Noname manuscript No.
(will be inserted by the editor)

A Study Examining Relationships between Micro Patterns and Security Vulnerabilities

Kazi Zakia Sultana · Byron J. Williams · Tanmay Bhowmik

Received: date / Accepted: date

Abstract Software security is an integral part of software quality and reliability. Software vulnerabilities make the software susceptible to attacks which violates software security. Metric-based software vulnerability prediction is one way to evaluate vulnerabilities beforehand so that developers can take preventative measures against attacks. In this study, we explore the correlation between software vulnerabilities and code level constructs called micro patterns. These code patterns characterize class-level object-oriented program features. Existing research addressed micro pattern correlation with software defects. We analyzed the correlation between vulnerabilities and micro patterns from different viewpoints and explored whether they are related. We studied the distribution of micro patterns and their associations with vulnerable classes in 42 versions of the Apache Tomcat and three Java web applications. This study shows that certain micro patterns are frequently present in vulnerable classes. We also show that there is a high correlation between certain patterns that co-exist in a vulnerable class.

Keywords Software security · software vulnerabilities · micro patterns · software quality

1 Introduction

Security is a major concern in the developer community for maintaining software quality and determining its reliability. If the software can be exploited by malicious attacks, it will not be reliable to the customers. Therefore, suspected software must be re-engineered or deprecated to reduce the risk of security violations. There are different security policies and practices that are followed in different phases of software development lifecycle. For the implementation

Department of Computer Science and Engineering
Mississippi State University
E-mail: ks2190@msstate.edu

phase, secure coding practices can be followed to create a robust codebase free from common security weaknesses[1]. While these coding practices have been vetted by the software engineering research and practitioner communities, following these practices does not always ensure vulnerability free code. There is need for methods that can highlight certain code constructs that exhibit vulnerable characteristics of that code area. Various security models focus on different phases of the software development including requirements analysis phase, design phase, implementation phase, and testing phase to ensure security. The existing metrics at the code or implementation level are basically software metrics that are used for security assessment, and they have high false negative rates [11, 19]. Moreover, these metrics do not consider the correlations of code constructs with vulnerabilities, and therefore, the developers do not get any guideline for secure coding practices. As our goal is to assess security based on code constructs, we focus on the implementation phase using class-level code patterns to highlight potentially vulnerable areas in the code. These patterns capture the object-oriented features of code at class-level and have not been analyzed extensively yet. The suspected code areas having vulnerability prone patterns can then be marked for targeted testing or more rigorous reviews based on the vulnerability and history of the project.

This study uses traceable, class-level software patterns easily identified in the source code and determines their relationships with vulnerabilities. Gil et al. [1] developed the concept of traceable patterns that can be automatically (mechanically) recognized. These patterns are related to a specific programming language and have different levels of abstraction. Class-level traceable patterns are called *micro patterns* whereas method-level traceable patterns are called *nano-patterns*. Gil et al. [1] defined 27 micro patterns organized into eight categories with respect to the formal conditions on the structure of Java classes. They capture class properties whereas nano-patterns are method-level patterns and capture properties of methods within a class. Batarseh first introduced the idea of nano-patterns in [7]. Singer et al. [8] listed 17 fundamental nano-patterns organized into four groups.

The fault-proneness of micro patterns has been studied in several research [2, 3]. They focused on improving software quality by reducing the use of bug-prone micro patterns. In our earlier work, we found potentially vulnerable areas in code using method-level patterns (nano-patterns) [9, 10]. Fault-proneness of the nano-patterns has also been detected by Deo et al. in their recent work [18]. Although nano-patterns are more granular than micro patterns, they are method-based patterns and do not capture the object-oriented features. On the other hand, micro patterns are defined based on object-oriented features, and they can better capture the vulnerabilities that result from object-oriented characteristics including encapsulation, inheritance and polymorphism. For example, according to [1], *Extender* is a class which extends the interface inherited from its superclass and super interfaces but does not override any method. In [2], the author showed that *Extender* is a fault-prone pattern. Therefore,

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

in this work, we focus on class-level patterns to detect the likelihood of their relationship with security defects. Our research goal is to determine the relationships between these micro patterns and vulnerable code which can later be used for vulnerability prediction. We have found the micro patterns' distributions in both the vulnerable and non-vulnerable classes and then identified the micro patterns that frequently exist in vulnerable classes compared to the non-vulnerable classes. We have also analyzed the relationship between different micro patterns and different types of vulnerabilities, including Denial of Service, Information Disclosure, and Security Bypass, so that developers can also predict which type of vulnerability can be resulted from the code construct. We statistically analyzed the micro patterns both in vulnerable and non-vulnerable Java classes and then identified how they are modified when a change is made to eliminate the vulnerability.

In our study, we also analyzed how micro patterns are changed from vulnerable classes to non-vulnerable classes. Kim et al. found how micro patterns are evolved from buggy code to non-buggy code and detected the evolution types that are more bug-prone than others [4]. Some researches investigated how the associations among the micro patterns result in code smell, which is a key indicator of software quality degradation [16]. Sultana et al. identified some association rules among the nano-patterns that are more frequent in vulnerable code [9]. In this study, we computed the phi-coefficient between each pair of micro patterns to find its association in vulnerable classes and non-vulnerable classes. We identified high and medium associations in the pairs of micro patterns in vulnerable classes. This analysis will help the developers to be concerned about the use of connected patterns in code. Moreover, it will assist them to predict the existence of vulnerability from another viewpoint by identifying connected micro pattern pairs in code. When the developer uses one micro pattern from the connected pair, he should try to avoid the use of its peer micro pattern in order to reduce the risk of security violation or at least be careful about using them together.

Our research goal is to use micro patterns to detect vulnerable classes in order to address vulnerabilities during implementation prior to release. Software vulnerabilities are usually reported separately as they need to be addressed more carefully than general defects. For example, Apache Ant[2] and Microsoft[3] have security advisories reported separately from their bug database. Moreover, software vulnerabilities may not always be visible to the testing tools during software testing as they fundamentally capture faults that are visible during runtime. For example, Bau et al. conducted a study over eight blackbox web application vulnerability scanners which probe web applications for security vulnerabilities without accessing source code [50]. The study showed that some forms of Cross Site Scripting (XSS) and SQL Injection (SQLI) vulnerabilities are not currently found by many tools [50]. Testing guidelines of

[2] http://ant.apache.org/security.html
[3] https://technet.microsoft.com/en-us/library/security/4010323.aspx

OWASP[4] also determine that many security problems are extremely difficult to discover with technical testing. Therefore, vulnerability detection during the implementation phase is as important as defect detection while both cases need to be analyzed separately. Camilo et al. in [48] showed that bugs and vulnerabilities are empirically dissimilar groups, and more research is needed for targeting vulnerabilities specifically. Therefore, developers need to find the vulnerabilities (particularly those caused by poor design choices) implemented in their code as often as defect by testers in the early stage of development. In order to accomplish our goal, we developed the following research questions to guide our work:

- **Research Question 1 (RQ1):** *Is there any significant difference in micro patterns' distributions between vulnerable and non-vulnerable classes?*
- **Research Question 2 (RQ2):** *How are the micro patterns related to different types of vulnerabilities?*
- **Research Question 3 (RQ3):** *How do the micro patterns evolve from vulnerable classes to non-vulnerable classes?*
- **Research Question 4 (RQ4):** *How are the micro patterns associated with each other in vulnerable and non-vulnerable code?*

We address all research questions using the Apache Tomcat datasets. Research questions RQ1 and RQ4 are evaluated using our secondary dataset, the Stanford Securibench, which did not contain information across development versions needed to address questions RQ2 and RQ3.

Our motivation is to reduce the number of vulnerabilities inserted into code during implementation and maintenance. The existence of micro patterns that have been shown to be more vulnerable than others highlights code that is potentially vulnerable code. As developers are made aware of the more problematic coding constructs, they can be more cautious when using them. When no alternatives exist, developers can program using the constructs but from a defensive standpoint (recognizing that their approach may be troublesome). In doing so, both developers and testers can further target these problematic areas for testing, which can focus on specific types of vulnerabilities [31]. A variety of metrics defined with a focus on complexity, code churn, cohesion, and coupling have been developed for measuring security [11–14, 19, 21]. Although they can fairly accurately assess overall software quality, there is still a need for mechanisms to detect vulnerable code at varying levels of granularity. To that end, our study on micro patterns and their correlation with specific types of vulnerabilities gives an indication of what type of vulnerability may occur from the code. Furthermore, the study on associations among the micro patterns in vulnerable classes tells us how their joint occurrence affects code.

Our empirical model focuses on specific types of vulnerabilities that are present in the subject system. This model can be extended for any type of software defects. The major contributions of this paper are as follows:

[4] https://www.owasp.org/index.php/Testing_Guide_Introduction

- We analyze the distribution of micro patterns in vulnerable classes and classes where no vulnerabilities have been reported. Our comparative analysis on the distribution of micro patterns will discriminate among the micro patterns regarding their effects on making a class vulnerable. This will assist the developers to be more restrictive in the usage of the micro patterns as well as guide them to inspect their code for potential vulnerability.
- The findings of this study will help testers become more focused on the potentially vulnerable areas of code instead of the entire code base. It will definitely save time and effort of the testers and ensure efficient testing for the suspected code.
- Our vulnerability study at the class level will help the developers to concentrate at the lower granularity level of code and pinpoint the vulnerable components more easily. This analysis will develop a good understanding of secure code among the developers. They will be able to ensure reliable system evolution by re-engineering the later versions with the proper usage of these code constructs.
- This study is a foundation for developing a security metric using these patterns and building an effective prediction model along with other code level security metrics.

The remainder of the paper is organized as follows: Section 2 presents the necessary terms that we use in this study. Section 3 discusses the related work. Section 4 illustrates the methodology of the proposed technique. Section 5 presents the case study conducted on Apache Tomcat. Section 6 presents the second case study conducted on Stanford SecuriBench. Section 7 elaborates all the results in details. Section 8 shows the limitations of our work and Section 9 concludes the paper.

2 Background

This section describes terms used in this paper.

- **Vulnerability:** "A vulnerability is a security exposure that results from a product weakness that the product developer did not intend to introduce and should fix once it is discovered."[5] Another definition is "An information security 'vulnerability' is a mistake in software that can be directly used by a hacker to gain access to a system or network."[6] Examples of vulnerabilities are Cross-site scripting (XSS) which allows remote authenticated users to inject arbitrary web script or HTML, denial of service that refuses users' requests for a service through the network, injection flaws which occur when untrustworthy data is sent to an interpreter as part of a command or query, and Broken Authentication and Session Management

[5] https://msdn.microsoft.com/en-us/library/cc751383.aspx
[6] https://cve.mitre.org/about/terminology.html

which allows attackers to compromise passwords, keys, or session tokens or to exploit other implementation flaws to assume other users' identities.[7]
- **Micro Pattern:** Micro patterns are mechanically recognizable patterns that are captured using formal conditions of the structure of a Java class such as use of inheritance, immutability, data management and wrapping, restricted creation etc. They are similar to design patterns except that they are defined at a lower level of abstraction [1]. Design patterns are defined on the entire design of the system. They are abstract and not directly traceable from code. On the other hand, micro patterns are defined on class level, and each single class may contain a set of micro patterns. The combination of these micro patterns represent that class. These micro patterns are traceable and can be extracted from source code. So micro patterns are more granular than design patterns [1]. The detailed description of 27 micro patterns is presented in Table 1.
- **Micro Pattern Evolution:** Examining the development history of a system explores facts about the software and enables a better understanding of its qualities [4]. In this study, we examine the change history of micro patterns in different versions of the same system. For example, a vulnerability has been detected in foo.java of version 6.0.1, and its pattern type is A, later the class is changed to fix the vulnerability in version 6.0.2, and now its pattern type is B after fixing the vulnerability in that class. This change will be termed as a $A \rightarrow B$ evolution.
- **Affected Class or Vulnerable Class vs Non-Affected Class or Non-Vulnerable Class:** If a vulnerability is fixed in foo.java of version 6.0.2, that class of version 6.0.2 or later versions will be termed as a 'non-affected' or 'non-vulnerable' class. On the other hand, the source code of foo.java in all the previous versions of that release will be termed as an 'affected' or a 'vulnerable' class.
- **Phi-Coefficient:** Phi-Coefficient is the linear correlation between postulated underlying discrete univariate distributions of X and Y [30]. It measures the degree of association between two binary variables. The strength of the association is determined by following criteria [32]: a small association ($.10 \leq \phi < .30$), a medium association ($.30 \geq \phi < .50$), a high association ($\phi \geq .50$).

3 Related Work

This section presents relevant research on software security, code patterns, security 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]. These patterns are based on modules including code fragments, routines, classes and packages. Gil and Maman described the catalog of 27

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

Table 1 Micro patterns in the catalog [1]

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

micro patterns on Java classes and interfaces [1]. Table 1 presents the categories of micro patterns and their description. Arcelli and Maggioni suggested a different approach for interpreting micro patterns based on the number of attributes (NOA) and the number of methods (NOM) of a type [5]. Their identified types are not completely aligned to the constraints and definitions of the micro patterns [5]. Destefanis et al. identified micro patterns that were more error-prone than others and detected some correlations among the patterns [2, 3]. They also found that the classes having no micro pattern are supposed to be more fault-prone than the classes with micro patterns. Kim et al. detected the micro patterns evolution along with the program's evolution throughout the development process [4]. Their analysis on micro pattern evolution from one revision to another reported some types of micro pattern evolution to be more bug-prone than others. They performed their micro pattern evolution analysis on three open source projects: ArgoUML, Columba, and jEdit. The kinds of evolution are almost identical across all the projects. In another study, Singer et al. extracted the association between micro patterns with class name suffix which might be useful for run-time bug detection by the developers [6].

There have been a number of security patterns proposed for different phases of software development [37]. According to Schumacher et al., "A Security Pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution. A Security Pattern System is a collection of security patterns, together with guidelines for their implementation, combination and practical use in security engineering" [44]. Yoder et al. presented seven security patterns in [45]. In [46], Bunke et al. showed that only one-fourth of the software security patterns provide code examples, and their quality is not as good as the design pattern examples. Moreover, in contrast to design patterns, security patterns are not investigated as often in the area of software maintenance. There are also security patterns designed to offer guidelines for developers to follow during implementation [38–43]. Many of the security patterns and guidelines for the implementation phase have been adopted. However, there is no concrete and consistent set of patterns that are easily traceable [37]. In addition, in [49], security patterns are presented at the class level using class diagrams, which implies that the correlation between micro patterns and security patterns can be investigated in further research. Therefore, our research on traceable patterns and software security will play an important role in this area.

Our study contributes new knowledge in the field of software security assessment. Our analysis on the correlation of code patterns with vulnerabilities can later be used for identifying vulnerable source code. Therefore, our work is a step forward towards a vulnerability prediction system for software. Researchers developed a number of software security metrics to assess the security of a software system. These metrics are used to predict vulnerabilities in code based on some predefined threshold values at a certain confidence level. Most of the existing literature focuses on complexity metrics for software quality assessment. Researchers have found that complexity is related with software faults and other problematic issues [13]. Researchers also determined that the fault prediction models based on complexity metrics might be useful for vulnerability prediction, and some metrics, such as nesting complexity metrics, are more effective for locating vulnerable code than locating faulty code [15]. Shin et al. [12,14] conducted an empirical study to analyze the impacts of complexity metrics on vulnerable and non-vulnerable files. They also determined that vulnerable functions have distinctive characteristics from non-vulnerable functions and from faulty but non-vulnerable functions in terms of code complexity. They showed that fault prediction models based on traditional metrics such as code churn, complexity and fault history provide similar performances in vulnerability prediction across a wide classification threshold [14]. In another study, the authors analyzed how different complexity metrics, code churn, and developer activity metrics can be used to predict vulnerabilities [11]. They showed that these metrics are positively correlated to vulnerabilities. These studies attempted to improve on existing code-level analysis studies for vulnerability prediction. Although the techniques resulted in adequate vulnerability prediction, the methods suffered from high false negative rates [11–13]. Chowdhury et al. in [19, 21] defined how different code structures cause vulnerable source code. The authors empirically showed that complexity, coupling, and cohesion (CCC) can be effectively used as metrics to detect vulnerability in the early stage of development. They used four alternative data mining and statistical techniques, such as C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive-Bayes, and compared their performance on vulnerability prediction [21]. They did not define any threshold of the metrics which can measure the vulnerability level of a class. On the other hand, we clearly define some patterns as vulnerable or unsafe patterns and some patterns as safe or non-vulnerable patterns. Another study on complexity metrics showed that these metrics are good predictors of vulnerability between different releases of a project and also between different projects in various fields [17].

Structural complexity metrics can exist at the code or design level. Alshammari et al. in [23] introduced seven metrics for assessing the security of an object-oriented class. Chowdhury et al. also proposed three code level metrics: stall ratio, coupling corruption propagation, and critical element ratio to assess the security in a software [20]. 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 [20]. In another study, authors showed that relative code churns can be used as a predictor of defect density in software systems [22]. Zimmermann et al. [24] demonstrated a large-scale empirical study to evaluate the efficacy of classical metrics such as complexity, churn, coverage, dependency measures, and organizational structure of the company to predict vulnerabilities.

Neuhaus et al. [12] introduced a new metric called import and function-call to construct the prediction model for vulnerability. The imports in programming languages (e.g. #include in C++, import in Java) allow developers to reuse service provided by other libraries. They successfully leveraged machine learning techniques to predict vulnerabilities with an average precision of 70% and recall of 45%. Walden et al. in [26] employed different code and design based software metrics as vulnerability predictors and used text mining techniques to build vulnerability prediction models for web applications written in PHP. In [27], the authors used bag-of-words representation considering a Java source file as a series of terms with associated frequencies. However, the reason why bag-of-approach method or the frequency of several terms are relevant to vulnerability has not been defined clearly. Gopalakrishna et al. [28] measured vulnerability likelihood based on four artifacts of computer programming, including Privileged Lines of Code, Error-prone Constructs, Programming mistakes and Program Style. They suggested to weight these metrics based on the extent of their correlations with vulnerabilities, but they did not identify how they could be weighted and how their threshold values would be determined. Nguyen et al. [29] introduced a new prediction model using dependency graphs of a software system based on the relationship among software elements such as components, classes, functions, and variables which can be obtained from static code analyzers.

These studies focused on identifying code-based metrics that quantify software security. In some works, the authors trained their model using machine learning approaches. In this work, we investigate the relationship between Java micro patterns and security vulnerabilities. This relationship has been studied from different viewpoints such as micro patterns' distribution for different types of vulnerabilities, associations among the micro patterns, and micro patterns evolution types in vulnerable versus non-vulnerable code. This analysis will lead us to develop effective security measures to assess software security. For example, if we identify micro patterns or their associations or their evolution types that relate to vulnerable code, we will be able to identify potentially vulnerable areas in the newly-implemented code. It may not be claimed as direct causation of vulnerability, but the resulting data can be used to assess security or predict the likelihood of vulnerability in code area based on micro pattern types.

4 Methodology

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

The research goal is to use micro patterns for detecting vulnerable classes so that vulnerability can be addressed early during coding. We chose micro patterns because they are defined on the Java class characteristics and can better represent a Java class. In order to accomplish this goal, we developed the following research questions to guide our work:

- **Research Question 1 (RQ1):** *Is there any significant difference in micro patterns' distributions between vulnerable and non-vulnerable classes?*
  This question will establish the relationship between micro patterns and vulnerabilities. We explore the distribution of micro patterns in both vulnerable and non-vulnerable code. This analysis will help us to determine if two distributions are significantly different from each other. The statistically-significant difference in the presence of micro patterns in vulnerable code and non-vulnerable code indicates their high correlation to the respective code.
- **Research Question 2 (RQ2):** *How are the micro patterns related to different types of vulnerabilities?*
  The answer to this question explores if the distributions of micro patterns vary with the types of vulnerabilities in the code. If we find statistically significant difference in micro patterns' distributions for different types of vulnerabilities including denial of service and information disclosure, we will be able to classify the vulnerability types based on the micro patterns' distributions.
- **Research Question 3 (RQ3):** *How do the micro patterns evolve from vulnerable classes to non-vulnerable classes?*
  This question serves to strengthen the relationships between patterns and vulnerabilities identified through RQ1. For example, if we see that micro pattern A is frequent in vulnerable classes and micro pattern B is frequent in non-vulnerable classes, and we get the evolution type $A \rightarrow B$ (i.e. vulnerable code is fixed by removing pattern A and including pattern B), this result strengthens the claim that pattern A is more vulnerability-prone than pattern B.
- **Research Question 4 (RQ4):** *How are the micro patterns associated with each other in vulnerable and non-vulnerable code?*
  To answer this research question, we examine the micro patterns data for vulnerable and non-vulnerable classes to get the phi-coefficient for each pair of micro patterns. In case of both the vulnerable and non-vulnerable classes,

we find the pairs of micro patterns where the association is high or medium. This analysis helps us to determine the joint effect of micro patterns on vulnerability generation. For example, if we see that $CompoundBox - Immutable$ is a highly associated pair in affected classes, whereas they have no association in non-affected classes, we can conclude that their joint occurrence has some effect on making the class vulnerable.

We formulate our Hypothesis $H0$ as follows:

$H0$: *Software vulnerabilities and micro patterns contained in their source code are not associated with each other.*

We will use our findings to prove or disprove this hypothesis using some statistical analysis.

4.1 Study Design

4.1.1 System Selection

For this study, we used Apache Tomcat as the subject system for our first case study. Apache Tomcat is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies and powers numerous large-scale, mission-critical web applications[8]. We used Apache Tomcat for two major reasons. First, all the vulnerability reports including the affected classes for every release are documented on the Apache website. Second, we needed a Java-based system along with vulnerability information because micro patterns are defined for Java classes, and this system adequately serves the purpose. We used Apache Tomcat Vulnerability Reports[9] to find the list of all vulnerabilities reported in the versions that we targeted for analysis. The software consists of about half a million lines of code and about 3000 classes. The vulnerabilities reported on the site are listed in Table 3. This table shows the reported vulnerability types across three major releases of Apache Tomcat. We considered a total 42 versions of Tomcat where the reported vulnerabilities are detected. We selected 39 versions from three releases because the vulnerabilities of Apache Tomcat are reported for these 39 versions on their website. In Figure 2, we see that the vulnerability affects 7.0.0- 7.0.52. We considered 7.0.52 as the last affected version for that vulnerability. By doing so, we collected the last affected version for all the reported vulnerabilities resulting in 39 total versions for the three releases. We then considered three additional versions as non-affected for the three releases to evaluate our results. In total, we considered 42 versions from Apache Tomcat. During this study, releases 6, 7, and 8 were the major releases of Tomcat and the previous releases such as 3, 4, 5 have few reported vulnerabilities. Although, in most of the cases, the versions are identical in terms of source code,

[8] https://tomcat.apache.org
[9] https://tomcat.apache.org/security.html

we are mainly interested in the classes where the vulnerabilities were identified and later fixed. As the vulnerabilities identified in different versions are different and affected different classes in different versions, our data is diverse. The source code is contained in the Apache Tomcat Archives[10], and it was used to download the source files across the versions specified. We extracted micro patterns from all the affected classes in the listed versions of releases 6, 7, and 8. Then, we extracted the micro patterns from the non-affected classes of three non-vulnerable versions in release 6, 7, and 8. In this case, we considered the last reported version of every release as the non-vulnerable version for that release. For example, 6.0.45 was the last non-vulnerable version in release 6, 7.0.69 in release 7, and 8.0.33 in release 8 at the time of this study. The reported vulnerabilities in each release do not exist in the last version of that release as they are already removed. Although the last version of a release may contain some other vulnerabilities as reported in the following releases, vulnerabilities reported for that release in earlier versions have been fixed and do not exist any more in the last reported version of that release. In our second case study, we conducted the experiment for three J2EE web applications: Pebble 1.6-beta1, Personalblog 1.2.6 and Roller 0.9.9 found on Stanford SecuriBench[11] which is a set of open source real-life programs to be used as a testing ground for static and dynamic security tools [35,36]. The size of all the systems under study has been presented in Table 2.

Table 2 Statistics of the Systems

	Systems/Releases	Version	Files/ Classes
Apache Tomcat	6	6.0.45	2000
	7	7.0.69	2800
	8	8.0.33	3000
Stanford	Pebble	1.6-beta1	333
	Personalblog	1.2.6	39
	Roller	0.9.9	276

4.1.2 Tool Selection

The micro patterns are extracted using a pattern extraction tool developed by Gil and Maman [1]. The command line tool is available at Maman's webpage[12]. The tool is interfaced via the command line and accepts the class name or a jar file as input. It then parses the class file or set of class files inside the jar one by one and matches the properties of the class based on the micro-patterns definitions as in Table 1. After that, the tool returns the micro patterns identified in that class or classes in the jar file to the caller.

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

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

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

Table 3 Vulnerabilities

Release 6	**Release 7**	**Release 8**	**Stanford Securibench**
Information disclosure	Security Manager bypass	Security Manager bypass	Cross-Site Scripting
Security Manager bypass	Request Smuggling	Request Smuggling	HTTP Response Splitting
Request Smuggling	Denial of Service	Denial of Service	SQL Injections
Information disclosure	Information Disclosure	Information Disclosure	Path Traversal
Frame injection in documentation Javadoc	Remote Code Execution		Log Forging.
Session fixation	Session fixation		
DIGEST authentication weakness	Bypass of CSRF prevention filter		
Denial of Service	DIGEST authentication weakness		
Bypass of security constraints	Bypass of security constraints		
Bypass of CSRF prevention filter	Privilege Escalation		
Authentication bypass and information disclosure	Multiple weaknesses in HTTP DIGEST authentication		
Multiple weaknesses in HTTP DIGEST authentication	Security constraint bypass		
Cross-site scripting	Remote Denial Of Service		
SecurityManager file permission bypass	Cross-site scripting		
Remote Denial Of Service and Information Disclosure	SecurityManager file permission bypass		
Information disclosure in authentication headers			
Arbitrary file deletion and			
or alteration on deploy			
Insecure partial deploy after failed undeploy			
Unexpected file deletion in work directory			

For running this tool, JDK 1.5 need to be installed. After extracting the micro patterns we analyzed the data for answering research questions RQ1, RQ2, and RQ3. The phi-coefficients were measured using the SPSS tool[13] for answering RQ4.

5 Case Study 1: Apache Tomcat

The steps we followed for collecting, extracting and analyzing data are presented in Figure 1.

5.1 Data Collection

We collected vulnerabilities from Apache Tomcat vulnerability reports that describe the vulnerabilities fixed in different versions. The vulnerability reports provide the information about the vulnerability type, its CVE id, affected versions, revision number, fixed version and severity level as shown in Figure 2. The reports also provide the list of classes that were modified for fixing the respective vulnerability as in Figure 3. If a vulnerability affects the versions 8.1, 8.2, 8.3 and is fixed in 8.5, we consider 8.3 as the last affected version and ignore the previous versions. In this way, we collect the last affected code versions for the listed vulnerabilities as shown in Table 4. In addition, the column "Affected Class Instances" in Table 4 is the total number of classes that have been modified to fix the vulnerabilities in all the revisions of the respective releases 6, 7 or 8. The column "Non-Affected version" contains the version where no vulnerability was reported for the releases 6, 7 or 8. Then we downloaded the code base for all different affected and non-affected versions listed in Table 4 and recorded the name of the classes which were modified to fix the vulnerabilities in respective versions. For example, as in Figure 2, the vulnerability Denial of Service (CVE-2014-0075) was fixed in revision 1578341 of version 7.0.53[14]. The last affected version is 7.0.52. If we follow the link

[13] http://www-01.ibm.com/software/analytics/spss/products/statistics/index.html
[14] https://tomcat.apache.org/security-7.html

to revision number[15], we will get the list of classes modified for fixing the vulnerability as shown in Figure 3. So our collected data is as follows:

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

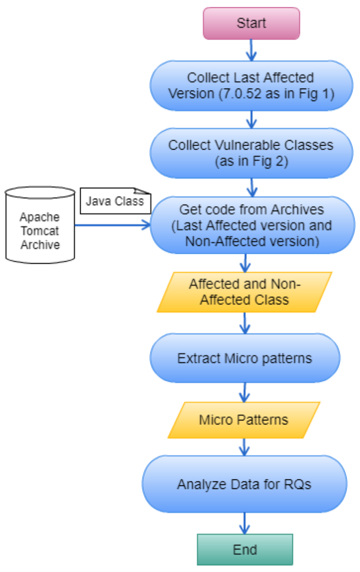
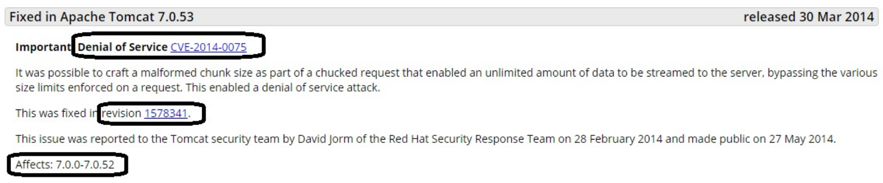

Start
Collect Last Affected Version (7.0.52 as in Fig 1)
Collect Vulnerable Classes (as in Fig 2)
Apache Tomcat Archive
Java Class
Get code from Archives (Last Affected version and Non-Affected version)
Affected and Non-Affected Class
Extract Micro patterns
Micro Patterns
Analyze Data for RQs
End

Fig. 1 Flow Chart of the Steps for Case Study 1


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.

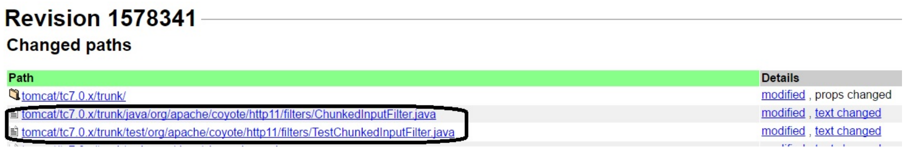
Affects: 7.0.0-7.0.52

Fig. 2 Apache Tomcat Security Page

5.2 Data Extraction

We downloaded the affected versions listed in Table 4 from the Apache Tomcat Archive. The micro pattern tool is interfaced via the command line and accepts

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


Revision 1578341

Changed paths

Path	Details
tomcat/tc7.0.x/trunk/	modified , props changed
tomcat/tc7.0.x/trunk/java/org/apache/coyote/http11/filters/ChunkedInputFilter.java	modified , text changed
tomcat/tc7.0.x/trunk/test/org/apache/coyote/http11/filters/TestChunkedInputFilter.java	modified , text changed

Fig. 3 Affected class names for a vulnerability.

Table 4 Affected and Non-Affected Versions and Classes of Apache Tomcat

Major Release	Affected Versions	Non-Affected Version	Affected Class Instances	Non-Affected Class Instances
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	104	2211
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	108	2789
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	55	2972

the class name or .jar file as input and extracts all micro patterns identified in that class or classes in the .jar file. If a particular micro pattern exists in that class, the entry is ‘1’; otherwise it is ‘0’. We collected the micro pattern data for 104 affected class instances in release 6, 108 in release 7, and 55 in release 8. We then analyzed micro patterns from the source code of all classes after the vulnerabilities were fixed. There were 2211 classes from version 6.0.45, 2789 classes from version 7.0.69, and 2972 classes from version 8.0.33.

5.3 Data Analysis

- **Research Question 1 (RQ1):** *Is there any significant difference in micro patterns’ distributions between vulnerable and non-vulnerable classes?*
  We have the micro patterns for all the affected classes of releases 6, 7, and 8. Then we calculated the percentage of each micro pattern in affected classes for 3 different releases. We also have the micro pattern data for the non-affected versions 6.0.45, 7.0.69, and 8.0.33. We calculated the percentage of each micro pattern in non-affected classes of 6.0.45, 7.0.69, and 8.0.33. We then illustrated the micro pattern frequency for affected vs non-affected classes using grouped vertical bar charts as in Figures 4, 5, and 6. Bar charts provide a visual presentation of categorical data. The height of the bars indicates the percentage of each type of pattern.
- **Research Question 2 (RQ2):** *How are the micro patterns related to different types of vulnerabilities?*
  For RQ2, we separated the classes affected by each type of vulnerability including denial of service and information disclosure. We then computed the percentage of each micro pattern in those classes. For example, we have three classes with the *Pool* micro pattern out of total 19 classes reported to contain the denial of service vulnerability for release 6. We present the

percentage of *Pool* micro pattern in denial of service vulnerability classes of release 6 based on this frequency.

– **Research Question 3 (RQ3):** *How do the micro patterns evolve from vulnerable classes to non-vulnerable classes?*
  We identified how the micro patterns evolve across the vulnerable versions to the non-vulnerable version. For example, test.java is an affected class of version 6.0.17. This class has micro pattern A. Now the vulnerability has been fixed, and the class test.java does not contain this micro pattern anymore. This evolution is denoted as $A \rightarrow None$. In this way, we detected pattern types in all the affected classes of releases 6, 7, and 8 separately and identified pattern types in the same classes after fixing their vulnerability in non-affected versions 6.0.45, 7.0.69, and 8.0.33. Then, we compared the micro patterns of the affected class of release 6, 7, and 8 with the micro patterns of the same class in versions 6.0.45, 7.0.69, and 8.0.33, respectively. This kind of analysis will help to strengthen the conclusion about vulnerability-proneness of the micro patterns related to the affected classes. We also identified the micro patterns' evolution across different releases. For example, a class has a micro pattern A in release 6, but the class no longer contains the pattern A in release 7. This change is not reported as a change for fixing a vulnerability, but it presents how micro patterns evolve over time in a software.
– **Research Question 4 (RQ4):** *How are the micro patterns associated with each other in vulnerable versus non-vulnerable code?*
  For answering this question, we measured phi-coefficient for each pair of micro patterns in affected classes of releases 6, 7, and 8 as well as classes of non-affected versions 6.0.45, 7.0.69, and 8.0.33. We found the high and medium association among the pairs by following the criteria in Section 2 and then presented the pairs that are not common in the affected and non-affected classes. If any pair is significantly associated in both the affected and non-affected classes, it will not contribute to finding its relation with vulnerability generation.

5.4 Results

In this section, we present the results obtained from the analysis of four research questions we described in Section 4.

5.4.1 Research Question 1 (RQ1)

Is there any significant difference in micro patterns' distributions between vulnerable and non-vulnerable classes?

We found that some micro patterns exist frequently in vulnerable classes whereas others are completely absent. In the same way, some micro patterns are frequent in non-vulnerable classes, and others are generally not present in them (for the target system we analyzed). The statistics of their frequency

Table 5 Micro patterns Frequencies in Affected versus Non-Affected Versions of Tomcat

Version 6		**Version 7**		**Version 8**	
Affected (104)	**Non-Affected (2211)**	**Affected (108)**	**Non-Affected (2789)**	**Affected (55)**	**Non-Affected (2972)**
Outline 1.9%	Outline 0.8%	Outline 10.2%	Outline 0.97%	Outline 9.09%	Outline 1.11%
Augmented Type 0.96%	Augmented Type 0.63%	Augmented Type 2.78%	Augmented Type 0.68%	Augmented Type 3.64%	Augmented Type 0.64%
Compound Box 1.9%	Compound Box 4.6%	Compound Box 1.85%	Compound Box 5.59%	Compound Box 1.82%	Compound Box 5.65%
Immutable 1.9%	Immutable 3.4%	Immutable 1.85%	Immutable 6.41%	Immutable 3.64%	Immutable 6.99%
Implementor 0.97%	Implementor 14.3%	Implementor 3.7%	Implementor 10.5%	Implementor 5.45%	Implementor 11.37%
Sink 4.8%	Sink 11.75%	Sink 5.56%	Sink 13.8%	Sink 5.45%	Sink 13.63%
Stateless 7.7%	Stateless 9.95%	Stateless 4.6%	Stateless 12.6%	Stateless 9.09%	Stateless 13.32%
Common State 4.8%	Common State 2.2%	Common State 0%	Common State 1.2%	Common State 0%	Common State 0.61%
Function Object 0.96%	Function Object 7.41%	Function Object 0%	Function Object 3.9%	Function Object 0%	Function Object 3.67%
Cobol Like 0.96%	Cobol Like 0.63%	Cobol Like 0.93%	Cobol Like 0.65%	Cobol Like 1.8%	Cobol Like 0.74%
Recursive 0.96%	Recursive 1.36%	Recursive 0%	Recursive 1.25%	Recursive 0%	Recursive 1.11%
Limited Self 14.4%	Limited Self 17%	Limited Self 6.48%	Limited Self 19.07%	Limited Self 9.09%	Limited Self 19.11%

in affected versus non-affected versions are presented in Table 5. The number of class instances considered for computing the percentage is presented in parenthesis.

Our significant observations are as follows:

– Affected classes in all versions of releases 6, 7, and 8 do not contain the patterns *Box*, *Canopy*, *RestrictedCreation*, *PureType*, *Extender*, *DataManager*, *Trait*, and *StateMachine* whereas they are present in non-affected classes as shown in Figure 4, 5, and 6. Therefore, we can recognize them as safe patterns to be used in code.
– There are some micro patterns that have statistically significant presence in non-affected classes compared to their presence in affected classes as shown in Figure 4, 5, and 6. They are *CompoundBox*, *Immutable*, *Implementor*, *Overrider*, *Sink*, *Stateless*, *FunctionObject*, and *LimitedSelf*.
– There are some micro-patterns which are significantly present in affected classes but are almost absent in non-affected classes. They are *Outline* and *AugmentedType*. The percentage of *Outline* pattern in affected versions of release 7 is 10.2% and in non-affected version of release 7 is 0.97%. The percentage of *Outline* pattern in affected versions of release 8 is 9.09%, and in non-affected version of release 8, it is 1.11%. The percentage of *AugmentedType* pattern in affected versions of release 7 is 2.78%, and in non-affected version of release 7, it is 0.68%. The percentage of *AugmentedType* pattern in affected versions of release 8 is 3.64%, and in non-affected version of release 8, it is 0.64%. *CommonState* pattern is available in version 6, but its uses have been reduced in the later versions. The percentage of *CobolLike* pattern is higher in affected versions than in non-affected version. The statistics of their frequency in affected versus non-affected versions are presented in Table 5.
– Other micro patterns such as *Useless*, *Sampler*, *PseudoClass*, *Joiner*, *Designator*, *Record*, *Taxonomy*, and *Recursive* are almost absent in both the affected and non-affected classes in all the versions of the target system.

We performed a chi-square test for measuring the statistical significance of our findings. We formulated our Hypothesis as $H0$: *Software vulnerabilities and micro patterns contained in their source code are not associated with each other* at $\alpha = 0.05$. We have tested this hypothesis for all the micro patterns considered in this study. We have reported the chi-square values for all of them that we found using SPSS tool in Table 6. In our case, degrees of freedom is

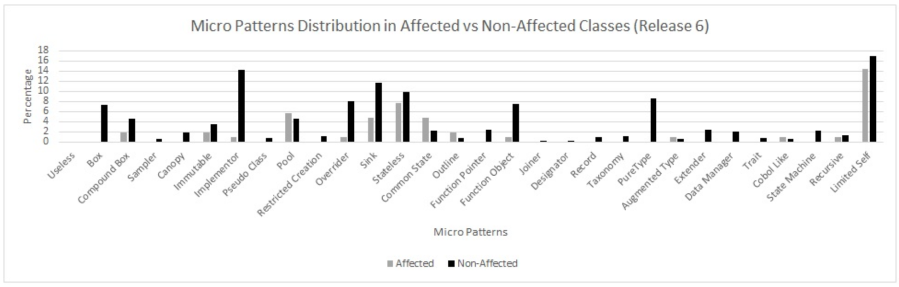


Fig. 4 Micro Patterns Distribution in Affected vs Non-Affected Classes of Tomcat (Release 6).

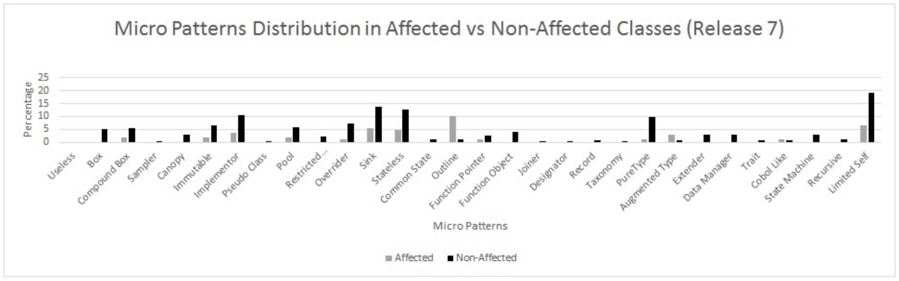


Fig. 5 Micro Patterns Distribution in Affected vs Non-Affected Classes of Tomcat (Release 7).

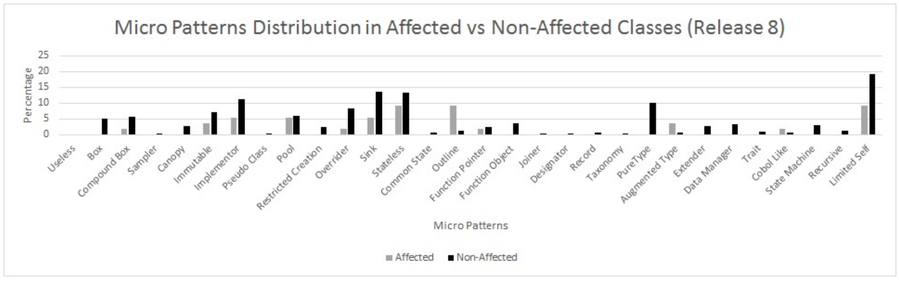


Fig. 6 Micro Patterns Distribution in Affected vs Non-Affected Classes of Tomcat (Release 8).

1. For degrees of freedom=1 and at 5% level of significance, the appropriate critical value is 3.84, and the decision rule is: *Reject* $H0$ *if* $\chi^2 \geq 3.84$. Therefore, we reject $H0$ for the micro patterns whose chi-square values are greater than 3.84 according to Table 6. In other words, we find statistically significant associations between Tomcat vulnerabilities and these micro patterns at $\alpha = 0.05$ ($p \leq 0.005$). On the other hand, there are some micro patterns such as $Sampler$, $PseudoClass$, $Pool$, $CommonState$, $FunctionPointer$, $Joiner$, $Designator$, $Record$, $Taxonomy$, $Trait$, $CobolLike$, and $Recursive$ for which we cannot reject the null hypothesis as for them $\chi^2 \leq 3.84$. It means we are

not evident enough at $\alpha = 0.05$ to claim that Tomcat vulnerabilities and these micro patterns are dependent where $p \leq 0.005$. We also provide Welch's Test result in order to support our results as shown in Table 7. Welch's Test for Unequal Variances (also called Welch's t-test, Welch's adjusted T or unequal variances t-test) is a modification of a Student's t-test to determine if two sample means are significantly different[16]. We compared the distribution of micro patterns between vulnerable and neutral classes, and this test captures the significant difference in the distribution of micro patterns between two groups: vulnerable and neutral. In Table 7, we present the F-values of the test for which p-value is less than $\alpha = 0.05$, and thus we reject the null hypothesis in favor of the alternative hypothesis. This result shows that for these micro patterns, there is sufficient evidence at $\alpha = 0.05$ level that the average number found in vulnerable classes differs from the average number found in neutral classes. F-value is a ratio of two quantities that are expected to be roughly equal under the null hypothesis which produces an F-value of approximately 1.

Table 6 Chi Square values for micro patterns in Tomcat classes

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

Table 7 Significant micro patterns in Welch's Test

Micro Pattern (Release 6)	**F-value**	**Micro Pattern (Release 7)**	**F-value**	**Micro Pattern (Release 8)**	**F-value**
Implementor	120.864	Implementor	17.059	Overrider	10.863
Overrider	40.357	Overrider	18.966	Sink	6.328
Sink	9.837	Sink	10.820	Outline	4.179
FunctionObject	33.740	Stateless	4.478	LimitedSelf	5.929
		Outline	7.226		
		PureType	37.127		
		LimitedSelf	8.723		

5.4.2 Research Question 2 (RQ2)

How are the micro patterns related to different types of vulnerabilities?

When answering this research question, we considered all the vulnerabilities listed in Table 3. But we present the graph only for three major types

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

of vulnerabilities: Denial of Service, Information Disclosure, and Security Bypass as most of the reported vulnerabilities of Tomcat in all releases are of these types. For example, in release 7[17], Denial of Service is 23%, Information Disclosure is 30%, and Security Bypass is 23% of the 60 total vulnerabilities. Moreover, OWASP top 10 vulnerabilities[18] investigated on eight large datasets from firms that specialize in application security and released top 10 vulnerabilities in 2013. Most of those vulnerabilities are related to information disclosure and security bypassing. For example, A1-injection flaws, such as SQL, OS, XXE, and LDAP injection occur when attackers send untrusted data to a user as part of a command or query which can trick the user into executing unintended commands and making a way for the attacker to access data without proper authorization. This is a kind of attack which discloses security keys to attackers. The results are shown in Figure 7, 8, and 9 for Denial of Service, Information Disclosure, and Security Bypass respectively.

Our significant observations are as follows:

– The micro patterns *Outline* and *LimitedSelf* are present in all three types of vulnerabilities.
– The micro patterns more frequent in the classes reported for Denial of Service are *Implementor*, *Pool*, *Sink*, *Stateless*, and *Recursive*.
– The micro patterns more frequent in the classes reported for Information Disclosure are *CompundBox*, *Immutable*, *Pool*, *Overrider*, *Sink*, *Stateless*, *CommonState*, and *AugmentedType*.
– The micro patterns more frequent in the classes reported for Security Bypass are *CompundBox*, *Immutable*, *Stateless*, *FunctionPointer*, and *CobolLike*.

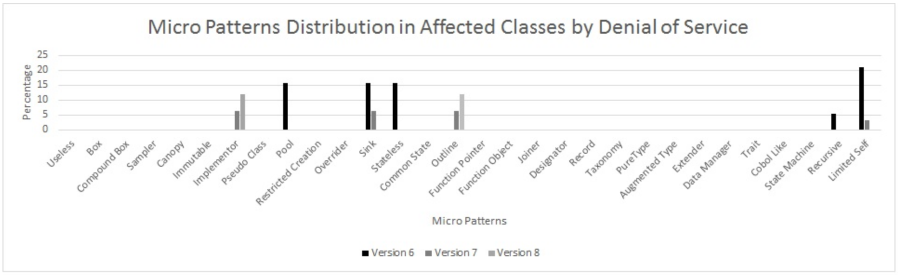


Fig. 7 Micro Patterns Distribution in Affected Classes by Denial of Service.

5.4.3 Research Question 3 (RQ3)

How do the micro patterns evolve from vulnerable classes to non-vulnerable classes?

[17] https://tomcat.apache.org/security-7.html
[18] https://www.owasp.org/index.php/Top_10_2013-Top_10

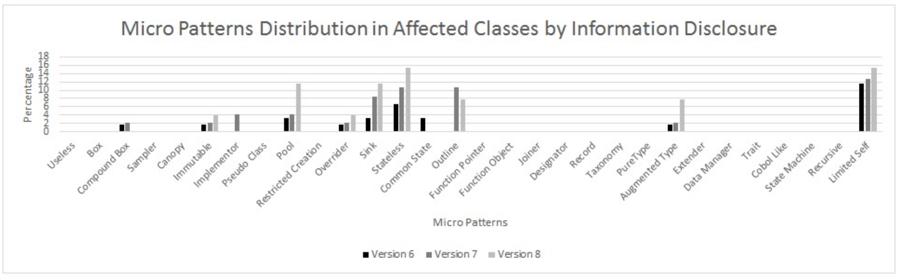


Fig. 8 Micro Patterns Distribution in Affected Classes by Information Disclosure.

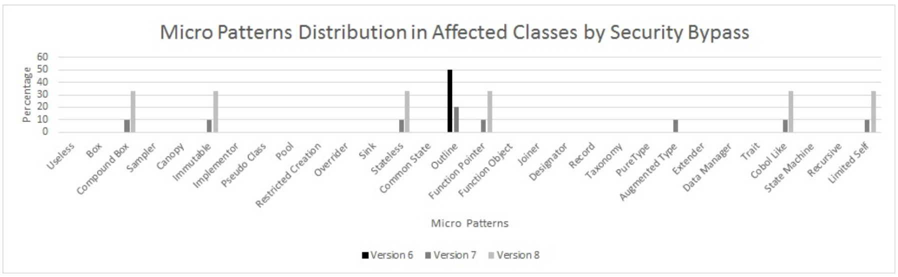


Fig. 9 Micro Patterns Distribution in Affected Classes by Security Bypass.

To answer this question, we analyzed all the affected classes of a particular release that contained vulnerable code, and then, analyzed those classes in the non-affected version of the same release, where the vulnerability had been fixed. We also determined how the micro patterns changed across the releases and identified several evolution types. For example, in Figure 10, we see that test.java has been evolving across versions 6.0.1 through 6.0.5. The vulnerability was active in version 6.0.2 and 6.0.3. So we consider 6.0.3 as the last affected version. The vulnerability has been fixed in version 6.0.4, and after that, all the released versions have been considered as non-affected versions for that vulnerability. So, in this example, version 6.0.3 is the last affected version and any version after 6.0.4 can be considered as non-affected.

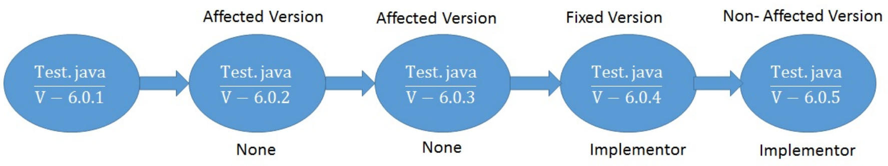


Fig. 10 Example of micro pattern evolution across affected to non-affected version.

If we focus on the evolution of the micro pattern in this example, we see that micro pattern *None* has been changed to *Implementor* as soon as the vulnerability has been fixed. So the micro pattern type of test.java has been evolved from *None* to *Implementor* (*None* → *Implementor*) once its vulnerability has been fixed and it has transitioned from affected to a non-affected class. In other words, we can say that micro pattern *Implementor* was introduced once the vulnerability was removed. The micro patterns evolution types found in releases 6, 7, and 8 are listed in Table 8. We also noticed that in many cases, the micro patterns do not change across the affected to the non-affected version. When a class is generally modified to fix a vulnerability, the pattern type remains same. In those cases, the vulnerability is fixed without changing the pattern. Therefore, the findings of our study does not encourage avoiding the use of some micro patterns in code, rather it helps the developers to use them carefully such that they do not make the program vulnerable. We also detected how the micro patterns evolve across the releases with time. The types of micro patterns evolution across the releases have been listed in Table 9. The frequency is computed with respect to the total number of evolution types found in the entire system of Apache Tomcat. From this Table, we see that some evolution types such as *None* → *Implementor*, *Implementor* → *None*, *CommonState* → *Stateless* are more frequent compared to other types on evolution.

Table 8 Micro patterns evolution types from Affected to Non-Affected versions of Tomcat

Micro Pattern Evolution
Recursive → *None*
AugmentedType → *None*
None → *Sink*

Table 9 Micro patterns evolution types across the releases of Tomcat

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

5.4.4 Research Question 4 (RQ4)

How are the micro patterns associated with each other in vulnerable and non-vulnerable code?

We computed the phi-coefficient for each pair of micro patterns in both the affected and non-affected versions of releases 6, 7, and 8 [34]. We present

Table 10 Micro patterns association types in release 6 of Tomcat

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

the connected pairs of micro patterns of affected and non-affected versions of release 6, 7, and 8 in Table 10, 11, and 12 respectively. Here we present the associations that are different in affected versus non-affected versions in respective releases. According to Table 10, Compound Box and Immutable patterns are highly correlated (phi-coeff: 1) to each other in affected classes which indicates that they coexist together in most of the affected classes of release 6. Similarly, Implementor and Function Object patterns are highly related in non-affected classes. They exist together in most non-affected classes of release 6. The pairings are significant because when they are used in practice, analysis tools can alert developers to their paired existence and respond accordingly with the appropriate review and tests of the constructs to determine if they are vulnerable.

Table 11 Micro patterns association types in release 7 of Tomcat

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

Table 12 Micro patterns association types in release 8 of Tomcat

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

6 Case Study 2: Stanford SecuriBench

In this case study, we conducted our experiment on some Java based web applications that contain vulnerable classes. Stanford SecuriBench[19] is a set of

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

open source programs to be used as a testing ground for static and dynamic security tools [35, 36]. Release .91a of Stanford SecuriBench focuses on Web-based applications written in Java. We downloaded the source code from the Stanford Securibench download site. The package contains a set of vulnerable J2EE web applications including jboard 0.30, blueblog 1.0, webgoat 0.9, blojsom 1.9.6, personalblog 1.2.6, snipsnap 1.0-BETA-1, road2hibernate 2.1.4, pebble 1.6-beta1, and roller 0.9.9. Among them, we selected pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9 for our study as these three applications provide war file as needed by the micro pattern extraction tool. The statistics of these three applications are given in Table 2. The steps we followed for collecting, extracting and analyzing data are presented in Figure 11.

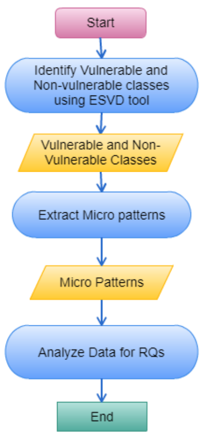

Start
Identify Vulnerable and Non-vulnerable classes using ESVD tool
Vulnerable and Non-Vulnerable Classes
Extract Micro patterns
Micro Patterns
Analyze Data for RQs
End

Fig. 11 Flow Chart of the Steps for Case Study 2

6.1 Data Collection

We used a static analyzer tool called Early Security Vulnerability Detector - ESVD[20] for identifying the vulnerable classes of the three projects: pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9. The reason behind using ESVD for finding the vulnerabilities instead of other tools was that it was shown to have less false positives in its results with higher precision and recall for the projects of Stanford Securibench[21]. The tool has Eclipse plugin that we installed in the Eclipse editor to analyze the projects. We got 23 vulnerabilities in pebble 1.6-beta1, 34 vulnerabilities in personalblog 1.2.6, and 207 vulnerabilities in roller 0.9.9. The vulnerability types were Cross-Site Scripting, HTTP Response Splitting, SQL Injections, Path Traversal, and Log Forging. We then explored

[20] https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/
[21] http://docplayer.net/1619013-Early-vulnerability-detection-for-supporting-secure-programming.html

the classes that are associated to these types of vulnerabilities. To that end, ESVD detected in total, 12, 7, and 72 vulnerable classes in pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9, respectively, as shown in Table 13.

Table 13 Class Statistics of Stanford Securibench

Version	**Affected Class Instances**	**Non-Affected Class Instances**
personalblog 1.2.6	7	30
pebble 1.6-beta1	12	321
roller 0.9.9	72	200

6.2 Data Extraction and Analysis

Using the micro pattern extraction tool, we extracted the micro patterns of all the vulnerable classes detected by Early Security Vulnerability Detector - ESVD in these three web applications. After that, we analyzed the percentage of each type of micro patterns in the vulnerable classes and also found the associations in each pair of micro patterns using SPSS. We got 91 classes identified as vulnerable by ESVD. Therefore, we considered the remaining 551 classes as non-vulnerable and extracted micro patterns from them. We then compared the micro patterns distribution in vulnerable or affected vs non-vulnerable or non-affected classes as we did for our first case study.

6.3 Results

In this section, we present the results obtained from the analysis for two research questions (RQ1 and RQ4) in pebble 1.6-beta1, personalblog 1.2.6, and roller 0.9.9. We did not perform any analysis related to RQ2 and RQ3 in this case study due to two major reasons. First, the types of vulnerabilities detected by ESVD in these applications are not same as in Apache Tomcat. Second, we did not have the versions of these applications after fixing the vulnerabilities for detecting micro patterns evolution from affected to non-affected classes.

6.3.1 Research Question 1 (RQ1)

Is there any significant difference in micro patterns' distributions between vulnerable and non-vulnerable classes?

After analyzing the affected classes of the three web applications, we found *LimitedSelf*, *Implementor*, *Overrider*, *Sink*, *Stateless*, *FunctionObject* as the most frequent patterns in the non-affected classes compared to the affected classes (cf. Figure 12). To obtain the statistical significance of our findings, we performed a chi-square test and had statistically significant evidence at $\alpha = 0.05$ to show that vulnerabilities present in the web applications, and these micro patterns are not independent (i.e. they are dependent or related in some way) where $p \leq 0.005$. We have reported the chi-square values that are greater than 3.84 in Table 14. In our case, degrees of freedom is 1. For $LimitedSelf$, $Implementor$, $Sink$, $Stateless$, $FunctionObject$ micro patterns, we can say that its presence in non-affected methods is significantly different than their presence in affected methods. We provide the micro patterns that have significant Welch's test values in Table 15. We compared the distribution of micro patterns between vulnerable and neutral classes, and this test can capture the significant difference in the distribution of micro patterns between vulnerable and neutral groups. In Table 15, we presented the F-values of the test for which p-value is less than $\alpha = 0.05$, and thus we reject the null hypothesis.

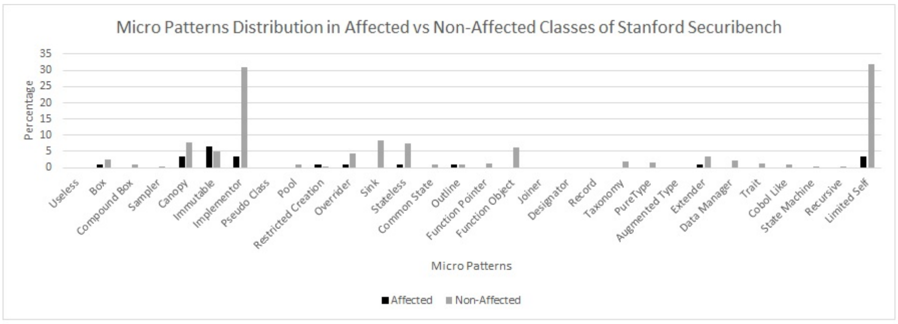


Fig. 12 Micro Patterns Distribution in Affected vs Non-Affected Classes of Stanford Securibench.

Table 14 Chi Square values for micro patterns in Stanford Securibench web applications

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

Table 15 Welch's Test values for micro patterns in Stanford Securibench classes

Micro Pattern	**F-value**
Implementor	65.639
Stateless	10.886
LimitedSelf	97.399

Table 16 Micro patterns association types in Stanford Securibench web applications

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

6.3.2 Research Question 4 (RQ4)

How are the micro patterns associated with each other in vulnerable and non-vulnerable code?

We present the connected pairs of micro patterns in the affected vs non-affected classes of three web applications in Table 16. The results for three applications are combined together in the table. According to Table 16, Outline and Extender are highly associated patterns in affected classes (phi-coeff: 1) whereas Canopy and Function Object are highly associated in non-affected classes (phi-coeff: .772). All other related pairs are listed in the table.

7 Discussion

7.1 Research Question 1 (RQ1):

Is there any significant difference in micro patterns' distributions between vulnerable and non-vulnerable classes?

We found *Outline* and *AugmentedType* micro patterns to be more frequent in vulnerable classes than they are in non-vulnerable classes of Apache Tomcat. Gil et al. [1] defined an *Outline* pattern as an abstract class where two or more declared methods invoke at least one abstract method of the current ("this") object. On the other hand, a class, having only abstract methods and three or more static final fields of the same type, is known as *AugmentedType* [1]. As an abstract class can not be instantiated and it is only for other classes to extend, the abstract classes with abstract methods need to be used carefully such that other classes can ensure their secured use. Kim et al. in [4] also observed *Outline* micro pattern as defective pattern as they found high bug rates in a class having this pattern. According to Destefanis [3], *"The Cobol Like anti pattern is a class having a single static method, one or more static variables, and no instance methods or fields. Cobol Like classes do not declare any method or instance field. Classes of this kind are very far from the Object Oriented programming paradigm and should be very rare."* This statement supports our result as *CobolLike* pattern is also not declared as a safe pattern in our analysis. Destefanis in [3], declared five micro patterns *Pool*, *CobolLike*, *Record*, *PseudoClass*, and *FunctionPointer* as anti-patterns as they are associated to bad programming practices. In our study, they are also not present in the list of safer micro patterns that are significantly frequent in non-affected classes compared to the affected classes.

Another observation was *Containment* category patterns such as *Box*, *CompundBox*, *Canopy*, *DataManager* are almost absent in vulnerable classes of Tomcat. *Box*, *CompundBox*, *Canopy* patterns are the classes that wrap a central instance field with their methods. The main purpose of *DataManager* and *Sink* patterns is related to the management of data stored in a set of instance variables. The *DataManager* pattern is a set of setter and getter methods which encapsulate all its fields and control the access to these fields [1]. This is an example of good object-oriented programming practice that ensures encapsulation. The presence of the patterns in the *inheritors* category such as *Implementor*, *Overrider*, and *Extender* in non-affected classes is more significant than their presence in the affected classes of Tomcat. Kim et al. in [4] found the evolution type *Implementor* $\rightarrow$ *None* as bug prone. It indicates that the removal of *Implementor* pattern makes a class defective which supports our claim that *Implementor* micro pattern is more frequent in non-vulnerable classes.

On the other hand, the results of three web applications in Stanford Securibench indicate that several micro patterns such as *LimitedSelf*, *Implementor*, *Overrider*, *Sink*, *Stateless*, *FunctionObject* frequently exist in non-affected classes. This finding is similar to the findings from Apache Tomcat, as they are also significantly frequent in Tomcat's non-affected classes.

The hypothesis testing validated our findings that several micro patterns are significantly associated with vulnerable code. Some patterns make the code more reliable whereas some of them reduce the strength of code resulting in less reliability.

7.2 Research Question 2 (RQ2):

How micro patterns are related to different types of vulnerabilities?
According to the Figures 7, 8, and 9, *CompundBox*, *Immutable*, *Stateless*, and *Sink* are present in three types of vulnerabilities. We cannot relate them to these types of vulnerabilities as they are frequent in all classes and their frequency is significantly higher in non-vulnerable classes which are not presented in these figures. On the other hand, the association of *Pool* and *AugmentedType* with information disclosure as well as the association of *CobolLike* and *FuntionPointer* with security bypass vulnerability need to be investigated more in our later study.

7.3 Research Question 3 (RQ3):

How do the micro patterns evolve from vulnerable classes to non-vulnerable classes?
In Table 9, we see that micro patterns do not usually evolve across different releases. Kim et al. also claimed that the micro patterns of Java classes do not change often [4]. Since micro patterns are at the class level, developers typically are not required to modify the class patterns to correct a vulnerability.

Developers usually fix the vulnerability in other ways which may not affect the micro patterns found in the class. The goal of our study is to present how the presence of certain micro patterns impacts the likelihood of an issue. For example, classes with abstract methods need to be more carefully implemented for avoiding any vulnerability. But it does not mean that abstract methods will be removed to fix a vulnerability. The evolution of micro patterns from vulnerable classes to non-vulnerable classes implies that these defective patterns have been carefully investigated in later versions and after investigation, some micro patterns were changed and many micro patterns remained unchanged. In addition, we studied their evolution across releases that supports the claim by Kim et al. in their paper [4]. Two evolution types $None \rightarrow Implementor$ and *CommonState* $\rightarrow$ *Stateless* show that $Implementor$ and *Stateless* micro patterns are more frequent in later releases. So using these micro patterns in classes may improve the code quality by fixing issues. We observe that *CommonState* pattern is frequent in Apache Tomcat release 6 and its uses have been abruptly reduced in later versions. We can describe this scenario in a way that many *CommonState* patterns have been converted to *Stateless* patterns in later releases, resulting in the frequent presence of *Stateless* pattern in later releases. *CommonState* pattern has been detected as bug-prone pattern in [4]. $None \rightarrow$ *Sink* type of evolution also supports that the pattern *Sink* is more prevalent in later releases.

We found that *AugmentedType* patterns are more frequent in vulnerable classes than they are in non-vulnerable classes. According to Table 8, the evolution type *AugmentedType* $\rightarrow$ $None$ from affected to non-affected version declares that *AugmentedType* patterns have been removed from classes to make them non-vulnerable. Therefore, the affected classes having *AugmentedType* pattern in releases 6, 7, and 8 are not containing this pattern in 6.0.45, 7.0.69, and 8.0.33 respectively. Similarly, $None \rightarrow$ *Sink* type of evolution also supports that the pattern *Sink* is more prevalent in non-vulnerable classes than they are in vulnerable classes. The evolution of the micro patterns across different releases as well as affected to non-affected classes strengthens our claim about micro patterns involvement in making a class vulnerable.

7.4 Research Question 4 (RQ4):

How are the micro patterns associated with each other in vulnerable and non-vulnerable code?

If we analyze the associated pairs in affected versions of Tomcat release 6, 7, and 8, we see that *CompundBox* $-$ $Immutable - Implementor$, *Pool* $-$ *Sink* $-$ *Stateless*, and *Pool* $-$ *Sink* $-$ $LimitedSelf$ are strong triangular relations of the micro patterns. Therefore, the developers can be more careful about the existence of these micro patterns' triangles in code. If any two of the three micro patterns in each group are present, the class needs to be tested rigorously. On the other hand, some associations such as *Sink* $-$ $DataManager$, *Stateless* $-$ $FunctionPointer$, and $Implementor - FunctionObject$ are found in the non-affected classes of both the Apache Tomcat and Stanford Securibench web applications. It indicates that these associations among the micro patterns are safe and testers can avoid further testing if they see them together in code.

8 Threats to Validity

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

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

Internal Validity: This threat refers to the possibility of having unwanted or unanticipated relationships. We are not claiming causation, just relating software vulnerabilities with the presence of micro patterns. We do not claim that the suspected micro patterns are the cause of the vulnerability-proneness of the classes but we recommend rigorous testing of the classes containing those micro patterns. On the other hand, we encourage the use of micro patterns that seem to be more frequent in non-vulnerable classes, although it cannot be claimed that the classes with these micro patterns will never be susceptible to vulnerability. But we can at least declare that the classes with these patterns will not need rigorous testing compared to the classes with other micro patterns. In addition, for RQ3, we ignored the changes of micro patterns in other versions between the affected version and non-affected version which may have an impact on the results of the study.

9 Conclusion and Future Plan

In this study, we conducted experimental analysis on micro pattern data extracted from the affected and non-affected classes of 42 different versions of three major releases of Apache Tomcat and three web applications. The goal of this analysis is to find out any hidden relationship between micro patterns and security vulnerabilities so that we can build a vulnerability prediction model based on these micro patterns. Correlating micro patterns with vulnerabilities will guide developers to use them properly to avoid security breach in Java code. The testers will also be able to ensure time and cost effective testing as they would not need to check all classes. Rather they will concentrate on the classes having unsafe patterns. In addition, this analysis can later be used for developing a metric which can quantify security of code and be a key for vulnerability prediction in the future. We found some patterns to be more frequent in affected classes while some patterns are relatively more frequent in non-affected classes. Moreover, we conducted experiments on the relations between micro patterns and specific types of vulnerabilities. We found the micro pattern distributions vary with the vulnerability types. We did not figure out the logical reasoning behind the result in this study. In the future, we plan to investigate more in this direction and explore how to describe this result. We also found the micro pattern evolution across the releases as well as from affected version to non-affected version with the vulnerability fixing. The results we found on evolution types strengthen our claim on the relationship between micro pattern and vulnerability. In addition, we extracted the association between every two micro patterns in affected versus non-affected classes so that we can make assumption about their collaborative effect on making a class vulnerable or non-vulnerable.

In this study, we limited our case study to 42 different versions of Apache Tomcat and three Java-based web applications. The main reason behind this was that micro patterns are defined for Java and we were in need of a system which is Java based as well as provides information on vulnerable classes in different versions. We could not find any appropriate Java based system that has well-formed vulnerability information as well as classes affected by those vulnerabilities. Therefore, our experiments may suffer from some false positive and false negative results. For replications, we provide access to the software tools and statistical results which can be found at dataset[22]. In future, we plan to extend our study to more systems. We also aim to measure the micro patterns in numeric scale rather than binary scale in order to extend their use in software security as well as develop more micro patterns that can better represent a vulnerable class. We have a plan to build a metric consisting of traceable patterns (class-level or method-level) and traditional software metrics that will overcome the shortcomings of current metrics and predict vulnerability with less false positive and false negative rates.

References

1. 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.

[22] http://archive.soma-research.org/a/sqj-ks-2017-data.rar

2. 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.
3. G. Destefanis, *Assessing software quality by micro patterns detection*, PhD Thesis, University of Cagliari, 2012.
4. S. Kim, K. Pan, and E. Whitehead Jr., *Micro pattern evolution*, Proceedings of the International Workshop on Mining Software Repositories, pages 4046, 2006.
5. S. Maggioni and F. Arcelli, *Metrics-based detection of micro patterns*, Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics, p.39-46, Cape Town, South Africa, May 04-04, 2010.
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. F. Batarseh, *Java nano patterns: a set of reusable objects*, Proceedings of the 48th Annual Southeast Regional Conference, New York, NY, USA, 2010.
8. 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.
9. K. Z. Sultana, A. Deo, and B. J. Williams, *A Preliminary Study Examining Relationships between Nano-Patterns and Software Security Vulnerabilities*, Proceedings of the 40th IEEE Computer Society International Conference on Computers, Software and Applications, Atlanta, Georgia, USA, June 10-14, 2016.
10. K. Z. Sultana, A. Deo, and B. J. Williams, *Correlation Analysis among Java Nano-patterns and Software Vulnerabilities*, Proceedings of the 18th IEEE International Symposium on High Assurance Systems Engineering, Singapore, January 12-14, 2017.
11. 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.
12. Y. Shin and L. Williams, *An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics*, Proceedings of the International Symp. Empirical Software Eng. and Measurement, pp. 315-317, 2008.
13. 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.
14. Y. Shin and L. Williams, *Can traditional fault prediction models be used for vulnerability prediction?*, Empirical Software Engineering, vol. 18, no. 1, pp. 2559, December 2013.
15. Y. Shin and L. Williams, *Is complexity really the enemy of software security?*, Proceedings of the 4th ACM workshop on Quality of protection, Alexandria, Virginia, USA, October 27-27, 2008.
16. F. A. Fontana, B. Walter, and M. Zanoni, *Code smells and micro patterns correlations*, RefTest 2013 Workshop, co-located event with XP 2013 Conference, 2013.
17. S. Moshtari, A. Sami and M. Azimi, *Using complexity metrics to improve software security*, Computer Fraud & Security, vol. 5, pp. 817, 2013.
18. 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.
19. I. Chowdhury, and M. Zulkernine, *Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?*, Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, March 22-26, 2010.
20. I. Chowdhury, B. Chan, and Mohammad Zulkernine, *Security metrics for source code structures*, Proceedings of the fourth international workshop on Software engineering for secure systems, pp.57-64, Leipzig, Germany, May 17-18, 2008.
21. I. Chowdhury and M. Zulkernine, *Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities*, Journal of Systems Architecture, Volume 57, Issue 3, March 2011, Pages 294-313.
22. N. Nagappan and T. Ball, *Use of relative code churn measures to predict system defect density*, Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, May 15-21, 2005.

23. B. Alshammari, C. Fidge and D. Corney, *Security Metrics for Object-Oriented Class Designs*, Proceedings of the 2009 Ninth International Conference on Quality Software, p.11-20, August 24-25, 2009.
24. 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 (ICST '10), pp. 421-428, IEEE Computer Society, Washington, DC, USA, 2010.
25. S. Neuhaus, T. Zimmermann, C. Holler and A. Zeller, *Predicting vulnerable software components*, In Proceedings of CCS'07, pp. 529-540, October, 2007.
26. J. Walden, J. Stuckman and R. Scandariato, *Predicting vulnerable components: Software metrics vs text mining*, Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on. IEEE, pp. 23-33, 2014.
27. R. Scandariato, J. Walden, A. Hovsepyan and W. Joosen, *Predicting vulnerable software components via text mining*, IEEE Trans Softw Eng, vol. 40, no. 10, pp. 9931006, 2014.
28. R. Gopalakrishna, E. Spaord, and J. Vitek, *Vulnerability likelihood: A probabilistic approach to software assurance*, In CERIAS Tech Report 2005-06, 2005.
29. V. H. Nguyen and L. M. S. Tran, *Predicting vulnerable software components with dependency graphs*, in International Workshop on Security Measurements and Metrics (MetriSec), 2010.
30. J. Ekstrm, *The Phi-coefficient, the Tetrachoric Correlation Coefficient, and the Pearson-Yule Debate*, Department of Statistics, UCLA. Retrieved from https://escholarship.org/uc/item/7qp4604r.pdf
31. B. Smith and L. Williams, *On the Effective Use of Security Test Patterns*, Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, pp.108-117, June 20-22, 2012.
32. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.
33. IBM Corp. Released 2015. IBM SPSS Statistics for Windows, Version 23.0. Armonk, NY: IBM Corp.
34. H. Cramer, *Mathematical Methods of Statistics*, Princeton University Press, pp. 282, 1946.
35. V. B. Livshits, *Findings security errors in Java applications using lightweight static analysis*, Work-in-Progress Report, Annual Computer Security Applications Conference, Nov, 2004.
36. V. B. Livshits and M. S. Lam, *Finding security errors in Java programs with static analysis*, In Proceedings of the 14th Usenix Security Symposium, pp. 271-286, Aug, 2005.
37. N. Yoshioka, H. Washizaki, and K. Maruyama, *A survey on security patterns*, Progress in Informatics, Special issue: The future of software engineering for security and privacy, vol. 5, pp. 35-47, October, 2008.
38. M. G. Graff and K. R. Wyk, *Secure Coding: Principles and Practices*, Chapter 4: Implementation, pp. 99123, OReilly, 2003.
39. D. A. Wheeler, *Secure Programming for Linux and Unix HOWTO*, 1999. http://www.dwheeler.com/secure-programs/.
40. R. C. Seacord, *Secure Coding in C and C++*, Addison Wesley, 2006.
41. S. Oaks, *Java Security*, 2nd ed. Addison-Wesley, 2001.
42. M. Howard and D. LeBlanc, *Writing Secure Code*, Second Edition, Microsoft Press, 2002.
43. C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, *The Art of Software Security Testing*, Addison-Wesley, 2006.
44. M. Schumacher and U. Roedig, *Security engineering with patterns*, In 8th Conference on Pattern Languages of Programs, July 2001.
45. J, Yoder and J. Barcalow, *Architectural Patterns for Enabling Application Security*, PLoP, 1997.
46. M. Bunke, *Software-security patterns: degree of maturity*, Proceedings of the 20th European Conference on Pattern Languages of Programs, Kaufbeuren, Germany, July 08-12, 2015.
47. P. Ponde, S. Shirwaikar and C. Kreiner *An analytical study of security patterns*, Proceedings of the 21st European Conference on Pattern Languages of Programs, Kaufbeuren, Germany, July 06 - 10, 2016.

48. F. Camilo, A. Meneely and M. Nagappan *Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project*, 12th Working Conference on Mining Software Repositories, 2015.
49. S. T. Halkidis, A. Chatzigeorgiou and G. Stephanides *A qualitative analysis of software security patterns*, Computers and Security, vol. 25, Issue 5, pp. 379-392, July, 2006.
50. J. Bau, E. Bursztein, D. Gupta and J. Mitchell *State of the Art: Automated Black-Box Web Application Vulnerability Testing*, 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2010, pp. 332-345.