Correlation Analysis among Java Nano-patterns and Software Vulnerabilities

Kazi Zakia Sultana
Mississippi State University
Mississippi, USA
Email: ks2190@msstate.edu

Ajay Deo
Mississippi State University
Mississippi, USA
Email: akd175@msstate.edu

Byron J. Williams
Mississippi State University
Mississippi, USA
Email: williams@cse.msstate.edu

***Abstract*—Ensuring software security is essential for developing a reliable software. A software can suffer from security problems due to the weakness in code constructs during software development. Our goal is to relate software security with different code constructs so that developers can be aware very early of their coding weaknesses that might be related to a software vulnerability. In this study, we chose Java nano-patterns as code constructs that are method-level patterns defined on the attributes of Java methods. This study aims to find out the correlation between software vulnerability and method-level structural code constructs known as nano-patterns. We found the vulnerable methods from 39 versions of three major releases of Apache Tomcat for our first case study. We extracted nano-patterns from the affected methods of these releases. We also extracted nano-patterns from the non-vulnerable methods of Apache Tomcat, and for this, we selected the last version of three major releases (6.0.45 for release 6, 7.0.69 for release 7 and 8.0.33 for release 8) as the non-vulnerable versions. Then, we compared the nano-pattern distributions in vulnerable versus non-vulnerable methods. In our second case study, we extracted nano-patterns from the affected methods of three vulnerable J2EE web applications: Blueblog 1.0, Personalblog 1.2.6 and Roller 0.9.9, all of which were deliberately made vulnerable for testing purpose. We found that some nano-patterns such as $objCreator$, $staticFieldReader$, $typeManipulator$, $looper$, $exceptions$, $localWriter$, $arrReader$ are more prevalent in affected methods whereas some such as $straightLine$ are more vivid in non-affected methods. We conclude that nano-patterns can be used as the indicator of vulnerability-proneness of code.**

I. Introduction

Software security is diminished when a system becomes susceptible to security attacks stemming from vulnerable code. The vulnerable code is often due to mistakes in programming practices which allow malicious attackers to exploit the weakness. If these weaknesses can be identified based on the structure of the vulnerable code, it will enable tools to automatically identify vulnerable areas and alert developers to potential problems. Researchers have identified traceable structural code constructs at both the method and class levels that can allow for such an analysis. The class-level constructs are known as *micro-patterns*, and the method-level constructs are known as *nano-patterns* [1], [2]. This study examines method-level code constructs and how they relate to vulnerabilities in the source code. The objective is to make developers aware when using constructs with a historical relationship to vulnerabilities. Method-level nano-patterns were chosen as the focus of this study because of their granularity level. Developers can focus on finding faulty methods and then make small changes or mark units for further testing. The goal of this study is not to find causation of vulnerability because the patterns may not be directly related to making code vulnerable. Rather it will make a distinction between patterns that are more frequent in vulnerable methods and those that are frequent in non-vulnerable methods. Frequent patterns in vulnerable methods may be directly or indirectly involved with code weaknesses that can be easily exploited by attackers, so the developers will feel secure about using safer patterns in their code and at the same time be more aware of those patterns that need rigorous testing.

Gil et al. [1] first introduced the concept of traceable patterns at the class-level. The micro-patterns he identified are formal conditions of Java classes. Gil defined 27 micro-patterns organized into 8 categories [1]. Batarseh developed the concept of nano-patterns in [2] applying Gil's ideas to Java methods. Singer et al. [3] then identified 17 fundamental nano-patterns organized into 4 groups and supplemented the list with additional types of nano-patterns [3].

The research goal is to determine if nano-patterns can be used as a way that can help reach vulnerable code. Preliminary work uncovered vulnerable areas in source code using nano-patterns [4]. The original study was limited to one system. We found frequent association rules among the nano-patterns in vulnerable methods [4]. We found that some pairs of nano-patterns are more frequent in vulnerable methods than they are in non-vulnerable methods. For example, $objCreator$ 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 our case, the association rules found at 90% minimum support and 100% confidence level were mostly related to two nano-patterns. Therefore, in our current study, we did not consider the association among multiple nano-patterns in vulnerable methods, rather we used phi-coefficient for every pair of nano-patterns in order to get frequent pairs of nano-patterns in vulnerable methods. One reason behind it is to reduce the amount of time wasted for generating the unnecessary association rules. In this study, we identified nano-pattern distributions in both vulnerable and non-vulnerable methods of Apache Tomcat [1]. We also

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

extended our analysis to three additional web applications which are deliberately made as vulnerable projects for testing purpose and detected patterns in vulnerable methods in order to substantiate the Apache Tomcat findings. We identified highly associated pairs of nano-patterns in vulnerable methods vs non-vulnerable methods. This analysis can be used to predict the existence of vulnerability by identifying connected nano-pattern pairs in code. Using one nano-pattern from the connected pair will keep the developer away from using of its peer pattern in order to reduce the risk of a security violation.

Our study is motivated for making developers aware of some vulnerability prone area in their code based on their coding practices. Although its primary concern is raising awareness about vulnerable methods, the results from this study can be used as a foundation for building a vulnerability prediction model. The metric will be the method-level patterns for assessing vulnerability. If we cannot determine the relationships between the patterns and vulnerability, it would not be possible to build any metric for assessing vulnerability based on patterns. A number of metrics have already been developed to measure security [5], [6], [7], [8], [9]. These metrics were then used for vulnerability prediction. However, the granularity level of these metrics is not well-defined. For example, in [5], complexity was determined for the file level where a file can be a set of methods and extremely large or extremely small in size. The performance of these metrics also varies depending on their threshold values. For example, $Countdecl Function$ is one of the complexity metrics that indicates the number of functions defined in a file and the hypothesis was *"Vulnerable files have a higher intrafile complexity than neutral files"* [5]. Now the question arises what is the cut-off point for the number of functions in a file that can make it more complex and vulnerable indeed. Therefore, the accuracy of the vulnerability prediction varies with the chosen cut-off value for the metrics. The major contributions of this paper are as follows:

- Our comparative analysis on the distribution of nano-patterns in vulnerable methods and methods where no vulnerabilities have been reported (we use the term non-vulnerable to refer these methods) will discriminate among the nano-patterns regarding their effect on making a method vulnerable. This will help the developers for deciding on the usage of the nano-patterns in their code. They will be more defensive about using the patterns that seem to be more frequent in vulnerable methods. Thus this study contributes in guiding vulnerability inspection during code development.
- The testers will also be able to be more focused during their testing. They will concentrate more on the methods having the suspected patterns and make rigorous testing on these methods. So this study also ensures efficient and time-saving testing efforts.

The remainder of the paper is organized as follows: Section II presents the related work. Section III discusses the research goal, research questions, and how the experiments were designed on Java systems. Section IV presents the case study conducted on Apache Tomcat. Section V presents the case study conducted on Stanford SecuriBench. Section VI shows all the findings in details and reasons behind them. Section VII shows the limitations of our work and Section VIII concludes the paper.

II. Related Work

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

A traceable pattern can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components [1]. The class-level traceable patterns are micro-patterns while the method-level patterns are called nano-patterns. Destefanis et al. first analyzed the fault proneness of micro-patterns and detected the micro patterns that are more likely to cause defects in code [10]. On the other hand, nano-patterns represent coding actions at the method level frequently used in Java software development [2]. Table I defines the 17 fundamental nano-patterns [3]. Singer et al. supplemented the fundamental nano-patterns by incorporating additional patterns and classified Java methods using data mining concepts: frequent itemset generation and association rule mining [3]. They applied their work to clustering and categorizing Java methods based on the associated nano-patterns. We used a similar approach for our analysis on nano-patterns to explore the relationship between nano-patterns and vulnerabilities [4]. We extended this prior work by adding versions of Apache Tomcat and three J2EE web applications with known vulnerabilities. In addition, we computed the phi-coefficient for every pair of nano-patterns in vulnerable versus non-vulnerable methods. Deo et. al. found certain nano-patterns to be more fault-prone than others [11]. This study was limited to finding nano-patterns associated with software defects and did not consider security defects.

Our study relates to software security assessment in the sense that we analyze the correlation of code patterns with vulnerabilities that will later be used to assess the security of the code. Software security metrics are used to detect vulnerabilities in code and provide some measure of confidence in the code base. The most widely used metrics for vulnerability prediction measure code complexity [5], [6], [7], [8], [12]. Complexity, at various levels, such as problem complexity, algorithmic complexity, cognitive complexity, and structural complexity, can be used for fault and vulnerability prediction. Shin et al. [6], [8] conducted an empirical study to analyze the impacts of complexity metrics on vulnerable and non-vulnerable files and identified distinct features of vulnerable functions compared with faulty functions. These methods suffered from high false negative rates. Their results revealed a weak correlation between code complexity and vulnerability. In another study, the authors used complexity, coupling, and cohesion (CCC) as early stage vulnerability detectors [9]. In order to make complexity metric more effective, they modified their training dataset as well as used three additional statistical and data mining techniques resulting in higher recall with

reasonable FP rate. Zimmermann et al. [13] conducted a large-scale empirical study to evaluate the efficacy of classical metrics such as complexity, churn, coverage, dependency measures, and the organizational structure of the company to predict vulnerabilities. In their study, they showed that classical metrics predict vulnerabilities with higher precision but lower recall and they concluded that vulnerability prediction is not as simple as defect prediction. For the accurate vulnerability prediction domain, and usage of the program components are needed to be captured.

TABLE I
CATALOGUE OF FUNDAMENTAL NANO-PATTERNS [3]

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 *Looping* — 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

III. METHODOLOGY

In this section, we describe the research goal, questions, and the procedure followed to address the questions.

A. Research Goal

The research goal is to determine if nano-patterns can be used as a way to help reach vulnerable code. The purpose of this goal is to reduce vulnerability prone coding by addressing vulnerability while developing the code. To determine vulnerability proneness of code, we first need to represent code with some coding constructs for which we chose nano-patterns that are traceable patterns defined on the properties of a Java method. If we can investigate some relationships between these nano-patterns and vulnerability, it will guide developers to develop a code having lower risks for being vulnerable. Moreover, the testers will also be facilitated by concentrating more on some types of patterns in code rather than searching for a needle in huge haystack resulting in time-efficient testing efforts. In addition, this analysis will be a foundation for building a more reliable security metric for vulnerability prediction in future.

B. Research Questions

Our research questions are as listed below:

- **Research Question 1 (RQ1):** *Is there any significant difference in nano-patterns' distributions in vulnerable vs non-vulnerable methods?*
  This question will determine the correlation between nano-patterns and vulnerabilities. We will answer this question by exploring the distribution of nano-patterns in both vulnerable and non-vulnerable code. We will find some interesting nano-patterns that show a significant difference in their frequencies in vulnerable versus non-vulnerable methods. For example, if the frequency of the pattern A is significantly high in vulnerable methods compared to the non-vulnerable methods, the pattern will definitely be an interesting pattern that may represent a vulnerable method by its presence and give a message to the developer to find a way for making the method non-vulnerable. For finding significant presence or absence, we will not only compare its frequencies in vulnerable versus non-vulnerable methods but also conduct a hypothesis testing to validate the finding statistically.
  We formulate our Hypothesis $H0$ as follows:
  $H0$: *Software vulnerabilities are independent of nano-patterns contained in their source code*.
  If we can reject this hypothesis for a particular nano-pattern, we can say that the particular nano-pattern and software vulnerability are related to each other. In other words, vulnerability may sprout out of the method having that pattern. We do not claim that the presence of that nano-pattern is the cause for that vulnerability; rather, we suggest to consider the method for further testing.
- **Research Question 2 (RQ2):** *How are the nano-patterns associated with each other in vulnerable versus non-vulnerable code?*
  To answer this research question, we got the phi-coefficient for each pair of nano-patterns in vulnerable and non-vulnerable code using a statistical tool. The connected pairs found in methods indicate that they frequently exist together. The developers can avoid using them together in order to make the code secured if they understand that their co-existence may make the code vulnerable. As phi-coefficient is a statistical measure, we did not conduct any further statistical test for the findings as we did for the first research question.

C. Study Design

Apache Tomcat [1], an open source web application server, was the target for our initial case study. We retrieved the list of vulnerabilities reported for all versions from the Apache Tomcat Vulnerability Reports site [2]. Apache Tomcat contains about a half million lines of code and more than 3000 classes. The vulnerabilities reported on the site are listed in Table II. This table shows the reported vulnerability types across 3 major releases; 6, 7, and 8. In this study, we concentrate on

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

TABLE II
VULNERABILITIES

Information disclosure
Security Manager bypass
Request Smuggling
Information disclosure
Frame injection in documentation Javadoc
Session fixation
DIGEST authentication weakness
Denial of Service
Bypass of security constraints
Bypass of CSRF prevention filter
Authentication bypass and information disclosure
Multiple weaknesses in HTTP DIGEST authentication
Cross-site scripting
SecurityManager file permission bypass
Remote Denial Of Service and Information Disclosure
Information disclosure in authentication headers
Arbitrary file deletion and or alteration on deploy
Insecure partial deploy after failed undeploy
Unexpected file deletion in work directory

these vulnerabilities for Tomcat as they reported these vulnerabilities for the releases. The source code was downloaded from Apache Tomcat Archives [3]. We considered 39 versions of Tomcat where vulnerabilities were reported and later fixed in the source code. In our second case study, we did the experiment for three more J2EE web applications Blueblog 1.0, Personalblog 1.2.6 and Roller 0.9.9, all of which are completely different from Apache Tomcat. Table VI shows the statistics of these applications.

Singer et al. in [3] developed a tool to detect nano-patterns in Java byte code. This tool provides the list of all methods and their associated nano-patterns for a given code base. We used another tool, JiraExtractor [11], that uses this nano-pattern tool to dump the nano-patterns' information of the methods in a particular version into the database. Then, this tool extracts the methods modified for each revision number involved in fixing a vulnerability of that version. After that, it pulls nano-patterns of those methods from the database that already contains the nano-patterns' information for the software version. We computed the phi-coefficients for each pair of nano-patterns to answer RQ2 using the SPSS tool [4].

IV. CASE STUDY 1: APACHE TOMCAT

A. Data Collection

We collected Apache Tomcat vulnerability reports that provide the information about the vulnerability type, its CVE (Common Vulnerabilities and Exposures) [5] id, affected versions, revision number, fixed version, and severity level of the vulnerabilities fixed in the identified versions. For example, if a vulnerability affects the versions 6.1, 6.2, 6.3 and is fixed in 6.4, we consider 6.3 as the last affected version. By doing so, we collected the last affected code versions for the listed vulnerabilities as shown in Table III. On the other hand, the column "Non-Affected version" contains the version where no vulnerability was reported for the releases 6, 7 or 8. For example, 6.0.45 was the last non-affected version in release 6, 7.0.69 was the last non-affected version in release 7, and 8.0.33 was the last non-affected version in release 8. The reason behind it is that the vulnerabilities that we considered for each major release are not available in the last version of that release as they are already fixed. Although some other vulnerabilities may still exist in this last version, we are only interested about the vulnerabilities that were reported only in that release, and no more exist. Then, we downloaded the code for all different *affected* and *non-affected* versions listed in Table III. The collected data from the Apache Tomcat Security page as shown in Figure 1 are CVE id (CVE-2014-0075), Vulnerability Type (Denial of Service), Last Affected Version (7.0.52), and Revision No (1578341).

TABLE III
AFFECTED VERSIONS

Major Release	Affected Versions	Non-Affected Version	Affected Methods
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	6.0.45	124
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	7.0.69	106
8	8.0.0-RC1, 8.0.0-RC5, 8.0.1, 8.0.3, 8.0.5, 8.0.8, 8.0.15	8.0.33	21

B. Data Extraction

We used the JiraExtractor tool to get the list of methods changed for a revision [11]. The tool fetches the nano-patterns of those methods using Singer's nano-patterns extraction tool. For example, we found the list of vulnerabilities which affected a specific version of Apache Tomcat listed in **Affected Versions** column of Table III and then used the nano-patterns tool to dump all methods and their nano-patterns of that version in the database. After that, the JiraExtractor tool gets all the revision numbers for that version and fetches the list of all methods in that version that have been revised for vulnerability removal. Finally, the tool extracts the nano-patterns of those methods from the database table. We found 124, 106, and 21 affected methods in the releases 6, 7, and 8 respectively as presented in Table III. Then, we collected nano-patterns from the source code of three non-vulnerable versions 6.0.45, 7.0.69, and 8.0.33 in releases 6, 7, and 8 respectively by using the nano-patterns tool by Singer.

C. Data Analysis

- **Research Question 1 (RQ1):** *Is there any significant difference in nano-patterns' distributions in vulnerable vs non-vulnerable methods?*
  Our dataset contains the nano-patterns for all the affected methods of the affected versions in releases 6, 7, and

[3]http://archive.apache.org/dist/tomcat/
[4]http://www-01.ibm.com/software/analytics/spss/products/statistics/index.htm
[5]https://cve.mitre.org/

Fig. 1. Apache Tomcat Security Page

8 of Apache Tomcat. We calculated the percentage of each nano-pattern in affected methods for three different releases. We also calculated the percentage of each nano-pattern in non-affected methods of 6.0.45, 7.0.69, and 8.0.33 separately. We then illustrated the nano-pattern frequency for affected vs non-affected methods using grouped vertical bar charts. Bar charts provide a visual presentation of categorical data.

- **Research Question 2 (RQ2):** *How are the nano-patterns associated with each other in vulnerable versus non-vulnerable code?*
  To address this question, we found the phi-coefficient for each pair of nano-patterns in affected and non-affected methods of releases 6, 7, and 8 for Apache Tomcat. The strength of the association is determined by following criteria [14]: Small association ($.10 \leq \phi < .30$), Medium association ($.30 \geq \phi < .50$), High association ($\phi \geq .50$). We found the connected pairs in all types and presented how their associations differ in affected versus non-affected methods. This analysis will help us to determine if the association among nano-patterns contributes to the vulnerability generation.

D. Results

In this section, we present the results obtained from the analysis for two research questions we described.

1) Research Question 1 (RQ1): **Is there any significant difference in nano-patterns' distributions in vulnerable vs non-vulnerable methods?**

The comparative study between nano-patterns' distribution in the vulnerable methods and that in the non-vulnerable methods shows significant differences between them which lead to the conclusion that some nano-patterns are more frequent in vulnerable methods than they are in non-vulnerable methods and vice versa. For space restriction, we present the result for release 8 as the results for other two releases are similar to release 8.

Our significant observations are as follows:

- There are some nano-patterns that are more widely present in affected methods compared to the non-affected methods as shown in Figure 2. They are $objCreator$, $thisInstanceFieldReader$, $thisInstance-FieldWriter$, $otherInstanceFieldReader$, $otherInstance-FieldWriter$, $looper$, $exceptions$, $localWriter$, $arrReader$.
- There are some nano-patterns that are more widely present in non-affected methods compared to their presence in affected methods as shown in Figure 2. They are $void$, $samename$, $leaf$, $tailCaller$, and $straightLine$.

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

TABLE IV
CHI SQUARE VALUES FOR NANO-PATTERNS

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

2) Research Question 2 (RQ2): **How are the nano-patterns associated with each other in vulnerable versus non-vulnerable code?**

We computed the phi-coefficient for each pair of nano-patterns in the affected methods of releases 6, 7, and 8 [15]. We considered only high and medium associations and present the connected pairs of nano-patterns of all the affected methods in release 6, 7, and 8 in Table V.

V. CASE STUDY 2:STANFORD SECURIBENCH

In this case study, we conducted our experiment on three Java web applications that contain vulnerabilities explicitly

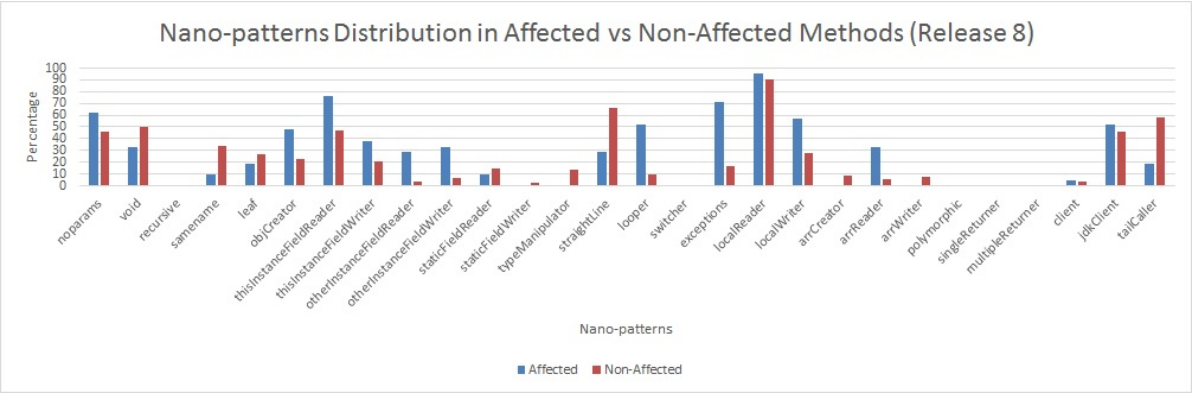

Fig. 2. Nano-patterns Distribution in Affected vs Non-Affected Methods (Release 8)

TABLE V
NANO-PATTERNS ASSOCIATION TYPES IN AFFECTED METHODS OF APACHE TOMCAT

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

injected for security testing purposes. Stanford SecuriBench [6] is a set of open source programs used as a test-bed for static and dynamic security tools [16]. Release .91a focuses on Web applications written in Java. We downloaded the source code from their download site where Version .91a of SecuriBench is available and selected blueblog 1.0, personalblog 1.2.6, and roller 0.9.9 for our study. The statistics of these three applications are given in Table VI.

TABLE VI
SECURIBENCH PROGRAM STATISTICS

Benchmark	Version Number	File Count	Line Count
blueblog	1.0	32	4191
personalblog	1.2.6	39	5591
roller	0.9.9	276	52089

A. Data Collection

We installed the Eclipse plugin for Early Security Vulnerability Detector - ESVD [7] in order to detect the vulnerable methods of three vulnerable projects. We got 6 vulnerabilities in blueblog 1.0, 34 vulnerabilities in personalblog 1.2.6, and 207 vulnerabilities in roller 0.9.9. We used ESVD for finding the vulnerabilities as it was shown to have less false positive with higher precision and recall [8]. The vulnerability types that were found in these three applications are Cross-Site Scripting, HTTP Response Splitting, SQL Injections, Path Traversal, and Log Forging. We then explored the methods that are associated with these types of vulnerabilities. There are 3, 10, and 139 vulnerable methods in blueblog 1.0, personalblog 1.2.6, and roller 0.9.9 respectively detected by ESVD.

[6]http://suif.stanford.edu/ livshits/securibench/

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

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

B. Data Extraction and Analysis

We extracted the nano-patterns of all the vulnerable methods in these three web applications using the nano-patterns tool. Then, we analyzed the percentage of each type of nano-patterns in the affected methods and also found the phi-coefficient in each pair of nano-patterns in the vulnerable methods.

C. Results

1) Research Question 1 (RQ1): **Is there any significant difference in nano-patterns' distributions in vulnerable vs non-vulnerable methods?**

According to the Figure 3, the most prominent nano-patterns in vulnerable methods of three web applications are $void$, $objCreator$, $thisInstanceFieldReader$, $staticFieldReader$, $typeManipulator$, $looper$, $exceptions$, $localReader$, $localWriter$, $jdkClient$, and $tailCaller$. On the other hand, $samename$, $leaf$, and $straightLine$ were identified as significantly less prominent in affected methods compared to their existence in non-affected methods.

2) Research Question 2 (RQ2): **How are the nano-patterns associated with each other in vulnerable versus non-vulnerable code?**

The connected pairs of nano-patterns in these projects have been presented in Table VII.

VI. DISCUSSION

A. Research Question 1 (RQ1)

Is there any significant difference in nano-patterns' distributions in vulnerable vs non-vulnerable methods?

Deo et al. in the paper [11] found that some nano-patterns such as $localReader$, $localWriter$, $fieldReader$, and $objCreator$ have a high presence in defective methods (i.e., they are more error-prone than other patterns). In our experiment, we also discovered that a set of patterns

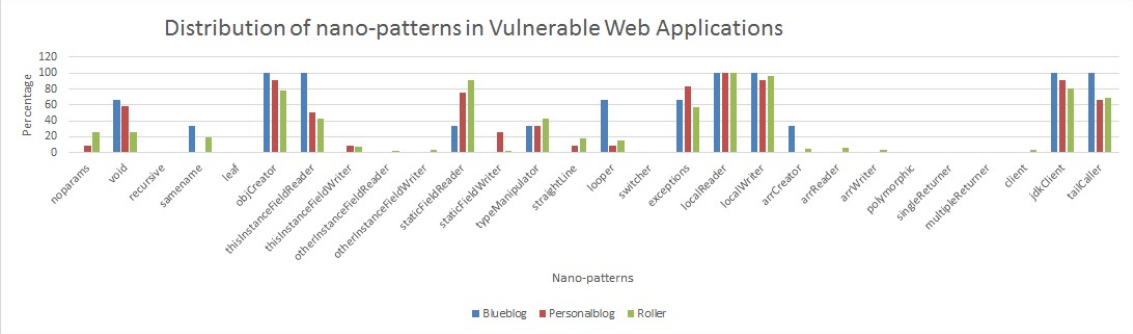

Fig. 3. Nano-patterns Distribution in Affected Methods of Vulnerable Web Applications

TABLE VII
NANO-PATTERNS ASSOCIATION TYPES IN AFFECTED METHODS OF WEB APPLICATIONS

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

including these patterns have a high correlation with our examined vulnerabilities. Deo et al. [11] focused on defect prone nano-patterns while we analyzed patterns of their potential involvement and their pair-wise associations in security vulnerabilities. We found that some patterns are highly frequent in affected methods compared to their presence in non-affected methods. If we see the following code of the $importOldData$ method in $PersonalBlogService.java$ file of PersonalBlog as in Figure 4, it contains $FieldReader$ pattern as it reads field values from an object. The line $xyz.setTitle(rs.getString("title"))$; of the method that reads field values of the ResultSet object has been identified as having vulnerability because $rs.getString("title")$ was not sanitized before its use. Therefore, we can generalize that the methods that read values from objects must be checked for proper sanitization in order to keep the code secured. In this case, our target is not to detect the cause of any vulnerability; rather, reduce the risk of vulnerability by making developer aware of the code weakness that can easily be exploited by an attacker. For this, we have taken advantages of the nano-patterns that represent the properties of a Java method. On the other hand, $samename$, $leaf$, and $straightLine$ are comparatively more frequent in non-affected methods. Their relations with vulnerabilities have also been statistically verified using chi-square tests. Table IV presents the chi-square values in decreasing order. Moreover, the distributions of these nano-patterns in three vulnerable web applications and in Apache Tomcat are identical which strengthen the claim about the proper use of nano-patterns.

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

Fig. 4. A code snippet from a method in $PersonalBlogService.java$ of PersonalBlog

B. Research Question 2 (RQ2)

How are the nano-patterns associated with each other in vulnerable versus non-vulnerable code?

To answer RQ2, we computed the phi-coefficient for each pair of nano-patterns in the affected methods. The associations of Apache Tomcat are listed in Table V. In our study, we got phi-coefficient 0.514 for the pair $looper - arrReader$ in Apache Tomcat which means $looper$ and $arrReader$ are highly associated with each other in affected methods. According to the study in [3], $arrReader \rightarrow looper$ is an interesting rule that is more frequent in Java methods due to the iterating over an entire array, reading each element per iteration, so we see that our study also supports their finding adding new knowledge that this association may make a Java method insecure and need attention in vulnerability testing.

Some of the associations such as $objCreator - jdkClient$, $localWriter - jdkClient$, $objCreator - localWriter$, $typeManipulator - jdkClient$, $objCreator - typeManipulator$, and $objCreator - straightLine$ are common in all types of applications studied here. If we see the following code of the $getUserInfo$ method in $BloggerAPIHandler.java$ file of roller as in Figure 5, it contains $objCreator$ pattern as it creates a new $XmlRpcException$ object here. It also contains $localWriter$ pattern because it assigns local variable msg a value. This line of the method has been identified as having vulnerability because e may contain sensitive information and can be exploited by an attacker for information leakage. Figure 6 shows another code snippet from $ConsistencyCheck.java$ that contains $localWriter$, $objCreator$, $jdkClient$ together. This code was marked as having SQL Injection vulnerability as string concatenation is not allowed in queries. Therefore, methods where a local variable is assigned a string value or a

newly created object should be rigorously tested.

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

Fig. 5. A code snippet from a method in $BakeWeblogAction.java$ of roller

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

Fig. 6. A code snippet from a method in $ConsistencyCheck.java$ of roller

VII. Threats to Validity

The vulnerable methods of a version of Apache Tomcat have been considered as non-vulnerable in the last version of that release, but the fact is that a vulnerability may exist in the method, just not yet reported. The experiment was conducted on one system - 42 versions of Apache Tomcat and then validated for three java based web applications with seeded vulnerabilities. As nano-patterns are defined only for Java methods, we limited this study to the methods of Java-based applications, so it cannot be concluded that the results are generic and applicable to other systems of different platforms and programming languages. Moreover, we do not claim causation between nano-patterns and vulnerabilities; rather, we relate software vulnerabilities with the presence of nano-patterns. The nano-patterns themselves may not be the cause of the vulnerabilities, but their frequency in affected methods indicates their potential to be problematic and susceptibility to security attacks.

VIII. Conclusion and Future Plan

We conducted an empirical analysis on nano-patterns extracted from the affected and non-affected methods of 42 versions of three major releases of Apache Tomcat and three vulnerable web applications: blueblog, personalblog, and roller. We found some patterns to be more frequent in affected methods while some patterns that were relatively more frequent in non-affected methods. These findings will certainly help developers to use them properly in their code such that they can avoid security leaks. Moreover, testers will pay special attention to the methods having the patterns that are more frequent in vulnerable code. In addition, we extracted the association between every two nano-patterns in affected methods in order to assess their co-occurrence in vulnerable code. Our case study was limited to Java based projects due to the fact that nano-patterns are defined for Java methods. This is the first effort at finding relationships between nano-patterns and security vulnerabilities. In the future, we plan to generalize the study for other systems. After that, we plan to develop a security metric based on these patterns by incorporating threshold values with each pattern weighting their responsibility for generating specific vulnerabilities.

Acknowledgment

We would like to thank Dr. Dave Dampier and the Distributed Analytics and Security Institute (DASI) at Mississippi State University for their support of this work. DASI is dedicated to excellence in the areas of distributed computing, big data analytics, cyber security, and critical infrastructure protection.

References

[1] 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.

[2] 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.

[3] 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).

[4] K. Z. Sultana, A. Deo, and B. J. Williams, "A preliminary study examining relationships between nano-patterns and software security vulnerabilities," in *The 40th IEEE Computer Society International Conference on Computers, Software and Applications*, Atlanta, Georgia, USA, 2016.

[5] 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.

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

[9] 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. New York, NY, USA: ACM, 2010, pp. 1963–1969.

[10] 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.

[11] 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.

[12] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM Workshop on Quality of Protection*, ser. QoP '08. New York, NY, USA: ACM, 2008, pp. 47–50.

[13] 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. Washington, DC, USA: IEEE Computer Society, 2010, pp. 421–428.

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

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

[16] 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.