A Comparison of Nano-patterns Vs. Software Metrics in Vulnerability Prediction

Kazi Zakia Sultana
Computer Science and Engineering
Montclair State University
Montclair, NJ, USA
sultanak@montclair.edu

Byron J. Williams
Computer & Info. Sci. & Engineering
University of Florida
Gainesville, FL, USA
byron.williams@ufl.edu

Amiangshu Bosu
Computer Science
Wayne State University
Detroit, MI, USA
amiangshu.bosu@wayne.edu

Abstract*—*Context: **Software security is an imperative aspect of software quality. Early detection of vulnerable code during development can better ensure the security of the codebase and minimize testing efforts. Although traditional software metrics are used for early detection of vulnerabilities, they do not clearly address the granularity level of the issue to precisely pinpoint vulnerabilities. The goal of this study is to employ method-level traceable patterns (nano-patterns) in vulnerability prediction and empirically compare their performance with traditional software metrics. The concept of nano-patterns is similar to design patterns, but these constructs can be automatically recognized and extracted from source code. If nano-patterns can better predict vulnerable methods compared to software metrics, they can be used in developing vulnerability prediction models with better accuracy.** ***Aims:*** **This study explores the performance of method-level patterns in vulnerability prediction. We also compare them with method-level software metrics.** ***Method:*** **We studied vulnerabilities reported for two major releases of Apache Tomcat (6 and 7), Apache CXF, and two stand-alone Java web applications. We used three machine learning techniques to predict vulnerabilities using nano-patterns as features. We applied the same techniques using method-level software metrics as features and compared their performance with nano-patterns.** ***Results:*** **We found that nano-patterns show lower false negative rates for classifying vulnerable methods (for Tomcat 6, 21% vs 34.7%) and therefore, have higher recall in predicting vulnerable code than the software metrics used. On the other hand, software metrics show higher precision than nano-patterns (79.4% vs 76.6%).** ***Conclusion:*** **In summary, we suggest developers use nano-patterns as features for vulnerability prediction to augment existing approaches as these code constructs outperform standard metrics in terms of prediction recall.**

***Index Terms*—Vulnerability, software security, software metrics, nano-pattern**

I. Introduction

Software Security is a major software quality concern. Security vulnerabilities make software susceptible to security exploits, resulting in high organizational risks. "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"[1]. For example, cross-site scripting (XSS) allows remotely authenticated users to inject arbitrary web scripts or HTML, denial of service refuses user request for a service, and injection flaws occur when untrusted data is sent to an interpreter as part of a command or query[2]. Software security is a dominant factor in mitigating those risks. An emphasis on security should pervade each phase of the software development lifecycle. For each phase, there are certain security practices that allow developers to build security into the system. For example, during the implementation phase, secure coding practices can be followed to create a robust code-base free from common security weaknesses (e.g., SQL injection). Researchers have developed security patterns and guidelines for developers to follow during the implementation phase [1]–[4] that guide developers in addressing common security issues in a similar fashion as design patterns for development issues. Another way to reduce the amount of vulnerable code being released is to analyze the source code repository to find code where vulnerabilities have been reported. This historical data can then be used to build a vulnerability prediction model using extracted code constructs that have been commonly associated with vulnerabilities [5]. In this study, we employed traceable, method-level software patterns extracted from source code and determined their relationship with vulnerabilities. This research is motivated by earlier works where code-level software metrics were used to predict vulnerabilities [5]–[12]. We present the performance of these patterns in vulnerability prediction and empirically compare their performance with a set of software metrics. Sultana *et* al. in [13], [14] introduced studying correlations between vulnerabilities and nano-patterns. They identified a set of patterns that are more prone to vulnerability than others. In this paper, we have identified statistically relevant patterns by analyzing historical data and applied machine learning techniques to determine their ability to predict which methods are vulnerable across a code-base. Our objective is *to compare the performance of nano-patterns in classifying vulnerable code with that of traditional software metrics.*

Class-level traceable patterns are called *micro patterns* whereas method-level traceable patterns have been named *nano-patterns*. Micro patterns are defined using formal conditions of the structure of a Java class [15]. Nano-patterns are categorized based on the properties of methods within a class. Singer *et* al. [16] listed 17 fundamental nano-patterns

[1] https://cve.mitre.org/about/terminology.html

[2] https://www.owasp.org/index.php/Top_10_2013-Top_10

organized into four groups: calling, object-oriented, control flow and data flow. In this paper, we developed a vulnerability prediction model featuring method-level patterns and compared the prediction accuracy with another model trained with traditional method-level software metrics. A variety of metrics have been developed to measure security [5]–[12]. Chowdhury *et* al. in [5] considered a set of software metrics to predict vulnerable files. As our goal is to compare method-level metrics with nano-patterns, we do not consider the entire set of metrics considered in [5]. We selected the set of metrics that are related to methods as listed in Table I. These metrics capture the properties of methods including their structural complexity, dependency on other methods, parameters, return conditions, and so on. Although earlier studies are reasonably accurate at assessing software quality, they did not work at lower granularity level of the vulnerable files. Identifying security issues at a lower granular level assists practitioners to precisely identify vulnerable code. As a file may consist of a set of classes or methods, it is more difficult to pinpoint vulnerable code in a file flagged as vulnerable using existing prediction techniques. According to Giger et al. [17], a developer needs a significant amount of time to examine all methods of a file in order to locate a particular bug if a file is significantly large. In addition, prediction at the level of source files exhibits lower accuracy [18]. Therefore, mechanisms to classify vulnerable code at varying levels of granularity are needed. This work is the first study on comparative analysis between nano-patterns and software metrics in vulnerability prediction.

Our motivation is to find out a better way of predicting vulnerabilities with fine-grained analysis on methods, and observe if that works better than software metrics. Developers can be more conscious about vulnerable code areas detected by the model. Testers will focus more effort on these vulnerable areas executing specific tests that target the identified vulnerabilities. Our empirical model targets the vulnerabilities reported and fixed in the systems under analysis. The major contributions of this paper are as follows:

- We developed a vulnerability prediction model using the nano-patterns extracted from vulnerable and neutral (we use the term "neutral" to refer to methods where no known vulnerability exists) code of different software systems. We executed three machine-learning techniques to classify vulnerable code. This study will help to understand how different nano-patterns relate to code prone to vulnerability.
- We present a comparison of performance measures including precision and recall for two types of features (nano-patterns vs software metrics) in vulnerability prediction. This analysis will assist developers to determine features to use in a customized prediction model based on their performance. They will be able to decide which type of metrics will better assess their code.
- The results of this study will help developers to focus on code at a lower granularity level. They can target the functions that are more vulnerability-prone than others.

TABLE I: Method-level Software Metrics

Metrics	**Description**
Count Input [19]	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 called by + Parameters read + Global Variables read.
Count Output [19]	The number of outputs that are SET. This can be parameters or global variables. So Functions calls + Parameters set/modify + Global Variables set/modify.
LOC [20]	The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple metrics.
Count Path [20]	It is the number of unique paths though a body of code, not counting abnormal exits or gotos.
Cyclomatic complexity	McCabes cyclomatic complexity counts the number of independent paths through a program unit (i.e., number of decision statements plus one).
Modified cyclomatic [21]	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.
Strict cyclomatic [21]	The Cyclomatic Complexity with logical conjunction and logical and in conditional expressions also adding 1 to the complexity for each of their occurrences.
Essential [21]	It is the cyclomatic complexity after iteratively replacing all well structured control structures (if-then-else and while loops) with a single statement.
Max nesting [22]	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.

The paper is organized as follows: Section II describes the related work. Section III presents the methodology of our proposed technique. Section IV analyzes the findings from our experiments. We discuss the results in Section V. Section VI identifies limitations of our work, and Section VII concludes the paper.

TABLE II: Catalog of Fundamental Nano-Patterns [16]

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	`objectCreator` — 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

II. Related Work

This section presents relevant research on software security, code patterns and metrics for defect and vulnerability prediction.

A. Traceable patterns

Gil *et* al. defined a code-level software pattern as traceable based on the attributes, types, name and body of a software module and its components [15]. They described 27 micro patterns on Java classes and interfaces [15]. Destefanis *et* al. first identified the correlations between defects and micro patterns [23], [24]. They also showed that the classes with no micro patterns are more fault-prone than the classes having micro patterns. In [25], the authors analyzed the correlation of micro patterns with vulnerable classes and identified certain micro patterns to be more prone to vulnerability than others.

They also found that some micro patterns are highly associated to each other when grouped in vulnerable classes than they are in other classes [25]. Kim *et* al. identified how micro patterns change with the program's evolution throughout the development process [26]. They analyzed the ways micro patterns evolve from one version to another and showed that some types of evolution are more bug-prone than others.

Nano-patterns represent coding constructs used at the method-level in Java software development [27]. Table II presents 17 fundamental nano-patterns [16]. Singer *et* al. later incorporated additional patterns by combining a number of fundamental patterns and used data mining concepts: frequent itemset generation and association rule mining to classify Java methods based on nano-pattern associations [16]. Sultana et al. used a similar approach for analyzing nano-patterns and exploring their relationship with vulnerabilities [13], [14]. Deo *et* al. also found certain nano-patterns to be more fault-prone than others [28]. In this work, we found statistically significant patterns and used them in vulnerability prediction and then compared their performance with current metrics. The output of the study will help developers to classify vulnerable code and decide what areas to test more rigorously than others.

B. Software Security

Researchers employed a number of software metrics to evaluate the security of a software system. Of them, complexity metrics are the metrics most used in existing literature for software quality assessment. Researchers found that complexity is related to software defects [7]. They identified complexity metrics-based fault prediction models that are useful for vulnerability prediction and found that certain metrics such as nesting complexity metrics are more effective for locating vulnerable code than others [10]. Shin *et* al. [6], [9] conducted an empirical study to analyze the impacts of complexity metrics on vulnerable and non-vulnerable files. They showed that traditional metrics such as code churn, complexity and fault history provide similar performance in vulnerability prediction as they perform in fault prediction models [9]. In another study, the authors explored how different complexity metrics, code churn, and developer activity metrics can better predict vulnerabilities [8]. They showed that complexity metrics do not have statistically significant discriminative and reliable predictive power, but development history metrics are stronger indicators of vulnerabilities compared to code complexity metrics [29]. Chowdhury *et* al. in [5], [12] defined how different code structures cause vulnerable source code. The authors used four alternative data mining and statistical techniques to show that complexity, coupling and cohesion can be effective as vulnerability metrics in the early stage of development [5].

Alshammari *et* al. in [30] introduced seven metrics for assessing the security of an object-oriented class. Chowdhury et al. introduced three code-level metrics: stall ratio, coupling corruption propagation and critical element ratio to measure the software security [11]. 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 [11]. Zimmermann *et* al. [31] evaluated the efficacy of some conventional metrics such as complexity, churn, coverage, dependency measures and the organizational structure of a company to predict vulnerabilities using a large-scale empirical study. They showed that these metrics predict vulnerabilities with higher precision but lower recall. They also remarked that vulnerability prediction is not as simple as defect prediction as accurate vulnerability prediction depends on the program domain and usage of the program components. Younis *et* al. [32] also determined the performance of software metrics for vulnerability prediction, by considering the vulnerabilities that were exposed by attacks, to train their model [32].

The earlier works did not distinguish between class-level and method-level metrics. File-level metrics cannot identify the problem precisely as a file can be of any size. The file can be composed of any number of classes and methods. Therefore, developers cannot be certain about the impact of the potentially vulnerable features in their code and need more time to find them [17]. We aim to assess security by targeting vulnerabilities at a lower granularity level than a file. In this study, we separated method-level metrics and trained the model with those metrics. Then, we compared their prediction accuracy with the accuracy of the model trained with nano-patterns. Developers and testers can then decide to use the appropriate metrics for their prediction model.

III. Methodology

In this section, we describe our research goal, research questions, study design and the experimental methodology applied to answer the research questions.

A. Research Goal & Questions

Our research goal is *to make a comparative study between the prediction models for security vulnerabilities using traditional software metrics and nano-patterns.* In order to accomplish this goal, we developed the following research questions:

- **Research Question 1 (RQ1):** *What is the performance of the vulnerability prediction model trained using nano-patterns as features?*
  The results of the related experiments evaluate nano-patterns as predictors for security vulnerabilities. We present various performance measures including false positive rate, recall, precision, roc, and F_2-measure while using nano-patterns as features for classifying code as vulnerable.
- **Research Question 2 (RQ2):** *What is the performance of the vulnerability prediction model trained using software metrics as features?*
  Similar to RQ1, this question allows us to determine how software metrics perform using the same techniques in vulnerability prediction. We present false positive rate,

recall, precision, roc, and F_2-measure while using software metrics as features in classifying code as vulnerable.

B. Study Design

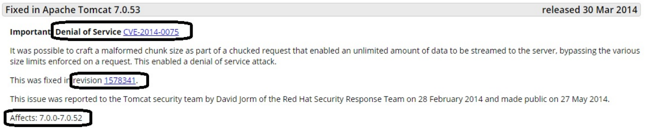
We used three different projects: Apache Tomcat (Release 6 and 7)[3], Apache CXF[4] and the Stanford Securibench dataset[5]. Apache projects were used for two major reasons. First, all vulnerability reports, including pointers to the classes affected by a vulnerability for every release, are documented as security advisories on the Apache website. Second, we needed a system written in Java as nano-patterns are currently only defined for Java methods. Apache Tomcat 6 and 7 are more recent releases and according to Hovsepyan et al., prediction models based on older versions could be less favourable compared to more recent versions [33]. We used Apache Tomcat vulnerability reports[6] and CXF vulnerability report[7] to find the list of all known vulnerabilities for them.

Stanford SecuriBench is a set of open source programs used as a test-bed for static and dynamic security tools [34], [35]. We conducted the experiment for J2EE web applications Personalblog 1.2.6 and Roller 0.9.9 in the Stanford dataset. We used a static analyzer tool called Early Security Vulnerability Detector - ESVD [36] to identify vulnerable code for the Stanford SecuriBench projects as it was shown to have fewer false positives in its results with higher precision and recall for the Stanford projects [37]. In addition, we manually checked the vulnerabilities reported by ESVD to determine if false positives existed in the dataset.

We considered the last reported version of every project as the neutral version, as the reported vulnerabilities in each project do not exist in the last version of that release. If an existing vulnerability is removed in a Java method of version 7.0.2, that method of version 7.0.2 and later versions will be termed as a "neutral" method. On the other hand, the source code of that method in the versions immediately preceding 7.0.2 will be deemed as a "vulnerable" method. At the time of study, 6.0.48 was the last version in Tomcat-6, 7.0.75 in Tomcat-7 and 3.1.10 in CXF (of the versions selected for this study). Although the last version may contain other vulnerabilities that may be reported in subsequent releases, the vulnerabilities reported in earlier versions and considered in this study have been fixed and do not exist in the final version. In the case of Stanford, we considered the methods as neutral where no vulnerabilities were detected by ESVD.

C. Vulnerability Collection

1) Apache Tomcat: We collected Apache Tomcat vulnerability reports that provide the information about the vulnerability type, its CVE (Common Vulnerabilities and Exposures) id, affected versions, revision number and fixed version in the selected versions. For example, if a vulnerability affects the versions 6.1, 6.2, 6.3 and is fixed in 6.4, we considered 6.3 as the last affected version. Thus, we collected the last affected code versions for the listed vulnerabilities as shown in Table III. We considered 14 versions having vulnerabilities in Tomcat-6 and 18 versions in Tomcat-7. The source code for Apache Tomcat is located in Apache Archives of Tomcat[8]. We found 124 vulnerable methods in Tomcat-6 and 106 vulnerable methods in Tomcat-7 shown in Table III. These methods are the total affected methods in two releases by different vulnerabilities. We considered a method multiple times if it is affected by different vulnerabilities in different versions. In figure 1, the Denial of Service (CVE-2014-0075) vulnerability was fixed in revision 1578341 of version 7.0.53. If we follow the link to its revision number[9], we obtain the list of classes modified to fix the vulnerability (figure 2).



Fixed in Apache Tomcat 7.0.53 released 30 Mar 2014

Important: Denial of Service CVE-2014-0075

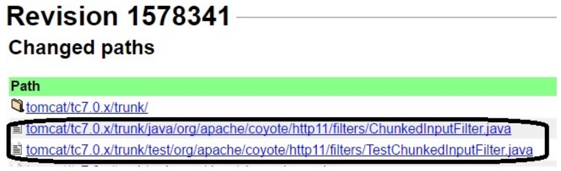
It was possible to craft a malformed chunk size as part of a chucked request that enabled an unlimited amount of data to be streamed to the server, bypassing the various size limits enforced on a request. This enabled a denial of service attack.

This was fixed in revision 1578341.

This issue was reported to the Tomcat security team by David Jorm of the Red Hat Security Response Team on 28 February 2014 and made public on 27 May 2014.

Affects: 7.0.0-7.0.52

Fig. 1: Apache Tomcat Security Page



Revision 1578341

Changed paths

Path
tomcat/tc7.0.x/trunk/
tomcat/tc7.0.x/trunk/java/org/apache/coyote/http11/filters/ChunkedInputFilter.java
tomcat/tc7.0.x/trunk/test/org/apache/coyote/http11/filters/TestChunkedInputFilter.java

Fig. 2: Affected class names for a vulnerability.

TABLE III: Systems

Systems	Affected Versions	Vulnerable methods	Neutral 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	124	8645
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	106	10296
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	45	26366
Stanford Securibench	Personalblog 1.2.6 and Roller 0.9.9	156	2734

2) Apache CXF: The vulnerability reports of Apache CXF provide information about the vulnerability type, its CVE id, affected versions, revision number and fixed version. We collected the last affected code versions for the listed vulnerabilities as shown in Table III. We considered 12 versions for Apache CXF. The reports also provide the list of classes that were modified to fix the vulnerability as shown in figure 3. We found 45 vulnerable methods in Apache CXF as shown in Table III. The source code for Apache CXF is located in Apache Archives of CXF[10].

[3] https://tomcat.apache.org/

[4] http://cxf.apache.org/

[5] https://suif.stanford.edu/ livshits/securibench/stats.html

[6] https://tomcat.apache.org/security.html

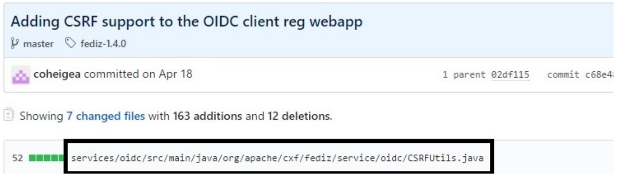
[7] http://cxf.apache.org/security-advisories.html

[8] http://archive.apache.org/dist/tomcat/

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

[10] http://archive.apache.org/dist/cxf/

3) Stanford Securibench: We used Early Security Vulnerability Detector (ESVD) to identify vulnerable classes and methods for the following projects: personalblog 1.2.6 and roller 0.9.9. We explored the vulnerable code and found 12 vulnerable methods in personalblog 1.2.6 and 144 vulnerable methods in roller 0.9.9.



Adding CSRF support to the OIDC client reg webapp

coheigea committed on Apr 18

Showing 7 changed files with 163 additions and 12 deletions.

52 services/oidc/src/main/java/org/apache/cxf/fediz/service/oidc/CSRFUtils.java

Fig. 3: Affected classes for a vulnerability in Apache CXF.

D. Nano-Patterns Extraction

The versions affected by the reported vulnerabilities of Apache Tomcat are listed in the **Affected Versions** column of Table III. We then used the nano-patterns tool as developed by Singer [16] to dump all methods and their nano-patterns for that version into our database. The tool extracts the nano-patterns inside a method and reports their presence as 1/0 (1 if the nano-pattern is present, 0 otherwise). After that, we collected the name of the methods inside every class that were revised to remove a vulnerability and stored them and their nano-patterns in a separate file from the database table. We found 124, 106, 45, 156 vulnerable methods in Tomcat-6, Tomcat-7, Apache CXF and Stanford, respectively (presented in Table III). Next, we collected nano-patterns from the source code of the neutral versions using the same tool. We obtained nano-patterns for 8,645 methods in Tomcat-6, 10,296 methods in Tomcat-7, 26,366 methods in CXF and 2,734 methods in the Stanford applications.

E. Software Metrics Extraction

We used a commercial tool called SciTools Understand 4.0[11] to compute the source code metrics. We created a separate project in Understand for every version of every project and ran a scheduler after specifying the metrics. This process takes almost 30 seconds for each project. We executed the scheduler for each project and generated csv files containing the method-level metrics as in Table I for that project. Then we recorded the vulnerable methods and stored them in a separate file.

F. Feature Selection for Prediction Model

We extracted a total of 24 nano-patterns (including 17 fundamental patterns) for every method in our codebase using the nano-patterns extraction tool [16]. Then, we computed Welch's t-test between the nano-patterns and vulnerabilities. Welch's test[12] 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 a significantly different mean in vulnerable and neutral methods using Welch's test, we assumed that pattern is distributed differently in two groups and considered it as significant for that system. By doing so, we collected the significant patterns out of 24 nano-patterns and then trained the machine using those nano-patterns as features for that system. As they showed a significantly different distribution in two groups, the nano-patterns can better distinguish the vulnerable and neutral methods, and we can ignore irrelevant patterns in the training model.

G. Vulnerability Prediction

In our problem, we identified a set of features from the program component (i.e., method). We collected the feature values from vulnerable methods and neutral methods. There are two groups of methods for each project, vulnerable and neutral. We collected labeled data (marked as vulnerable or neutral) and then trained the machine so that it can classify any method as vulnerable or neutral based on its learning algorithm. Supervised machine learning can help predict vulnerable methods based on the values of the method's features. We applied three supervised learning techniques (Naive Bayes, Support Vector Machine, and Logistic Regression) as they were used in earlier studies of vulnerability prediction [12]. 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. A Naive Bayes classifier categorizes an instance assuming that all of 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 relatively small set of training data to properly classify the data set [38]. SVM is a supervised learning model where the training points are separated by the widest possible gap. The new point is classified based on its side of the gap. SVMs can perform both the linear and non-linear classification mapping the input into higher dimensional space [39]. Logistic regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function [38]. In our study, LR calculates the probability of a method being vulnerable or neutral for the given values of the used features.

1) Tool Selection: 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[13]. We used WEKA 3.8 for our study. The parameters for each of the techniques were initialized using the default settings for WEKA.

2) Data Balance: As our dataset was not balanced, we needed to create a balanced dataset consisting of the same number of vulnerable and neutral methods. Earlier studies

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

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

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

either considered under-sampling of the majority class or over-sampling of the minority class. The problem with under-sampling is that information may be lost. In the case of over-sampling, duplicating the instances of the minority category (vulnerable methods) can make the system biased to the vulnerable methods when the number of vulnerable methods is too small. In this study, we applied the *ClassBalancer* filter in WEKA 3.8[14]. This filter re-weights the instances so that each method has the same total weight. This filter changes only the weight of the first batch of the data which is used for training the model: the testing dataset is not re-weighted by this balancer.

3) Data Analysis: We executed three machine-learning algorithms for the vulnerable and neutral methods of all the projects under study. We ran them separately for the nano-patterns and the software metrics. For each algorithm, we applied 10-fold cross-validation to ensure that the trained model will work accurately for an unknown dataset in practice [40]. The result of this experiment shows how accurately a model trained with historical data can predict vulnerable methods of later releases.

4) Performance Measures:

- **Precision:** Precision for vulnerable method prediction can be defined as the ratio of the number of predicted vulnerable methods that are actually vulnerable to the total number of methods retrieved as vulnerable [40]. This concept is also seen as the correctness of the prediction model.
- **Recall:** Recall of vulnerable method prediction is defined as the ratio of the number of predicted vulnerable methods that are actually vulnerable to the total number of vulnerable methods in the system. This measure is significant in the case of vulnerability prediction because the higher the recall is for vulnerable method prediction, fewer vulnerable methods remain undetected. However, there is a trade-off between recall and precision. For example, if all the methods predicted as vulnerable by the model are actually vulnerable, its precision will be 100 percent. On the other hand, there is still a possibility that many vulnerable methods remain as undetected or wrongly predicted as neutral. Similarly, if a model can successfully detect all the vulnerable methods, its recall will be 100 percent, but precision will degrade if it predicts many neutral methods as vulnerable as well.
- **False Negative Rate:** The FN rate indicates the percentage of vulnerable methods that are wrongly predicted as neutral. It is significant in our case as predicting a vulnerable method as neutral may cause serious security failures compared to the wrongly predicted neutral methods.
- **False Positive Rate:** The FP rate indicates the percentage of neutral methods that are wrongly predicted as vulnerable. A high FP rate makes a predictor useless.
- **F-measure:** F-measure is a weighted average of precision and recall [40]. It gives equal importance to both precision and recall by considering their harmonic mean [5], [40]. The formula for F-measure is defined as follows [41]:

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

  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. $\beta > 1$ gives more weight to recall 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 the ROC curve is a measure that evaluates the performance of the binary classifier in terms of TP and FP rate. An area close to 1 indicates a high-performance model and an area about 0.5 indicates low-performance model [42]. An increase in Recall (TP Rate) also results in an increase in FP Rate. Therefore, the ROC measure helps to track the optimum point where the metric performs well with respect to the TP and FP rate.

IV. Results

The list of nano-patterns that we found as significant for the systems is presented in Table IV. Some of the nano-patterns are significant for all the systems under study (e.g. exceptions). For machine learning, we trained the machine separately for each system as the significant patterns are not same for all of them (Table IV).

TABLE IV: Results of Welch's Test

Nano-pattern	Tomcat-6	Tomcat-7	Apache CXF	Stanford Securibench
noparams	-0.183		-0.416	-0.504
samename	0.333	0.291		
leaf	-0.633	-0.483		
objCreator	0.801	0.826	0.613	1.527
thisInstanceFieldReader	0.57	0.447		
thisInstanceFieldWriter	0.43	0.515		
staticFieldReader	0.846	0.530	1.016	1.745
typeManipulator	0.539	0.523	1.631	1.181
straightLine	-0.824	-0.601	-0.968	-1.042
looper	0.711	0.736	1.11	
exceptions	0.633	0.76	0.430	1.183
localReader		0.156		
localWriter	0.829	0.645	1.238	
arrReader	0.218			
arrWriter	-0.17			
jdkClient	0.736	0.109	0.994	1.092
tailCaller	0.420	0.260		0.461
otherInstanceFieldReader		-0.175		
otherInstanceFieldWriter		0.274		
void				-0.842

*Effect sizes are mentioned only for the significant correlations (i.e., $p < .05$).

RQ1: What is the performance of the vulnerability prediction model trained using nano-patterns as features?

The FN rate, FP rate, Precision, Recall, F_2-Measure, and ROC of nano-patterns based prediction model for Tomcat-6, Tomcat-7, Stanford Securibench, and Apache CXF have been presented in Table V, Table VI, Table VII, and Table VIII, respectively. The tables present results of three different machine

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

learning techniques: Naive Bayes, Support Vector Machine and Logistic regression. We see that using nano-patterns as features in Naive Bayes, the recall rate is more than 80 percent for every project except Tomcat-7, precision is more than 65 percent, FP rate is less than 33 percent, and F_2-Measure is greater than 75 percent for all except Tomcat-7. For the SVM technique, recall rate is more than 70 percent for all projects, precision is more than 73 percent, FP rate is less than 25 percent, and F_2-Measure is greater than 70 percent. In LR, recall rate is more than 71 percent for all projects, precision is more than 74 percent, FP rate is less than 24 percent, and F_2-Measure is greater than 71 percent.

Finding: *Using nano-patterns results in recall greater than 70 percent, precision greater than 65 percent, F_2-measure greater than 70 percent, and FP rate less than 33 percent.*

RQ2: What is the performance of the vulnerability prediction model trained using software metrics as features?

The FN rate, FP rate, Precision, Recall, F_2-Measure, and ROC of software metrics based prediction model for Tomcat-6, Tomcat-7, Stanford Securibench, and Apache CXF have been presented in Table V, Table VI, Table VII, and Table VIII, respectively. The tables present results of three different machine learning techniques: Naive Bayes (NB), Support Vector Machine (SVM) and Logistic regression (LR). We see that using metrics as features in Naive Bayes, the recall rate is not more than 69 percent, precision is more than 83 percent, FP rate is less than 12 percent, and F_2-Measure is not greater than 63 percent for all projects (except Apache CXF). For SVM, recall rate is more than 67 percent for all projects (except Stanford Securibench), precision is more than 78 percent, FP rate is less than 21 percent, and F_2-Measure is not greater than 68 percent (except Stanford Securibench). In LR, recall rate is not more than 71 percent for all projects (except Stanford Securibench), precision is more than 79 percent, FP rate is less than 16.9 percent, and F_2-Measure is not greater than 67.7 percent (except for Apache CXF).

Finding: *Method-level metrics have recall less than 71 percent for all projects (except Stanford Securibench), precision greater than 77.5 percent, F_2-measure greater than 88 percent, and FP rate less than 21.3 percent.*

V. Discussion

A. Prediction Performance of two models

1) False Negative rate and Recall: The False Negative rate for vulnerable methods using nano-patterns is lower than method-level metrics for all systems under study. This result shows that the nano-pattern 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. An analysis comparing the FN rate and Recall between nano-patterns and method-level metrics using LR is presented in figure 4a and figure 4b, respectively.

The maximum FN rate is 30 percent (SVM) in nano-patterns, which is 36 percent (SVM) in the method-level metrics based model. The lowest Recall is 69 percent (Naive Bayes) for nano-patterns, which is 50 percent (Naive Bayes) for method-level metrics. Sultana et al. in [14] found that certain nano-patterns are frequent in vulnerable methods compared to their presence in non-vulnerable methods. One observation based on the vulnerable patterns that were examined in the prior work confirms that methods which read values from objects must be checked for proper sanitization in order to keep the code secure (certain vulnerable patterns found in this case deal with user / system input). They statistically determined the significant difference in the presence of nano-patterns in vulnerable and neutral methods. Therefore, the performance of nano-patterns in predicting more vulnerable methods in this study is logical given the findings of the previous work.

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 developers and testers. After analyzing the results, we see that nano-patterns perform better than metrics in retrieving more vulnerable code. Shin et al. in [6] empirically showed that complexity metrics can predict vulnerabilities at a low FP rate but at a high FN rate. This result highlighted that the vulnerable files 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 [6].

2) Precision: Precision indicates the percentage of actual vulnerable code (method or class) in the total vulnerable code predicted as vulnerable by the model. An analysis comparing precision between nano-patterns and method-level metrics using LR is presented in figure 5a. The figure shows that method-level metrics outperform nano-patterns. The maximum precision is 83.7 percent (LR) in nano-patterns, which is 89 percent(NB) in method-level metric based model. 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.

The results show that percentage of actual vulnerable code in the total predicted vulnerable code is higher in the metric based prediction model. In other words, using metrics as features in vulnerability prediction can be more precise than using nano-patterns.

3) 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 desired. Alternatively, if neutral code is detected

TABLE V: Results for the Tomcat-6

	Method	FN	FP	Precision	Recall	F_2-Measure	ROC		Method	FN	FP	Precision	Recall	F_2-Measure	ROC
Metrics	NB	0.500	0.090	0.847	0.500	0.545	0.845	Nano-patterns	NB	0.177	0.321	0.720	0.823	0..800	0.831
	LR	0.347	0.169	0.794	0.653	0.677	0.852		LR	0.210	0.241	0.766	0.790	0.785	0.840
	SVM	0.331	0.195	0.775	0.669	0.688	0.737		SVM	0.258	0.231	0.763	0.742	0.746	0.756

TABLE VI: Results for the Tomcat-7

	Method	FN Rate	FP Rate	Precision	Recall	F_2-Measure	ROC		Method	FN	FP	Precision	Recall	F_2-Measure	ROC
Metrics	NB	0.475	0.088	0.856	0.525	0.569	0.812	Nano-patterns	NB	0.311	0.329	0.676	0.689	0.686	0.768
	LR	0.366	0.159	0.800	0.634	0.661	0.811		LR	0.292	0.243	0.744	0.708	0.715	0.784
	SVM	0.356	0.167	0.794	0.644	0.669	0.738		SVM	0.302	0.258	0.730	0.698	0.704	0.720

TABLE VII: Results for the Stanford Securibench

	Method	FN	FP	Precision	Recall	F_2-Measure	ROC		Method	FN	FP	Precision	Recall	F_2-Measure	ROC
Metrics	NB	0.408	0.120	0.832	0.592	0.628	0.858	Nano-patterns	NB	0.173	0.191	0.812	0.827	0.824	0.901
	LR	0.224	0.159	0.830	0.776	0.628	0.896		LR	0.135	0.168	0.837	0.865	0.824	0.915
	SVM	0.099	0.213	0.809	0.901	0.881	0.844		SVM	0.103	0.180	0.833	0.897	0.883	0.859

TABLE VIII: Results for the Apache CXF

	Method	FN	FP	Precision	Recall	F_2-Measure	ROC		Method	FN	FP	Precision	Recall	F_2-Measure	ROC
Metrics	NB	0.311	0.085	0.891	0.689	0.722	0.890	Nano-patterns	NB	0.178	0.244	0.771	0.822	0.771	0.876
	LR	0.289	0.110	0.866	0.711	0.786	0.865		LR	0.156	0.216	0.796	0.844	0.859	0.869
	SVM	0.356	0.095	0.871	0.644	0.679	0.775		SVM	0.200	0.215	0.788	0.800	0.798	0.792

TABLE IX: Prediction performance of different techniques

Techniques	**FP Rate**	**Precision**	**Recall**	**F_2-measure**
Naive Bayes (NB)	yes	yes	yes	yes
Logistic Regression (LR)	yes	no	yes	yes
Support Vector Machine (SVM)	no	no	no	no

as vulnerable, precision will be lower. This case may increase the workload of testers, but it is not as harmful. According to the figure 5b, F_2-measure is higher in the nano-patterns based model compared to the metrics-based model.

B. Prediction Performance of techniques

We considered three machine learning techniques (Naive Bayes, Logistic Regression, and Support Vector Machine). We performed a statistical significance test to determine whether the differences in performance measures by different techniques are statistically significant. Traditionally, a significance level of less than or equal to 0.05 (the 95% confidence level) is considered statistically significant. Therefore, we performed the standard t-test at 0.05 significance. Table IX reports the results of the significance test. "Yes" indicates that the difference of the measure in two models is statistically significant at 0.5 significance level. "No" means the specific performance measure of the two models might be different, but the difference is statistically insignificant. We found that the differences in NB and LR are statistically significant, but in SVM, the differences are not statistically significant.

C. Trade-off between Recall and FP Rate

We expected 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 7 presents the ROC curves using nano-patterns and method-level metrics in LR. According to figure 7, nano-patterns can correctly identify about 50 percent of the vulnerable methods while keeping the FP rate below 10 percent and about 75 percent of the vulnerable methods when the FP rate is below 20 percent. In the case of method-level metrics, 75 percent of the vulnerable methods are detected at the FP rate greater than 25 percent as shown in figure 7.

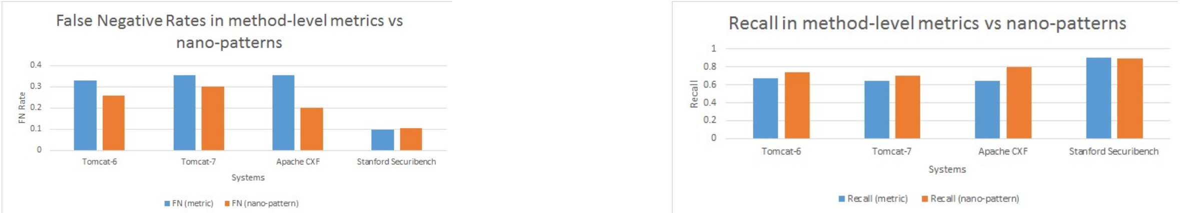
D. Comparison of results with other works

Chowdhury et al. in [5] showed that complexity, coupling, and cohesion can be effective metrics at vulnerability prediction in the early stages of software development. They did not distinguish between class-level and method-level software metrics in their study. The study was performed on different releases of Mozilla Firefox and used the source files (files with .c, .cpp, .java, and .h extensions) to compute the metrics. In our case, we did the study for Java-based systems and computed only method-level metrics. We compared the average FP rate and recall of their study with those of our study using Logistic Regression in figure 6. Method-level metrics based prediction results in a lower FP rate and nearly identical (65 percent vs. 69 percent) recall rate compared to the study in [5]. The nano-patterns based model achieves higher recall and lower FP rates using LR compared to the study by Chowdhury et al. in [5].

VI. Threats to Validity

In this section, we discuss the shortcomings of our study.

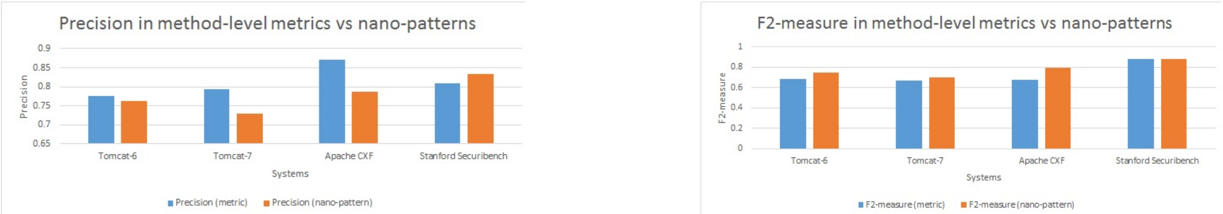
Construct Validity: In this study, we compared nano-patterns with software metrics as vulnerability predictors. Although nano-patterns represent properties of a Java method, they may not capture all types of method characteristics. In addition, we collected the vulnerable methods from the Apache vulnerability reports and the results of a static analysis tool. These reports and tools may miss vulnerabilities (e.g., undiscovered vulnerabilities) that may still exist in the neutral version.



(a) False Negative of nano-patterns and method-level metrics

(b) Recall of nano-patterns and method-level metrics

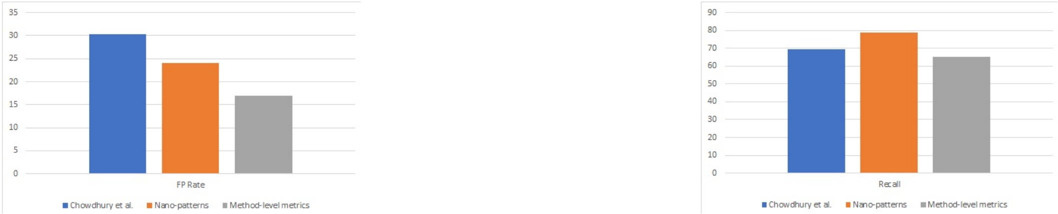
Fig. 4: Performance measures of nano-patterns and method-level metrics (LR)



(a) Precision of nano-patterns and method-level metrics

(b) F_2-measure of nano-patterns and method-level metrics

Fig. 5: Performance measures of nano-patterns and method-level metrics (LR)



(a) Comparison of False Positive rates with other studies

(b) Comparison of Recall with other studies

Fig. 6: Comparison of Performance measures with other studies

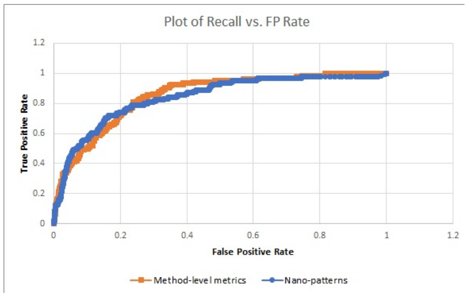


Fig. 7: ROC for nano-patterns and method-level metrics

External Validity: External Validity refers to the ability to generalize results. The experiment was conducted on two Apache projects and two J2EE web applications with known vulnerabilities. Although method-level metrics can be extracted from applications written in other languages, nano-patterns are currently defined only for Java methods. Therefore, it cannot be concluded that the results are valid for systems written in other programming languages.

Internal Validity: This threat refers to the possibility of having confounded or unanticipated relationships among variables. We are not claiming causation. We do not claim that these patterns and metrics are the direct cause of the vulnerabilities found in the methods, but we recommend rigorous testing of the methods which are predicted vulnerable by the model. It can be assumed that these methods contain properties that are related to vulnerable code.

VII. Conclusion

In this study, we analyzed how well nano-patterns and method-level metrics predict software vulnerabilities. This is the first study comparing the performance of these patterns in vulnerability prediction with traditional metrics. We found that the nano-patterns based prediction model better predicts vulnerable methods than the metrics based model as nano-patterns showed higher recall than metrics in vulnerability prediction. If any method is predicted as vulnerable in a nano-patterns based prediction model, developers can be more cautious using that method and suggest more rigorous testing. On the other hand, metrics have higher precision than patterns. This study will allow developers and testers to compare these two types of features and decide the best one for their project based on their characteristics. It will ensure software security by giving them the ability to target tests to potentially vulnerable files and allowing early detection of risks. In the future, we will extend the concept of nano-patterns by investigating the relationship of the nano-patterns with vulnerabilities and look to create new patterns that specifically address security related features. This research will open up new areas to investigate the metrics at different granularity levels so that they can be used more precisely in ensuring software security.

References

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

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

[3] R. Seacord, *Secure Coding in C and C++*, 1st ed. Addison-Wesley Professional, 2005.

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

[5] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, mar 2011.

[6] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. NY, USA: ACM, 2008, pp. 315–317.

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

[8] 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, pp. 772–787, nov 2011.

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

[10] ——, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM Workshop on Quality of Protection*, ser. QoP '08. NY, USA: ACM, 2008, pp. 47–50.

[11] I. Chowdhury, B. Chan, and M. Z. Chowdhury, "Security metrics for source code structures," in *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, ser. SESS '08. NY, USA: ACM, 2008, pp. 57–64.

[12] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. NY, USA: ACM, 2010, pp. 1963–1969.

[13] K. Z. Sultana, A. Deo, and B. J. Williams, "A preliminary study examining relationships between nano-patterns and software security vulnerabilities," in *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[14] ——, "Correlation analysis among java nano-patterns and software vulnerabilities," in *IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, 2017.

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

[16] 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, pp. 191 – 204, 2010, proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

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

[18] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. NY, USA: ACM, 2015, pp. 4:1–4:9.

[19] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, Sep. 1981.

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

[21] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[22] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," *SIGPLAN Not.*, vol. 16, no. 3, pp. 63–74, Mar. 1981.

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

[24] G. Destefanis, "Assessing software quality by micro patterns detection," Ph.D. dissertation, 2012.

[25] B. W. K.Z. Sultana and T. Bhowmik, "A study examining relationships between micro patterns and security vulnerabilities," *Software Quality Journal*, 2017.

[26] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Micro pattern evolution," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. NY, USA: ACM, 2006, pp. 40–46.

[27] F. Batarseh, "Java nano patterns: A set of reusable objects," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE '10. New York, NY, USA: ACM, 2010, pp. 60:1–60:4.

[28] A. Deo and B. J. Williams, "Preliminary study on assessing software defects using nano-pattern detection," in *Proceedings of the 24th International Conference on Software Engineering and Data Engineering (SEDE)*, 2015.

[29] 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, pp. 56:1–56:36, Aug. 2017.

[30] D. C. Bandar Alshammari, Colin Fidge, "Security metrics for object-oriented class designs," in *Proceedings of the 9th International Conference on Quality Software*, 2009, pp. 11–20.

[31] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. DC, USA: IEEE Computer Society, 2010, pp. 421–428.

[32] 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," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. NY, USA: ACM, 2016, pp. 97–104.

[33] A. Hovsepyan, R. Scandariato, and W. Joosen, "Is newer always better?: The case of vulnerability prediction models," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. NY, USA: ACM, 2016, pp. 26:1–26:6.

[34] V. B. Livshits, "Findings security errors in Java applications using lightweight static analysis," Work-in-Progress Report, Annual Computer Security Applications Conference, nov 2004.

[35] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *Proceedings of the 14th Usenix Security Symposium*, aug 2005, pp. 271–286.

[36] L. Sampaio. (2017-07-20) Early security vulnerability detector - esvd. https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/. Accessed: 2017-08-04.

[37] C. Alonso. (2015) Early vulnerability detection for supporting secure programming. http://docplayer.net/1619013-Early-vulnerability-detection-for-supporting-secure-programming.html. Accessed: 2017-08-04.

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

[39] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*, 2000.

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

[41] N. Chinchor, "Muc-4 evaluation metrics," in *Proceedings of the 4th Conference on Message Understanding*, ser. MUC4 '92. Stroudsburg, PA, USA: Association for Computational Linguistics, 1992, pp. 22–29.

[42] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. NY, USA: ACM, 2016, pp. 1415–1421.