Towards a Software Vulnerability Prediction Model using Traceable Code Patterns and Software Metrics

Kazi Zakia Sultana
Department of Computer Science and Engineering
Mississippi State University, MS, USA

***Abstract*—Software security is an important aspect of ensuring software quality. The goal of this study is to help developers evaluate software security using traceable patterns and software metrics during development. 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 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. This study explores the performance of some code patterns in vulnerability prediction and compares them with traditional software metrics. We use the findings to build an effective vulnerability prediction model. We evaluate security vulnerabilities reported for Apache Tomcat, Apache CXF and three stand-alone Java web applications. We use machine learning and statistical techniques for predicting vulnerabilities using traceable patterns and metrics as features. We found that patterns have a lower false negative rate and higher recall in detecting vulnerable code than the traditional software metrics.**

I. Introduction

Security is a major concern in the developer community. If software can be exploited by malicious attacks, its reliability suffers. There are different security policies and practices followed in different phases of the software development lifecycle. For the implementation phase, secure coding practices are followed to create a robust codebase free from common security weaknesses[1]. While these coding practices have been vetted by the software engineering community, following these practices does not ensure vulnerability free code. There is need for methods that can highlight certain code constructs that exhibit vulnerable characteristics of that code area. The existing security metrics at the code level are basically software metrics and they have high false negative rates [9], [10]. Moreover, these metrics do not consider the correlations of code constructs with vulnerabilities and therefore, the developers do not obtain any guidelines for secured coding practices. Our goal is to build a vulnerability prediction model using code constructs, the focus is on class-level and method-level code patterns to highlight potentially vulnerable areas in the code. The suspected code areas having vulnerability-prone patterns can be marked for targeted testing or rigorous reviews based on the vulnerability history of the project.

[1]https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf

Gil et al. [5] 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. defined 27 micro patterns organized into eight categories with respect to the formal conditions on the structure of Java classes. For example, $DataManager$ pattern is a set of setter and getter methods which encapsulates all its fields and controls the access to these fields. $Extender$ is a class which extends the interface inherited from its superclass and super interfaces, but does not override any method. Nano-patterns are method-level patterns and capture properties of methods within a class. Batarseh first introduced the idea of nano-patterns in [1]. Singer et al. [4] listed 17 fundamental nano-patterns organized into four groups. The fault-proneness of these patterns has been studied in several research [2], [3], [7], [8]. They focused on improving software quality by reducing the use of bug-prone patterns.

Kim et al. examined how micro patterns evolved from buggy code to non-buggy code and detected the evolution types that are more bug-prone than others [6]. In our study, we analyzed how micro patterns are changed from vulnerable classes to neutral classes (We use the term 'neutral' for the classes and methods where the reported vulnerabilities have already been fixed). We computed the phi-coefficient between each pair of patterns to find their association in vulnerable and neutral code. This analysis will help developers to be concerned about the connected patterns in code. When the developer uses one pattern from the connected pair, he should avoid the use of its peer pattern to reduce the risk of security violation.

A variety of metrics have been developed for defining code characteristics and assessing software quality [11], [12]. These metrics were later used for measuring software security [9], [10]. Although they can assess overall software quality accurately, they did not specify any particular granularity level of the vulnerable files. Moreover, they did not distinguish between class-level and method-level software metrics in their study. Security at a particular granular level can help developers to tag the vulnerable code easily. Moreover, these metrics do not consider the correlations of code constructs with vulnerabilities resulting in high false negative rates and therefore, the developers do not get any guideline for secured coding practices. On the other hand, traceable patterns are directly related to code constructs and the suspected code

having bad patterns can be marked for targeted testing or more rigorous reviews. In this study, we aim to build a vulnerability prediction model at class and method level that will be able to classify a class or method as vulnerable or neutral by utilizing the relations among patterns, metrics and vulnerabilities.

II. Research Proposal

A. Research Goal and Questions

The goal of this research is to develop a vulnerability prediction framework that reduces false negative rates using traceable patterns and code-level software metrics. 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 for testing them.

1) RQ1: What is the relation between traceable patterns and software vulnerabilities?: This question determines the correlation between traceable patterns and vulnerabilities. We answer this question by exploring the distribution of patterns in both vulnerable and neutral code. We obtain the phi-coefficient for each pair of patterns in vulnerable and neutral code. Phi-coefficient is used to measure the degree of association between two binary variables [14]. We also find how traceable patterns evolve from vulnerable code to neutral code.

2) RQ2: Can traceable patterns better predict vulnerable code than software metrics?: The results of the experiments related to this question evaluate traceable patterns and software metrics as predictors for vulnerabilities. We analyzed our data using machine learning algorithms and ran them separately for the patterns and the metrics feature set. We used 10-fold cross-validation to ensure that the trained model will work accurately for an unknown dataset in practice. The performance measures including False Negative rate, Recall, Precision, F-measure while using patterns and metrics as features for classifying a code as vulnerable or neutral have been measured.

3) RQ3: How do we build a framework to determine the most significant set of patterns and metrics for vulnerability prediction?: This question determines the correlation between patterns and software metrics in vulnerable versus neutral code. The highly correlated patterns and metrics will be examined to determine their correlation is significantly different between vulnerable and neutral code. Those pattern-metrics pairs can be used as features for vulnerability prediction. We will present various performance measures such as False Negative rate, Recall, Precision, F-measure while using this set as features for classifying a code as vulnerable or neutral.

4) RQ4: How can we evaluate the framework?: Once we have developed our framework, we will determine its effectiveness in predicting vulnerabilities. We will use vulnerability data extracted from different versions of open source software to train our predictive model. We will validate our framework by using the remaining versions of the software and compare the results with existing prediction tools / frameworks.

B. Study Design

We used three different projects: Apache Tomcat[2], Apache CXF[3], and the Stanford Securibench dataset[4]. Apache projects are 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 a Java based system 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. We considered 14 versions of Tomcat-6, 18 versions of Tomcat-7, 7 versions of Tomcat-8, and 12 versions of Apache CXF as the reported vulnerabilities were detected across these versions. The source code of Apache projects are stored in Archives[5]. Stanford SecuriBench is a set of open source programs to be used as a testing ground for static and dynamic security tools [13]. We conducted the experiment for traceable patterns on the J2EE web applications known as Blueblog 1.0, Pebble 1.6-beta1, Personalblog 1.2.6 and Roller 0.9.9 in Stanford dataset.

We used a static analyzer tool called Early Security Vulnerability Detector - ESVD to identify vulnerable classes in Stanford as it was shown to have fewer false positives with higher precision and recall for the Stanford projects. We used the Eclipse plugin of the tool. The micro patterns were extracted using a pattern extraction tool developed in [5]. The command line tool is available at Maman's webpage[6]. Singer et al. in [4] 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 [7], that extracts the methods modified for each revision number involved in fixing a vulnerability. JiraExtractor actually uses the nano-pattern tool developed by Singer in [4] to pull the nano-patterns' information of the methods. We used Understand 4.0[7] for extracting software metrics in our study. The phi-coefficients were measured using the SPSS tool[8]. We used WEKA 3.8 for building vulnerability prediction model[9].

C. Progress

We completed the experiments on research question 1 covering the distribution and association of micro patterns and nano-patterns as well as their evolution in vulnerable and neutral code. We also analyzed the performance of these patterns in vulnerability prediction and compared them with software metrics for answering research question 2. Our next plan is to answer remaining two questions by building an prediction model based on these results and evaluate the model.

[2]https://tomcat.apache.org/
[3]http://cxf.apache.org/
[4]https://suif.stanford.edu/ livshits/securibench/stats.html
[5]http://archive.apache.org/dist/
[6]http://www.cs.technion.ac.il/ imaman/mp/download.html
[7]http://www.scitools.com
[8]http://www-01.ibm.com/software/analytics/spss/products/statistics/index.html
[9]http://www.cs.waikato.ac.nz/ml/weka

D. Initial Results

We found that some micro patterns including $CompoundBox$, $Immutable$, $Implementor$, $Overrider$, $Sink$, $Stateless$, $FunctionObject$, and $LimitedSelf$ have statistically significant presence in neutral classes compared to their presence in vulnerable classes. $Outline$ and $AugmentedType$ are significantly present in vulnerable classes. Gil et al. [5] 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 $AugmentedType$. 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. We found several connected pairs of micro and nano-patterns in vulnerable and neutral versions of the systems. We also analyzed all the vulnerable classes of a particular version that contained vulnerable code, then analyzed those classes in the neutral version where the vulnerability had been fixed. We found that some evolution types such as $None \rightarrow Implementor$, $CommonState \rightarrow Stateless$ are more frequent compared to other types on evolution. For example, $CommonState \rightarrow Stateless$ indicates that vulnerable code contained $CommonState$ pattern which later changed to $Stateless$ pattern after vulnerability fix.

The False Negative (FN) rate, Precision, Recall and F-Measure of micro and nano-patterns based prediction model for Tomcat-7 is presented in Table I. The same information can be found for the class and method-level metrics in Table II.

TABLE I: MACHINE LEARNING RESULTS FOR TRACEABLE PATTERNS IN TOMCAT (RELEASE 7)

	Method	FN Rate	Precision	Recall	F-Measure
Micro Patterns	Logistic Regression	0.113	0.640	0.887	0.823
	SVM	0.104	0.644	0.896	0.831
Nano-patterns	Logistic Regression	0.292	0.744	0.708	0.715
	SVM	0.302	0.730	0.698	0.704

TABLE II: MACHINE LEARNING RESULTS FOR SOFTWARE METRICS IN TOMCAT (RELEASE 7)

	Method	FN Rate	Precision	Recall	F-Measure
Class metrics	Logistic Regression	0.125	0.888	0.875	0.878
	SVM	0.143	0.884	0.857	0.862
Method metrics	Logistic Regression	0.366	0.800	0.634	0.661
	SVM	0.356	0.794	0.644	0.669

E. Proposed Model

After collecting and extracting the data we require, we need to classify the code. We plan to devise two workflows as shown in Figure 1. If we want to test code from a new system, we can either use an already tuned machine based on the historic data of our analyzed systems (universal classification) or we can train the machine with the historic data of the new system (project-specific classification).

1) Universal Classification: We will build a dataset consisting of nano-patterns and method-level metrics to better predict vulnerable methods. For this approach, we will compute Point-Biserial Correlation[10] among the nano-patterns and metrics in vulnerable and neutral methods. The goal is to find highly correlated patterns and metrics in vulnerable and neutral code and utilize them to build a prediction model. We will train our model with the data and evaluate their performance in vulnerability prediction. If they exhibit high recall and precision and minimal false negative rates, these patterns and metrics will be recommended to the developers for testing their code. As this set of metrics and patterns will be utilized for vulnerability prediction in any system under testing, this classification will be known as "Universal Classification".

2) Project-Specific Classification:

- **Build Prediction Model:** Developers will need to run Welch's test for finding the nano-patterns that exhibit significantly different distributions in vulnerable and neutral code. Welch's Test for unequal variances (also called Welch's t-test or unequal variances t-test) is a modification of a Student's t-test to see if two sample means are significantly different[11]. After analyzing their historic vulnerable data, they will find the significant nano-patterns to be used for their system which can better be used for training a machine to build prediction model than the metrics identified in the universal approach. Significant nano-patterns for their target systems will be able to better predict vulnerable code of later releases.
- **Classification:** After building the prediction model, the code for testing will be fed into the machine. The nano-patterns for that code will be selected and used for its classification in that prediction model. As it depends on the historic vulnerability data of the respective system, it is called "Project-Specific Classification".

III. CONTRIBUTION

The published papers are listed in Table III. The major contributions of this research are as follows:

- We analyzed the distribution of micro and nano-patterns in vulnerable code and code where no vulnerabilities have been reported. Our comparative analysis on the distribution of traceable patterns will assist the developers to be more restrictive in the usage of the patterns as well as guide them to inspect their code for potential vulnerabilities. The findings will help testers become more focused on the potentially vulnerable areas of code instead of the entire code base and ensure efficient testing.
- Our vulnerability study at the class and method level will help developers to concentrate at lower granularity levels of code and pinpoint the vulnerable components more easily. They will be able to ensure reliable system evolution by re-engineering the later versions with the proper usage of these code constructs.

[10]https://statistics.laerd.com/spss-tutorials/point-biserial-correlation-using-spss-statistics.php#procedure

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

TABLE III: PUBLICATION LIST

Authors	Title	Venue
K. Z. Sultana, A. Deo, B. J. Williams	Correlation Analysis among Java Nano-patterns and Software Vulnerabilities	HASE 2017 (Accepted)
K. Z. Sultana, A. Deo, B. J. Williams	A Preliminary Study Examining Relationships between Nano-Patterns and Software Security Vulnerabilities	COMPSAC 2016 (Accepted)
Z. Codabux, K. Z. Sultana, B. J. Williams	The Relationship between Traceable Code Patterns and Code Smells	SEKE 2017 (Accepted)
A. Deo, Z. Codabux, K. Z. Sultana, B. J. Williams	Assessing Software Defects Using Nano-Patterns Detection	International Journal of Computers and Their Applications (Special issue), 2016 (Accepted)
K. Z. Sultana, B. J. Williams	A Study Examining Relationships between Micro Patterns and Security Vulnerabilities	Software Quality Journal (Submitted)

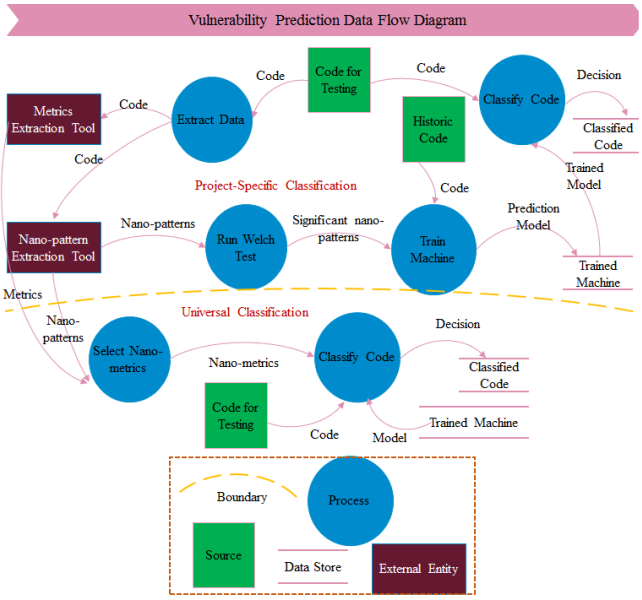

Fig. 1: Data Flow Diagram of the Proposed Model

- We identify a vulnerability prediction model using the micro and nano-patterns extracted from the vulnerable and neutral code data. We execute machine learning techniques to classify vulnerable code. This study will help practitioners to understand how different patterns contribute to the vulnerability proneness of code.
- We train a vulnerability prediction model using traditional class and method-level software metrics extracted from the vulnerable and neutral classes and methods respectively. This study will explore when the traditional metrics can better perform in vulnerability prediction.
- Finally, we propose a prediction model that can be utilized for predicting vulnerability in a new code. In this model, developers can choose either universal or project-specific method for their code under testing. Project-specific approach will allow them to train the machine with their own historic data whereas universal approach will offer them to use an already trained machine.

IV. CONCLUSION

This research analyzes traceable patterns and metrics to predict software vulnerabilities. We found that patterns based prediction model can better predict vulnerable code in terms of recall. This is the first study comparing the performance of patterns in vulnerability prediction with traditional metrics. This study will allow developers and testers to ensure software security by giving them the ability to target tests to potentially vulnerable files and allowing early detection of risks. Moreover, it proposes a prediction model based on the results. In this work, we examined Java based systems. In the future, we will extend this work for other frameworks by finding some universal code patterns that might be related to software vulnerabilities across multiple systems. In addition, we plan to develop some specialized traceable patterns for vulnerabilities which will only target the security issues of code.

REFERENCES

[1] F. Batarseh, *Java nano patterns: a set of reusable objects*, Proc. of the 48th Annual Southeast Regional Conference, New York, NY, USA, 2010.

[2] K. Z. Sultana, A. Deo, and B. J. Williams, *A Preliminary Study Examining Relationships between Nano-Patterns and Software Security Vulnerabilities*, Proc. of the 40th IEEE Computer Society International Conference on Computers, Software and Applications, Atlanta, GA, USA, June 10-14, 2016.

[3] K. Z. Sultana, A. Deo, and B. J. Williams, *Correlation Analysis among Java Nano-patterns and Software Vulnerabilities*, Proc. of the 18th IEEE International Symposium on High Assurance Systems Engineering, Singapore, Jan 12-14, 2017.

[4] J. Singer, G. Brown, M. Lujn, A. Pocock and P. Yiapanis, *Fundamental Nano-Patterns to Characterize and Classify Java Methods*, Journal Electronic Notes in Theoretical Computer Science archive, Vol. 253 Issue 7, pp. 191-204, September, 2010.

[5] J. Gil and I. Maman, *Micro patterns in Java code*, Proc. of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA, Oct 16-20, 2005.

[6] S. Kim, K. Pan, and E. Whitehead Jr., *Micro pattern evolution*, Proc. of the Intl. Workshop on Mining Software Repositories, pp. 40-46, 2006.

[7] A. Deo, and B. J. Williams, *Preliminary Study on Assessing Software Defects Using Nano-Pattern Detection*, Proc. of the 24th International Conference on Software Engineering and Data Engineering (SEDE), San Diego, CA, Oct 12-14, 2015.

[8] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, *Micro Pattern Fault-Proneness*, Proc. of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 302-306, 2012.

[9] Y. Shin, A. Meneely, L. Williams and J. Osborne, *Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities*, IEEE Transactions on Software Engineering, vol. 37, no. 6, pp. 772-787, Nov/Dec, 2011.

[10] I. Chowdhury and M. Zulkernine, *Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities*, Journal of Systems Architecture, Vol. 57, Issue 3, March 2011, Pages 294-313.

[11] S.R. Chidamber, and C.F. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.

[12] T.J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering, vol. 2, no. 4, pp.308-320, 1976.

[13] V. B. Livshits and M. S. Lam, *Finding security errors in Java programs with static analysis*, In Proc. of the 14th Usenix Security Symposium, pp. 271-286, Aug, 2005.

[14] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.