methods were found from the Apache project vulnerability reports. We also defined the last version of each release as non-vulnerable to compare the distribution of micro and nano-patterns in vulnerable versus non-vulnerable classes and methods. But the fact is the classes or methods that were considered as non-vulnerable may be reported as vulnerable in a later release. Moreover, for the Stanford project, we assumed some classes or methods to be non-affected as detected by ESVD which may result in some false negatives.

9.2 External Validity

External Validity refers to the ability to generalize results. The experiment was conducted on the systems including Apache, Stanford, and an evaluation dataset. As micro and nano-patterns are defined only for Java classes and methods and vulnerability data is not well formed for other Java projects, we limited this study to these applications. So it cannot be concluded that the results are valid for other systems written in different programming languages or that use different frameworks.

9.3 Internal Validity

This threat refers to the possibility of having unwanted or unanticipated relationships. We are not claiming causation, just relating software vulnerabilities with the presence of micro or nano-patterns. We do not claim that the suspected micro or nano-patterns are the cause of the vulnerability-proneness of the classes or methods, but we recommend rigorous testing of the classes or methods containing those patterns. On the other hand, we encourage the use of patterns that seem to be more frequent in non-vulnerable code, although it cannot be claimed that the code with these patterns will never be susceptible to vulnerability. But we can at least declare that the classes or methods with these patterns will not need rigorous (or extended) testing compared to the classes or methods with other patterns.

CHAPTER 10

CONCLUSION

An overview of the publication plan is presented in Table 10.1.

Table 10.1

Publication List

Paper Title	**Venue**	**Status**
A Preliminary Study Examining Relationships between Nano-Patterns and Software Security Vulnerabilities	IEEE International Conference on Computers, Software and Applications (COMPSAC 2016), June 10-14, 2016	Accepted
Assessing Software Defects Using Nano-Patterns Detection	International Journal of Computers and Their Applications (Special Issue)	Accepted
A Study Examining Relationships Between Micro Patterns and Security Vulnerabilities	Software Quality Journal	Accepted
Correlation Analysis among Java Nano-patterns and Software Vulnerabilities	IEEE International Symposium on High Assurance Systems Engineering (HASE 2017)	Accepted
Evaluating Micro Patterns and Software Metrics in Vulnerability Prediction	Software Mining Workshop, IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)	Accepted
Towards a software vulnerability prediction model using traceable code patterns and software metrics	Doctoral Symposium of the 32nd IEEE/ACM International Conference on Automated Software Engineering	Accepted
The Relationship between Code Smells and Traceable Patterns - Are They Measuring the Same Thing?	International Journal on Software Engineering and Knowledge Engineering (IJSEKE 2017)	Accepted
The Relationship between Traceable Code Patterns and Code Smells	International Conference on Software Engineering and Knowledge Engineering (SEKE 2017)	Accepted
Evaluating the Performance of nano-patterns and software metrics in vulnerability prediction	The ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM 2018)	Submitted
Proposed Model and its Evaluation	IEEE Transactions on Software Engineering (TSE)	In Progress
Study on Code Smell and Vulnerabilities	Journal 2018	In Progress

The timeline for completing the proposed plan is shown in Table 10.2.

In this study, we conducted experimental analysis on micro and nano-patterns extracted from vulnerable and neutral code in order to find out any hidden relationship between patterns and vulnerabilities in order to build a vulnerability prediction model. Correlating patterns with vulnerabilities will guide developers to use them properly to avoid security issues in Java source code. Testers will also be able to better ensure time and cost effective testing as they can pay closer attention to code where there is a higher likelihood for

Table 10.2

Dissertation Timeline

Tasks	**Deadline**
Extract Nano-patterns and Method-level metrics	January 2018
Correlation between Nano-patterns and Method-level metrics	January 2018
Build Universal Prediction Model	February 2018
Build Project-Specific Prediction Model	February 2018
Model Evaluation	March 2018 - April 2018
Write up	June 2017 - June 2018
Defense	June 2018

a security threat. They will be able to concentrate their efforts on the classes or methods having unsafe patterns. We also uncovered micro pattern evolution across the releases as well as from vulnerable version to non-vulnerable version as vulnerabilities were fixed. The evolution type results strengthen our claim regarding the relationship between micro patterns and vulnerability. In addition, we extracted the association between pairs of micro and nano-patterns in vulnerable versus non-vulnerable code so that we can make an assumption about their collaborative effect on making a class or a method vulnerable or non-vulnerable. In the future, we aim to measure the micro and nano-patterns in numeric scale rather than binary scale in order to extend their use in software security.

In the second part of this research, we analyzed traceable patterns and software metrics to predict software vulnerabilities. This is the first study comparing the performance of these patterns in vulnerability prediction with traditional metrics. The software metrics outperformed traceable patterns in determining neutral code with a higher precision and recall. The recall rate of predicting vulnerable classes or methods is high when using patterns as features. This observation indicates that patterns can be well suited as features

for detecting vulnerable code as opposed to neutral code. This study will allow developers and testers to compare these two types of features and decide the best for their projects based on the project characteristics. The results will better enable secure software by giving the software team the ability to target tests to potentially vulnerable files and allow for early detection of secuirty risks. This research will open up new areas to investigate metrics at different granularity levels (class or method-level) so that they can be used more precisely in improving software security. We proposed a framework consisting of a combination of nano-patterns and method-level metrics (nano-metrics). These metrics were evaluated and shown to perform better in terms of recall and F_2-measure compared to the metrics and nano-patterns alone. Moreover, they were cross-validated to work for all the systems under study in the same way. As a result, developers do not need to bother about training a machine with their own vulnerability history if resources are constrained. Rather, they can use a previously trained machine for classifying their vulnerable code. Developers will be able to develop more reliable code by re-engineering later versions using safe patterns. In this work, we examined Java based systems which released their own vulnerability history. In the future, we will study Android (currently, the most widely used operating system in the world) to investigate how nano-metrics work for vulnerability prediction in mobile applications. In the future, we will extend this work for other frameworks by identifying universal code patterns that might be related to software vulnerabilities across multiple systems. In addition, we plan to develop specialized traceable patterns for that specifically target vulnerabilities.

REFERENCES

[1] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Class Designs," *Proceedings of the 9th International Conference on Quality Software*, 2009, pp. 11–20.

[2] H. Alves, B. Fonseca, and N. Antunes, "Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study," *2016 12th European Dependable Computing Conference (EDCC)*, 2016, pp. 37–44.

[3] F. Batarseh, "Java Nano Patterns: A Set of Reusable Objects," *Proceedings of the 48th Annual Southeast Regional Conference*, New York, NY, USA, 2010, ACM SE '10, pp. 60:1–60:4, ACM.

[4] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, June 1994, pp. 476–493.

[5] N. Chinchor, "MUC-4 Evaluation Metrics," *Proceedings of the 4th Conference on Message Understanding*, Stroudsburg, PA, USA, 1992, MUC4 '92, pp. 22–29, Association for Computational Linguistics.

[6] I. Chowdhury, B. Chan, and M. Z. Chowdhury, "Security Metrics for Source Code Structures," *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, New York, NY, USA, 2008, SESS '08, pp. 57–64, ACM.

[7] I. Chowdhury and M. Zulkernine, "Can Complexity, Coupling, and Cohesion Metrics Be Used As Early Indicators of Vulnerabilities?," *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010, SAC '10, pp. 1963–1969, ACM.

[8] I. Chowdhury and M. Zulkernine, "Using Complexity, Coupling, and Cohesion Metrics As Early Indicators of Vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, mar 2011, pp. 294–313.

[9] H. Cramér, *Mathematical Methods of Statistics*, Princeton University Press, Princeton, 1946.

[10] A. Deo and B. J. Williams, "Preliminary Study on Assessing Software Defects Using Nano-Pattern Detection," *Proceedings of the 24th International Conference on Software Engineering and Data Engineering (SEDE)*, 2015.

[11] G. Destefanis, *Assessing software quality by micro patterns detection*, doctoral dissertation, 2012.

[12] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, "Micro Pattern Fault-Proneness," *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 2012, SEAA '12, pp. 302–306, IEEE Computer Society.

[13] J. Ekstrm, "The Phi-coefficient, the Tetrachoric Correlation Coefficient, and the Pearson-Yule Debate,".

[14] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd edition, PWS Publishing Co., Boston, MA, USA, 1998.

[15] F. A. Fontana, B. Walter, and M. Zanoni, "Code smells and micro patterns correlations," *RefTest 2013 Workshop, co-located event with XP 2013 Conference*, 2013.

[16] S. M. Ghaffarian and H. R. Shahriari, "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey," *ACM Comput. Surv.*, vol. 50, no. 4, 2017, pp. 56:1–56:36.

[17] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level Bug Prediction," *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2012, ESEM '12, pp. 171–180, ACM.

[18] J. Y. Gil and I. Maman, "Micro Patterns in Java Code," *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 2005, OOPSLA '05, pp. 97–116, ACM.

[19] M. G. Graff and K. R. V. Wyk, *Secure Coding: Principles and Practices*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[20] J. Han, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[21] W. A. Harrison and K. I. Magel, "A Complexity Measure Based on Nesting Level," *SIGPLAN Not.*, vol. 16, no. 3, Mar. 1981, pp. 63–74.

[22] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, Sept. 1981, pp. 510–518.

[23] M. Howard and D. E. Leblanc, *Writing Secure Code*, 2nd edition, Microsoft Press, Redmond, WA, USA, 2002.

[24] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Micro Pattern Evolution," *Proceedings of the 2006 International Workshop on Mining Software Repositories*, New York, NY, USA, 2006, MSR '06, pp. 40–46, ACM.

[25] V. B. Livshits, "Findings Security Errors in Java Applications Using Lightweight Static Analysis,", Work-in-Progress Report, Annual Computer Security Applications Conference, nov 2004.

[26] V. B. Livshits and M. S. Lam, "Finding Security Errors in Java Programs with Static Analysis," *Proceedings of the 14th Usenix Security Symposium*, aug 2005, pp. 271–286.

[27] M. Lorenz and J. Kidd, *Object-oriented Software Metrics: A Practical Guide*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[28] S. Maggioni and F. Arcelli, "Metrics-based Detection of Micro Patterns," *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, New York, NY, USA, 2010, WETSoM '10, pp. 39–46, ACM.

[29] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, July 1976, pp. 308–320.

[30] T. M. Mitchell, *Generative and discriminative classifiers: naive bayes and logistic regression*, McGraw-Hill, Inc., NY, USA.

[31] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with Applying Vulnerability Prediction Models," *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, New York, NY, USA, 2015, HotSoS '15, pp. 4:1–4:9, ACM.

[32] S. Moshtari and A. Sami, "Evaluating and Comparing Complexity, Coupling and a New Proposed Set of Coupling Metrics in Cross-project Vulnerability Prediction," *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2016, SAC '16, pp. 1415–1421, ACM.

[33] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, vol. 5, 2013, pp. 8–17.

[34] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan, "Do Bugs Foreshadow Vulnerabilities? An In-depth Study of the Chromium Project," *Empirical Softw. Engg.*, vol. 22, no. 3, June 2017, pp. 1305–1347.

[35] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," *Proceedings of the 27th international conference on Software engineering*, 2005.

[36] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting Vulnerable Software Components," *Proceedings of the 14th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2007, CCS '07, pp. 529–540, ACM.

[37] V. H. Nguyen and L. M. S. Tran, "Predicting Vulnerable Software Components with Dependency Graphs," *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, New York, NY, USA, 2010, MetriSec '10, pp. 3:1–3:8, ACM.

[38] E. S. R. Gopalakrishna and J. Vitek, *Vulnerability likelihood: A probabilistic approach to software assurance*, technical report, 2005.

[39] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, Oct 2014, pp. 993–1006.

[40] R. Seacord, *Secure Coding in C and C++*, first edition, Addison-Wesley Professional, 2005.

[41] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman & Hall/CRC, 2007.

[42] Y. Shin, "Exploring complexity metrics as indicators of software vulnerability," *The 3rd International Doctoral Symposium on Empirical Software Engineering (IDoESE 2008) , co-located with ESEM-2008*, 2008.

[43] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics As Indicators of Software Vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, nov 2011, pp. 772–787.

[44] Y. Shin and L. Williams, "An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics," *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2008, ESEM '08, pp. 315–317, ACM.

[45] Y. Shin and L. Williams, "Is Complexity Really the Enemy of Software Security?," *Proceedings of the 4th ACM Workshop on Quality of Protection*, New York, NY, USA, 2008, QoP '08, pp. 47–50, ACM.

[46] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?," *Empirical Software Engineering*, vol. 18, no. 1, 2013, pp. 25–59.

[47] J. Singer, G. Brown, M. Lujn, A. Pocock, and P. Yiapanis, "Fundamental Nano-Patterns to Characterize and Classify Java Methods," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, 2010, pp. 191 – 204, Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[48] J. Singer and C. Kirkham, "Exploiting the Correspondence between Micro Patterns and Class Names," *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. Sept. 2008, pp. 67–76, IEEE.

[49] B. Smith and L. Williams, "On the Effective Use of Security Test Patterns," *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 108–117.

[50] K. Z. Sultana, A. Deo, and B. J. Williams, "A Preliminary Study Examining Relationships Between Nano-Patterns and Software Security Vulnerabilities," *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[51] K. Z. Sultana, A. Deo, and B. J. Williams, "Correlation Analysis among Java Nano-Patterns and Software Vulnerabilities," *IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, 2017.

[52] K. Z. Sultana and B. J. Williams, "Evaluating micro patterns and software metrics in vulnerability prediction," *2017 6th International Workshop on Software Mining (SoftwareMining)*, Nov 2017, pp. 40–47.

[53] K. Z. Sultana, B. J. Williams, and T. Bhowmik, "A study examining relationships between micro patterns and security vulnerabilities," *Software Quality Journal*, 2017, pp. 1–37.

[54] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[55] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," *2014 IEEE 25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 23–33.

[56] D. A. Wheeler, *Secure Programming for Linux and Unix HOWTO*, 2003.

[57] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, San Francisco, USA, 2005.

[58] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Functionwith an Existing Exploit," *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, New York, NY, USA, 2016, CODASPY '16, pp. 97–104, ACM.

[59] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, "Combining Software Metrics and Text Features for Vulnerable File Prediction," *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015, pp. 40–49.

[60] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010, ICST '10, pp. 421–428, IEEE Computer Society.