A Preliminary Study Examining Relationships Between Nano-Patterns and Software Security Vulnerabilities

Kazi Zakia Sultana*, Ajay Deo* and Byron J. Williams*
**Department of Computer Science and Engineering*
Mississippi State University, Starkville, Mississippi, USA

***Abstract*—Software security plays a significant role in ensuring software quality. The goal of this study is to conduct a preliminary analysis to find hidden relationships between source code patterns and security defects. We describe a study in which we focus on evaluating software security using nano-patterns to reduce security risks during the development lifecycle. Nano-patterns are simple properties of Java methods. In our research, we investigate the correlation between software vulnerabilities and nano-patterns using data mining techniques. Identifying these relationships can assist developers to quickly assess the likelihood that they are writing vulnerable code and recommend tests to uncover the vulnerability. The goal of this research is to reduce the amount of vulnerable code developers write. We successfully apply data mining techniques to identify vulnerable code characteristics and apply hypothesis testing to validate the findings. This preliminary study shows that certain nano-patterns $localReader$, $jdkClient$, $tailCaller$ are significantly present in vulnerable methods. These findings can be used to recommend security test patterns to improve vulnerability testing and reduce the number of vulnerabilities in released code.**

I. Introduction

Today's software systems need to ensure the safeguarding of sensitive and confidential information while providing quality service to its users. Although security in software has been getting more attention recently, prediction models for possible attacks and detection mechanisms for security violations have yet to be explored in detail. The primary emphasis of this study is finding hidden relationships between source code and software vulnerabilities. As these relationships are identified, relevant code where there exists a high likelihood for vulnerabilities can be flagged for more rigorous or targeted testing. This study uses traceable, method-level software patterns easily identified in the source code to find associations between pattern and certain types of known vulnerabilities.

Gil et al. [3] developed the concept of traceable patterns. Unlike design patterns, traceable patterns can be automatically (mechanically) recognized. They are based on a specific programming language and are derived from a single software element. Traceable patterns are of various types depending upon their level of abstraction. Class-level traceable patterns are called micro-patterns. Method-level traceable patterns are called nano-patterns. Gil et al. [3] defined 27 micro-patterns organized into 8 categories which are generally defined 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. Singer et al. [2] listed 17 fundamental nano-patterns organized into 4 groups. Micro-patterns can be used to capture program characteristics to identify classes that are fault-prone [9]. Similarly, Deo et al. in paper [8] found that some nano-patterns such as $localReader$ has high presence (95.97%) in defective methods. As the nano-patterns have been found to have some relations with software defects as in [8], we chose this type of pattern for vulnerability prediction. The nano-patterns are method-level patterns, and after analyzing the patch files, we saw that most of the bug fixing patterns are at method or statement level.

There have been efforts at using code-level analysis to determine the likelihood of security vulnerabilities [10], [11]. These studies use metrics to predict security vulnerabilities. Although the technique resulted in highly accurate vulnerability prediction, the methods suffered from high false negative rates. This work attempts to improve on existing code-level analysis studies by adding a technique that uses a complimentary approach that could be combined to reduce the false negative rates.

This study focuses on identifying correlations between nano-patterns (method-level traceable constructs) and code vulnerabilities. To identify the correlations, we use statistical analysis and association rule mining. These techniques explore the associations between known vulnerabilities and nano-patterns. In addition, this analysis finds the relationship between the associations among the nano-patterns and vulnerable code. Association analysis is a well-known concept in data mining for discovering hidden relationships among large data-sets [4]. The data-set examined is the set of nano-patterns contained in the vulnerable methods (methods within the vulnerable code snippets). This analysis will be the foundation for building a model to categorize different vulnerabilities and using that model to facilitate vulnerability prediction and testing. Although our empirical model is limited to discovering correlation between patterns and vulnerabilities, it can be extended for any types of software defects. This study uses the following steps for its analysis:

- We collect a set of known vulnerabilities found in previous versions of Apache Tomcat[1]. We then extract nano-patterns for every method modified to fix a vulnerability. The nano-patterns of those methods in the versions that fixed the vulnerability are also generated to find any significant differences between vulnerable and non-vulnerable methods.
- We analyze the distribution of nano-patterns in vulnerable methods and non-vulnerable methods. We use this analysis to find the nano-patterns prone to vulnerability.
- The association rules among the nano-patterns are generated for the vulnerable and non-vulnerable 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.

These findings provide additional indicators to vulnerable code areas and are preliminary to predicting the location of vulnerabilities in the source code. Software testers will also be able to use the knowledge for selecting test cases that target the vulnerability.

II. Motivation

This study is motivated by the need to improve the security and quality of software code. This premise is based on finding potentially vulnerable areas using methods not previously examined. Positive results would allow for a more focused approach to vulnerability testing where the code with the highest potential for vulnerability is examined first. As developers are made knowledgeable of vulnerable code constructs, they will be conscious of their impacts when writing code. 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. This testing will ensure both developers and testers that the target code is not vulnerable. Although different metrics can assess software quality accurately, there is still a need for mechanism to detect vulnerable code at the lower levels of granularity and provide a mechanically traceable way to do so.

III. Related Work

In this study, we examine the relationship between nano-patterns and vulnerabilities. Nano-patterns represent coding actions frequently used in Java software development [1]. Singer et al. [2] supplemented the fundamental nano-patterns by incorporating additional patterns and classified Java methods using data mining concepts: frequent itemset generation and association rule mining. They applied their work to clustering and categorizing Java methods based on the associated nano-patterns. We use a similar approach for our analysis on nano-patterns to explore their relations with vulnerabilities. Kim et al. [7] explored bug-prone micro-patterns and found these patterns sparsely changed over time. In another study, Singer et al. [6] extracted the association between micro-patterns and class's name suffix which might be useful for run-time bug detection by the developers. Destefanis et al. [9] identified micro-patterns that were more error-prone than others while extracting correlations between the patterns.

Much of the previous traceable pattern work focused on finding relationships between the patterns and defects and relied predominantly on micro-patterns. Our approach takes these strategies further in an effort to correlate patterns with software vulnerabilities using a more granular scale. The previous studies on micro- and nano-patterns explored their relations with bugs or faults but did not analyze the collective behavior of the patterns generating those bugs.

Using software security metrics to assess the security of a software system has been explored by numerous studies. These metrics are used to detect vulnerabilities in code and thus measure "how secure" the software is. Most of the works concentrated on complexity metrics for software quality assessment as researchers found that complexity is related to software faults [12]. Problem complexity, algorithmic complexity, cognitive complexity, and structural complexity are four aspects of software complexity that can impact security [12]. Structural complexity metrics may be either in code-level or design level. All these studies focused on developing code-based or design-based metrics that quantify security. This work builds on these existing approaches to develop effective security measures to assess software security.

IV. Background

This section presents a sample application of association rule mining, the technique used to generate rules between nano-patterns in vulnerable code areas.

A. Binary Representation

Market Basket Transactions relate 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.

B. Itemset and Support Count

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

[1]"Apache Tomcat" https://tomcat.apache.org

is purchased in each transaction. We will 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 [5], [4]. An important property of an itemset is support count which represents the number of transactions containing a specific itemset [5]. 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. The example of Itemset and Support Count has been presented in Section V-B1.

C. Association Rule Mining

An association rule is an expression of the form

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

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 [5]. They can be mathematically represented as follows:

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

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 [5]. 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.

1) 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 [5].

2) 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 [5]. 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 $X \to I - X$ satisfies the confidence threshold. For example, $I = \{M_1, M_2, M_3\}$ is a frequent itemset. The relevant candidate rules are $\{M_1, M_2\} \to \{M_3\}, \{M_2, M_3\} \to \{M_1\}, \{M_1, M_3\} \to \{M_2\}, \{M_1\} \to \{M_2, M_3\}, \{M_2\} \to \{M_1, M_3\}, \{M_3\} \to \{M_1, M_2\}$. In this example, if $\{M_1, M_2\} \to \{M_3\}$ rule is a low confidence rule, all other rules containing M_3 in their consequent part such as $\{M_1\} \to \{M_2, M_3\}, \{M_2\} \to \{M_1, M_3\}$ will be considered low confidence rules and will be ignored.

Table I
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

V. METHODOLOGY

In this section, we will discuss the problem statement, how we have formulated our problem in the context of association rule mining and the experimental steps we followed to do the analysis.

A. Problem Statement

The research goal is to determine the relationship between nano-patterns and vulnerable code. We developed the following research questions:

- Is there a relationship between code vulnerabilities and nano-patterns?
- Are the association rules useful in identifying vulnerabilities?

These questions provide the necessary input to vulnerability model using the relationship between different nano-patterns, vulnerabilities and their associated rules. The outcome will provide a method to analyze newly developed code using the generated rules to determine if the written code conforms to existing rules and their associated vulnerabilities (relationship born out of the historical analysis). This model will help predict the existence of possible vulnerabilities in any code which can be further verified through code inspection and testing.

B. Problem Formulation

Historical nano-pattern and vulnerability data is used to build the association rules and frequent itemsets as described in Section IV. The following subsections highlight the rule building and itemset generation process for the data used in this study.

1) Binary Representation: The vulnerable method can be represented as a series of 1's and 0's as shown in Table I. 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 I shows that the support count of p_1, p_2 is 2 as they exist together in s_1 and s_4.

2) Association Rule Mining: In our problem, an association rule satisfying a pre-defined threshold of the support count as described in Section IV-C will be generated using a data mining tool. 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. In other words, he or she could run specific tests targeting vulnerabilities of the associated type when writing a method with pattern p_3.

C. Experimental Procedure

In this section, we discuss our procedures of the analysis and elaborate our discovery of correlations between software vulnerabilities and nano-patterns.

1) Nano-Patterns Detection: The first step is to collect a list of nano-patterns contained in the vulnerable methods. Our first research question is *Is there a relationship between code vulnerabilities and nano-patterns?* To answer this question, we used a tool to extract all nano-patterns from a software system (discussed later) that contains known vulnerabilities. Singer et al. in [2] developed a tool to detect nano-patterns in Java byte code (which we modified slightly to use in this workflow). This tool provides the list of all methods and their associated nano-patterns from a given class as input. We then employed another tool developed by our research group, JiraExtractor, which downloads patch files for any revision number of a given vulnerability [8]. The JiraExtractor tool locates all methods in a specified software version that have been modified to fix a particular vulnerability. The tool then extracts the nano-patterns from the output provided by the nano-pattern tool developed by Singer et al. [2]. For our analysis, we found the list of vulnerabilities in some versions of Apache Tomcat and then used the nano-patterns tool to identify all methods and their nano-patterns. After building a list of all nano-patterns in the chosen software versions and all nano-patterns contained in vulnerable methods, we created a table similar to the one shown in Table I. We followed the same procedure to extract the nano-patterns of the methods after fixing the vulnerabilities. The data stored for vulnerable and non-vulnerable methods will fulfill the goal of finding the correlations between the vulnerable code and nano-patterns.

2) Association Rule Generation: We address our second research question; *Are the association rules useful in identifying vulnerabilities?*, by finding all association rules above a specified threshold for the relevant nano patterns. In section IV, we presented data mining concepts related to frequent itemsets and association rules generation. In this step, we generate those rules by analyzing the collected nano-pattern and vulnerability data. We use WEKA [2], a data mining tool, to generate the association rules and classifications. WEKA uses the Apriori Algorithm for finding frequent itemsets and rules efficiently. We set the values for minimum Support and minimum confidence. Support is an important measure for our analysis as it reveals the set of patterns that exist together in a number of samples. The higher the value of support for a set of patterns, the higher the probability there is a relationship between the nano-patterns and the vulnerability. On the other hand, confidence indicates the probability of a pattern existing in a sample with another set of patterns. We assumed support count to be 50% because we were interested in considering the nano-patterns which are present together in at least 50% of the total dataset. We varied the confidence level from 90% to 100% and found that the result does not significantly vary.

VI. Results

We downloaded security vulnerabilities found in released versions of Apache Tomcat 6.0 available at *https://tomcat.apache.org/security-6.html* and 7.0 available at *https://tomcat.apache.crg/security-7.html*. There were 28 nano-patterns as described in [2] included in our dataset. Therefore, the number of rules should be 2^{28} which is around 300 million. Using WEKA, we generated 71000 rules with the minimum support of 50% and minimum confidence of 90%. Out of 71000 rules, there were 52116 rules with a 100% confidence level and 2060 rules with 95% confidence level.
Table II presents the number of affected classes and methods for the vulnerabilities included in this analysis for Apache Tomcat 6.0 and Apache Tomcat 7.0.

Table II
Experimental Dataset

Vulnerability	Affected Versions	Affected Classes	Affected Methods
Denial of Service	6.0.0-6.0.33	6	20
Information Disclosure	6.0.0-6.0.39	1	1
Security Manager bypass	7.0.0-7.0.39	2	2
Denial of Service	7.0.0-7.0.22	7	19
Security Constraint Bypass	7.0.0-7.0.10	2	2

A. Research Question 1

Is there a relationship between code vulnerabilities and nano-patterns?
Significant observations related to Research Question 1 are as follows:

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

- All the methods affected by the vulnerability contain the patterns $localReader$, $jdkClient$, $tailCaller$.
- $noparams$ pattern exists in 50% of the non vulnerable methods but it is totally absent in affected methods.
- $recursive$, $leaf$ and $samename$ patterns are completely absent in vulnerable methods.
- 70% of the non-vulnerable methods have $straigtline$ patterns whereas vulnerable methods do not have this pattern.

Based on these observations, we find that there are nano-patterns which are more susceptible to vulnerability than others due to their excessive presence in vulnerable methods compared to non-vulnerable methods. The statistics show that the identified nano-patterns behave the same for all the vulnerabilities studied.

We conduct a chi-square test to prove one of our above findings. We formulate a Hypothesis $H01$ as *Vulnerability generation is independent of the StraightLine nano-pattern* and assume $\alpha = 0.05$. The test statistics formula is as follows:

$$\chi^2 = \sum \frac{(O-E)^2}{E} \tag{1}$$

We compute the observed and expected frequency for the pattern in vulnerable and non-vulnerable methods. After analyzing Tomcat version 7.0.59 (considered as non-vulnerable version), we see that there are 19017 methods where the pattern exists. The remaining 12274 methods are free of this pattern. The computations have been organized in two-ways as shown in Table III. The value in each cell is the observed frequency and the value in parentheses is the expected frequency which is computed using the formula $(RowTotal * ColumnTotal)/N$.

Table III
CONTINGENCY TABLE FOR STRAIGHTLINE PATTERN-VERSION 7

Category	Present	Absent	Total
Vulnerable	6 (13.97)	17 (9.03)	23
Non-Vulnerable	19017 (19009.03)	12274 (12281.97)	31291
Total	19023	12291	31314

According to Equation 1, the value of χ^2 is 14.11 for Table III. We did the same test for Tomcat version 6.0.41 (considered as non-vulnerable version in Tomcat 6) and found 11.6 as the value of χ^2. In this case, the degrees of freedom is 1 (degrees of freedom = $(Row-1)*(Column-1)$). For degrees of freedom $= 1$ and at 5% significance level, the appropriate critical value is 3.84 and the decision rule is: *Reject $H01$ if $\chi^2 \geq 3.84$*. Therefore, we reject $H01$. In other words, we can say that we have statistically significant evidence at $\alpha = 0.05$ to show that vulnerability generation and StraightLine pattern are not independent (i.e., there is a relationship between using the StraightLine nano-pattern and generating vulnerabilities) where $p \leq 0.005$.

Table IV
BEST ASSOCIATION RULES

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

Table V
CONTINGENCY TABLE FOR NANO-PATTERN DEPENDENCY (VERSION 7)

Category	Present	Absent	Total
Vulnerable	13 (11.62)	3 (4.38)	16
Non-Vulnerable	9663 (9664.38)	3647 (3645.62)	13310
Total	9676	3650	13326

B. Research Question 2

Are the association rules useful in identifying vulnerabilities?

As we see from Table IV 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 non-vulnerable methods, there is no specific pattern between them.

The correlation measure among the two patterns $jdkClient$ and $tailCaller$ in non-vulnerable methods is 0.21716, which is not strong. The correlation among these two patterns is 100% within vulnerable methods. Our second hypothesis $H02$ is *Vulnerability generation is independent of dependencies among the nano-patterns*. We set $\alpha = 0.05$. According to Equation 1, the value of χ^2 is 2.74 for version 6 and 0.61 for version 7. For degrees of freedom $= 1$ and a 5% level of significance, the appropriate critical value is 3.84 and the decision rule is: *Reject $H02$ if $\chi^2 \geq 3.84$*. Therefore, we cannot reject $H02$ as chi-square value is 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 conclude that the association among the patterns may or may not have significant impact on vulnerability generation. In a future study, a further analysis of additional projects and vulnerabilities may yield a stronger result.

VII. DISCUSSION

From this experiment, we discovered rules that nano-patterns such as $localWriter$, $tailCaller$ and $jdkClient$ co-occur with the $localReader$ nano-pattern at 100% confidence. One goal for this study is to determine the co-occurrences of different patterns in a vulnerable code snippets. The rules that reveal co-existence of some nano-patterns in vulnerable code can lead the developers and testers to predict vulnerability due to their existence. The existence of the rule will give the developer an indication that the code is vulnerable. A tester would use the result to apply additional, targeted test to potentially vulnerable areas in the code. Thus we can say our study shows a preliminary indication of an answer to our second research question *Are the association rules useful in identifying vulnerabilities?* 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.

As Deo et al. in paper [8] found $localReader$ with high presence (95.97%) in defective methods, we also discovered that this pattern has a high correlation with our examined vulnerabilities which really confirms the results of the previous work. Our first research question is *Is there a relationship between code vulnerabilities and nano-patterns?* The preliminary findings indicate an association between nano-patterns with vulnerable code. Deo et al. [8] focused on figuring the defect prone nano-patterns while in this study, we also concentrate on finding the associations among the patterns in vulnerable code. Although we did not prove there is any correlation between vulnerability and dependencies among the patterns, we could at least figure out that some dependencies are frequent in defect prone methods compared to others. This will also help the developers to keep more concentration about their co-existence in code. This research will not only find out the correlations between nano-patterns and vulnerability, but it will also provide a layout of their distribution inside vulnerable versus non-vulnerable code. Therefore, the developers will be able to use the nano-patterns in the way that can make the code less defect-prone. In this regard, we can say that the experimental analysis with non-vulnerable code has made the hypothesis concluded by this work more reliable.

VIII. Conclusion

Software security is a major issue. In this paper, we analyzed Apache Tomcat versions 6.0 and 7.0 and focused on several vulnerabilities to reveal the hidden relationships among the nano-patterns and vulnerable code. In the future, we plan to analyze a larger dataset consisting of more data-points. Although the proposed study works with Java based patterns, the data mining method we adopted is generic. It can be easily employed for software in different platforms and frameworks for their vulnerability analysis with respect to other types of patterns. In this preliminary study, we focused on identifying rules indicating correlations among the nano-pattern with vulnerabilities. In our future work, we plan to incorporate class level patterns (micro-patterns) to make the process more effective. Moreover, we envisage building a classification model for the vulnerabilities based on these rules or relations. This model will play a significant role in predicting existence of any vulnerability inside codes and will indeed corroborate improved security in future systems.

References

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

[2] 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 (ENTCS) archive Volume 253 Issue 7, Pages 191-204, September, 2010.

[3] J. 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, San Diego, CA, USA, October 16-20, 2005.

[4] J. Han, and M. Kamber, *Data Mining: Concepts and Techniques*, 2000.

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

[6] J. Singer, and C. Kirkham, *Exploiting the correspondence between micro patterns and class names*, Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 6776, 2008.

[7] S. Kim, K. Pan, and E. Whitehead Jr., *Micro pattern evolution*, Proceedings of the International Workshop on Mining Software Repositories, pages 4046, 2006.

[8] 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), San Diego, CA, October 12-14, 2015.

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

[10] 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, November/December, 2011.

[11] Y. Shin, and L. Williams, *An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics*, Proc. Int'l Symp. Empirical Software Eng. and Measurement, pp. 315-317, 2008.

[12] Y. Shin, *Exploring complexity metrics as indicators of software vulnerability*, Proceedings of the third international doctoral symposium on Empirical Software Engineering, Kaiserslautem, Germany, October 2008.