2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing (HPSC), and IEEE Intl Conference on Intelligent Data and Security (IDS)

On Relating Code Smells to Security Vulnerabilities

Abdulrahman Abu Elkhail
Computer Science.
Baylor University,
76706 Waco, TX, USA
abdulrahman_abuelkh1@baylor.edu

Tomas Cerny
Computer Science.
Baylor University,
76706 Waco, TX, USA
tomas_cerny@baylor.edu

***Abstract*— In recent years there has been an abundance of well-known software design problems that fall under a variety of different terms such as flaws or code smells. Nowadays software systems place considerable importance to security concerns related to code flaws that lead to software vulnerabilities. In this paper, we present a study to identify the relationship between code smells and vulnerabilities in software code. Our study provides information about the impact of code smells on software security and considers open source implementation of technologies under the Apache Tomcat software. The results show the relationship between the code smells and the security vulnerabilities. While most code smells, have a slight impact, one particular smell is identified to have a more considerable impact.**

Keywords—code smells, vulnerability, code analysis, apache tomcat

I. INTRODUCTION

Code smells in software may point toward a problem. They appear in weak design and bad implementation methods [1]. Those smells could weaken the development process as well as increase the threat of bugs and failures. There are many types of code smells and they can be classified as application-level smells, class-level smells, or method-level smells. In the recent years, code smells have been studied for various reasons such as their effect on software quality or the high threat due to code faults that are exploited by potential attacks [2]. These issues affect the customer confidence in these particular software packages. Even after the code is tested before release, not all faults are detected. That's mainly due to runtime behavior of software faults during the test phase. These code faults will expose the software to potential attacks; this is commonly known as exploiting a vulnerability. Most code faults or vulnerabilities in software packages can be discovered by users or developers but require careful analysis. Therefore, due to the importance of the software security [24-26] it is necessary to study the relations between code smells and software security.

This paper presents a study to recognize the effect of code smells on software code vulnerabilities, aiming to understand the importance of producing good software design which significantly affects the software security. In order to assess our study, we consider the Apache Tomcat software open source implementation of the Java Servlet, Java Server Pages, Java Expression Language, and Java Web Socket technologies [3] and perform code analysis do detect code smells. Specifically, we use the PMD tool [4].

The specific contributions of this paper are the study of open source software - Apache Tomcat which has the vulnerability history information available online. This provides a trusted source of information to base our study on. Next, the relationship between the code smells and software security where we found the correlation between the number of code smells and code vulnerabilities and validated the correlation on multiple versions of the Apache Tomcat software.

The rest of the paper is organized as follows. Section II highlights related work. Section III described the study setup. Section IV presents the performance evaluation experiments and discusses the achieved results. Section V concludes the paper and gives some future directions for research in this area.

II. LITERATURE REVIEW

Many researchers studied code smells either by using software analyzing tools or custom code. However, very limited considerations were made when it comes to the correlation between code smells and security.

Vulnerability dataset was proposed in [5] to detect code smells, bugs and faults in code files. In the study, 15 java projects from GitHub, the source code elements were matched with known fixed bugs and a set of product metrics was calculated based on these elements. The dataset was processed by 13 machine learning algorithms to predict bugs. The study results with high and promising bug coverage values, but more projects need to be considered to ensure correctness. Furthermore, the applied tool is not publicly accessible.

A study of the effect of code smell on system maintainability and readability [6] found that one smell has no effect; however, the effect is increased when more than one fault coexists. This has a negative impact on maintainability. It affects fault detection and correction and may make the system vulnerable in the future. However, more systems and anti-patterns need to be to conform this to further understand the problem significance.

In [7] the authors studied how refactoring can improve the security of an application by removing code smell. They filtered code smells against security attributes. Next, refactoring is applied to remove security-related code smells. The results show that refactoring helps improve the security of an application without compromising the overall quality of software systems. However, the paper does not look at some object-oriented properties like extensibility. The approach used is also not fully automated. Furthermore, more testing on a larger set of projects might help generalize the obtained results.

The authors in [8], examined seven open source software projects and studied the lifespan of code smells. It was found that code smells are removed due to other activities such as maintenance or development of new functionalities. This is

978-1-7281-0006-7/19/$31.00 ©2019 IEEE
DOI 10.1109/BigDataSecurity-HPSC-IDS.2019.00013

evidence that code smell can survive in different software product releases. However, more code smells need to be investigated to validate the results. Furthermore, the work could be improved by analyzing industrial software systems. Finally, statistical analysis to determine the correlation between smells and developers would be desirable.

Authors in [9] attempt to perform bug-prediction in smell classes based on the common opinion of the effect of smell on code maintenance which may lead to bugs. The papers main input was the use of code smell intensity with the prediction model. Smell detection involves many metrics such as statistical distribution on data sets and these metrics can be used to measure the intensity of code smell. However, the number of analyzed systems should be increased to verify the proposed method and the obtained results. The outcome and applicability of the work should be compared to other existing work.

Similarly, in [10] the authors used early system versions to extract information about the pattern "a design fault" follows when a system progresses. Design flaw evolution is studied through a developed model. It is suggested by the paper that this pattern could be employed to predict design faults and might be useful for future identification of faults. However, more systems need to be analyzed in order to categorize design flaw evolution.

Code smells and software flaw study [11] found a correlation between them. It was concluded that some code smells such as shotgun surgery, large class and large method [12] are highly correlated with software flaws. On the other hand, other code smells like data class, feature envy and refused bequest [12] had little correlation with software flaws. However, more software versions should be considered to generalize obtained results.

The authors of [12] tried to quantitatively clarify the relation between software reliability and maintainability and code clones of old software. It was concluded that the software system reliability and maintainability is reduced by duplicated code. However, the relation between code clones and software reliability and maintainability still needs more clarification. More quantitative analyses are required to verify the results.

To identify duplicated code, 11 duplicated code patterns were identified in [13]. To judge the authenticity of each pattern, domain experts were invited. They concluded that no refactoring was required for the majority of the identified patterns because they were harmless. However, more patterns of cloning need to be identified to have a realistic implementation of the approach. Furthermore, identifying where code cloning succeeds and fails in development is a good direction to expand the study.

A secure information systems development paradigm is presented in [14]. Security design patterns were discovered using an analytical process. In the first stage it focused on finding shared subjects and objects among software development and security development. The second stage involved acknowledging the security constraints, abuse scenarios and policies. The last stage showed the gathered proposed patterns from expert views. To improve the work, there is a need to conduct empirical research to check whether this approach applies to conventional software systems.

The specification and code smell detection at the class level were studied by the authors of [15]. The paper developed a detection technique to automatically execute the described method. Code smells were classified according to relations between metrics and some of them are Blob: controller class, controller method, large class, low cohesion; Functional decomposition: private field, a class with one method; Spaghetti code: no parameter, use a global variable. The approach used in this paper needs to be compared with similar work to provide more insight into the results.

Code smells evolution is investigated by the authors of [16]. They focus their research on studying three code smells and look at successive versions of two open source systems. They found out that design problems still exist even in the most recent version they examined. Only a minor number of code smells is corrected from the project and this was not due to the application of refactoring techniques. However, the paper does not look at the results obtained from tools that can detect applied refactoring. Furthermore, the paper does not perform enough statistical analysis to determine whether some code smells tend to be erased faster than others.

An automated formal vulnerability analysis approach [17] located possible matches in a system. It was tested on seven open source applications of different domains. It could be improved by applying a dynamic analysis based on formal signatures.

Clones and non-cloned code vulnerabilities categories are studied in [18]. The authors conduct an empirical study and examine 97 source codes of software systems. It was concluded that there is no significant difference in code clones and clone-free code vulnerabilities. However, the findings of this paper are all related to software written in Java and may not be generalizable to code written in other languages.

In [19] the study was to find the relationship between nano-patterns and vulnerable code. They apply data mining techniques to detect code vulnerabilities then apply hypothesis testing to validate the results found. A larger dataset should be analyzed including class level patterns to improve the findings.

The correlation between software metrics and vulnerabilities is studied by [20]. Software metrics were calculated from the Linux Kernel, Mozilla, Xen Hypervisor, HTTP, and Glibc projects code and their relation to vulnerabilities was identified through the application of correlation algorithms. Finally, the results indicate that vulnerable functions are probable to have other vulnerabilities in the future.

The relation between certain types of code smells and merge conflicts is studied by the authors of [21]. They recreate 6,979 merge conflicts by mining 143 repositories from GitHub. They categorize the conflicts based on their effects on the Abstract Syntax Tree. It was concluded that it is three times more likely for entities that suffer from code smells to be involved in merge conflicts. The limitation of the study is that it only focuses on GitHub source code programs. This can be alleviated by analyzing commercial programs as well.

The papers referenced above focus on the correlation between code smell and security vulnerabilities. We think that there is a research opportunity in this area. Thus, in this research, we are looking to find the relation between code smell and security vulnerabilities through studying a vulnerability dataset and applying code detection tools on the dataset.

III. THE STUDY SETUP

The main objective of our work is to study the effect of code smells on software security by employing code analyzer tools to detect code smells on a vulnerable software and find the relationship between the software vulnerabilities and code smells. The research will focus on several types of vulnerabilities. This is to limit the scope and conduct more focused research that can yield relevant results.

We use the Apache Tomcat datasets [16] vulnerabilities that are used by users and developers. The most important part is to find which source code files have security bugs or vulnerabilities. By doing this, we can link the number of code smells and the number of vulnerabilities in each file. There are many software quality tools that can be used to detect the code smells; a suitable tool was selected to collect the number of code smells. The PMD tool was chosen as it is open source tool commonly used to analyze code and detect flaws and smells.

Software dataset: We used the Apache Tomcat software (Tomcat) with Java Servlet, Java Server Pages, Java expression language and web socket open source technologies. It is developed in collaboration of the best-of-breed developers from around the world; next, it controls numerous large-scale, mission-critical web applications across a diverse range of industries and organizations [2].

Code analysis: PMD tool is used for code analysis. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, etc. It supports various languages, and includes a copy-paste-detector to find duplicated code [4]. We use four different vulnerable versions of the Tomcat [3] (7.0.10, 8.0.1, 8.5.0 and 9.0.0.M1) to detect code smells. PMD can be executed from the command line or can be integrated into other tools, such as Eclipse and JCreator. We use PMD from the command line aiming to extract code smells into data sheets rather than make use of the refactoring capability.

Hypothesis: In order to achieve our goal, we developed the following research questions to guide our work. (RQ1) How are the code smells related to a different number of vulnerabilities? (RQ2) How do code smells effect software vulnerabilities? We formulate our hypothesis Lemma1 as follows: Software vulnerabilities and code smells are not correlated with each other. We will use our findings to prove or disprove this hypothesis using some statistical analysis.

The Vulnerabilities List: The vulnerabilities list is obtained from the publicly available Tomcat Vulnerabilities Dataset [16]. One of the main reasons Tomcat was used in this study is the availability of the vulnerabilities list which enabled easier analysis. Each version's vulnerability list is documented and maintained at the Tomcat website. The original vulnerabilities list is large and includes low priority vulnerabilities. In this study, we focus on a smaller number of vulnerabilities that are of high importance. The lists of vulnerabilities that we are interested in researching are summarized as follow:

- Authentication Bypass: attempting to inject the remote host that allows for the bypassing of authentication.
- Cache Poisoning: attempting to corrupt data is inserted into the cache database of the Domain Name System (DNS) name server.
- Bypass of CSRF Prevention Filter: attempting to trick the victim into submitting a malicious request.
- Bypass of Security Constraints: attempting to add an URL path parameter to a request that allows bypassing a security constraint.
- Cross-site Scripting: attempting to inject client-side scripts into web pages viewed by other users.
- Information Disclosure: aiming to gain information about a system.
- Denial of Service: attempting to bring down a server or make it unavailable for normal users during a given period of time.
- Directory disclosure: attempting to inject the directory through links on the web site.
- CSRF Token Leak: attempting to perform an unwanted action on a trusted site for which the user is currently authenticated.
- DIGEST Authentication Weakness: attempting to negotiate credentials, such as username or password with a user's web browser.
- Remote Code Execution: attempting to access computing device and make changes, no matter where the device is geo-located.
- Limited Directory Traversal: attempting to access files and directories that are stored outside the web root folder.
- Privilege Escalation: attempting to access more resources or functionality than they are normally allowed.
- Multiple Weaknesses in HTTP DIGEST Authentication: attempting to negotiate credentials, such as username or password with a user's web browser.
- Security Constraint Annotations applied too late: attempting to expose resources to users who were not authorized to access them.
- Request Smuggling: attempting to smuggle a request to one device without the other device being aware of it.
- Security Manager Bypass: attempting to inject system across platforms (operating systems and browsers).
- Session Fixation: attempting to inject client-side scripts into web pages viewed by other users.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our study by presenting the code smells and security vulnerabilities as well as the relationship between them. We use the PMD tool on four different versions of the Tomcat.

We execute PMD from the command line and direct its results to a data sheet. PMD divides the detected code smells into different rulesets based on their nature. E.g., an object that controls too many other objects (a God Class) and containers for data used by other classes (a Data Class) fall under the design ruleset, the uncommented empty methods fall under the documentation ruleset. Next, we calculated the correlation coefficient for each smell. That's mainly we know that the correlation coefficient, which is denoted by r tells us how closely data in a scatterplot fall along a straight line. The closer that the absolute value of r is to one, the better that the data are described by a linear equation. If r =1 or r = -1 then the data set is perfectly aligned. Data sets with values of r close to zero show little to no straight-line relationship. Therefore, we calculated the correlation coefficient for each smell as follow [22]:

- The data we are working with are paired data, each pair of which will be denoted by (x_i,y_i) where x_i is the number of detected smells for each smell, where i =1,2,3. which indicates the priority level for the smell from low to important level. Where y_i is the number of vulnerabilities, where i =1,2,3. which indicates the priority level for the vulnerability from low to important level.
- Calculate $\bar{x}$, the mean of all of the first coordinates of the data x_i.

- Calculate $\bar{y}$, the mean of all of the second coordinates of the data y_i.
- Calculate $s\,x$ the sample standard deviation of all of the first coordinates of the data x_i.
- Calculate $s\,y$ the sample standard deviation of all of the second coordinates of the data y_i.
- Use the formula $(zx)_i = (x_i - \bar{x}) / s\,x$ and calculate a standardized value for each x_i.
- Use the formula $(zy)_i = (y_i - \bar{y}) / s\,y$ and calculate a standardized value for each y_i.
- Multiply corresponding standardized values: $(zx)_i*(zy)_i$
- Add the products from previous step together.
- Divide the sum from previous step by $n - 1$, where n is the total number of points in our set of paired data. The result of all of this is the correlation coefficient r.

Table I shows a simple example on correlation coefficient calculation r which is equal 0.327327 in this example.

TABLE I. EXAMPLE OF CORRELATION COEFFICIENT CALCULATION

X(Code smell)	Y(Vulnerability)	Priority
3	6	Low
5	5	Moderate
2	4	Important

A. Results

Fig. 1 shows the distribution of the number of smells recorded by each version. The reported smells still exist even when new versions of Tomcat are developed. This shows that smells persist through different versions of the software.

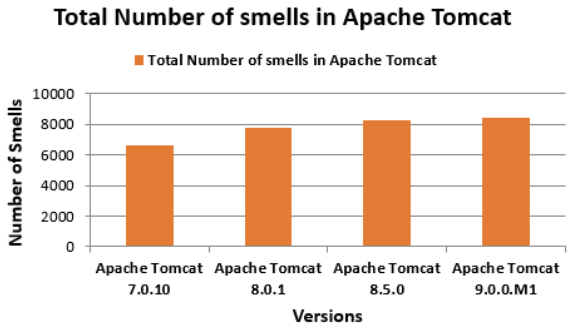

Fig.1. The total number of smells in Apache Tomcat of different versions.

TABLE II. PERCENT OF CODE SMELLS FOR RULE SET OF ALL VERSIONS.

Rule Set	Percent of Rule Set in Apache Tomcat			
	7.0.10	*8.0.1*	*8.5.0*	*9.0.0.M1*
Best Practices	29	39	41	41
Code Style	42	37	33	34
Design	18	12	11	11
Documentation	1	1	2	2
Error-prone	5	6	6	5
Multithreading	4	4	6	6
Performance	1	1	1	1

The percentage of the code smells for each rule set is shown in Table II. It can be observed that the best practices and code style rule sets have the highest percentages of the code smells in the four versions; the design rule set has the second highest number of smells. While the documentation and performance rule sets have the lowest rate of the smells in the four versions.

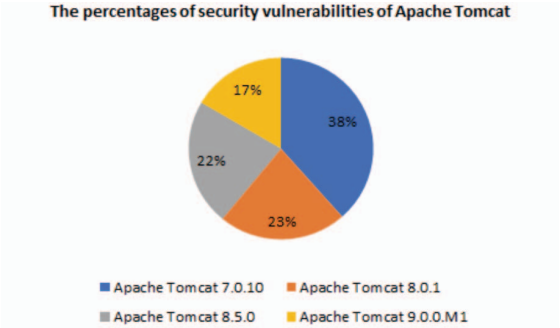

Fig.2. The Percentages of Vulnerabilities in each version.

Fig.2 illustrates the rates of vulnerabilities recorded by each version. It can be observed from Fig.2 that the reported vulnerabilities plummet when new versions of the software are developed which means new versions still have vulnerabilities but their occurrence decreases.

Some of the vulnerabilities are shared across the different versions. Once a vulnerability is found in one version, the chance it exists in another one becomes higher. With this in mind, the Tomcat team has been able to fix many vulnerabilities across many versions.

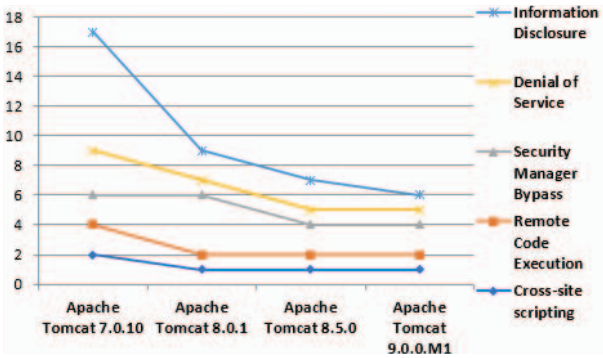

Fig.3. Vulnerabilities perversion.

Fig.3 shows the number of vulnerabilities identified in every version. It is clear that Tomcat 7.0.10 and 8.0.1 are the most vulnerable versions. However, the information disclosure and denial of service are of higher count than other vulnerabilities in all four versions. Some vulnerability such as cross-site scripting is of lower count than other vulnerabilities in all four versions.

Table III illustrates the correlation between the code smells and the security vulnerabilities of all four different versions. There is a small correlation between the code smells and the security vulnerabilities. However, the Field Declarations Not at Start of Class has the highest effect on software security. This because the Number of Field Declarations Not at Start of Class smell is about 25% or more of a total number of smells according to Fig.4. Furthermore, Fields should be declared at the top of the class, before any method declarations, constructors, initializers or inner classes [4]. Otherwise, it will cause wrong when dealing with in the code and this is known as a code smell. This code smell is known as Field Declaration Not Start At Class code smell. This code smell may slow down the development process and may increase the risk of bugs and failures. Thus, it will increase the risks of software security vulnerabilities.

TABLE III. THE CORRELATION BETWEEN THE CODE SMELLS AND THE SECURITY VULNERABILITIES OF FOUR DIFFERENT VERSIONS.

Code Smell	Correlation in Tomcat Versions			
	7.0.10	***8.0.1***	***8.5.0***	***9.0.0.M1***
Abstract Class Without Abstract Method	0.007	0.009	0.011	0.009
Accessor Class Generation	0.008	0.058	0.055	0.049
Accessor Method Generation	0.085	0.143	0.161	0.143
Deeply Nested If statements	0.071	0.077	0.067	0.059
Protected Field in Final Class	0.075	0.060	0.060	0.051
Protected Method in Final Class	0.013	0.018	0.019	0.012
Reassigning Parameters	0.026	0.024	0.022	0.020
Method Level Synchronization	0.027	0.025	0.031	0.034
Resource Left Unclosed	0.009	0.082	0.089	0.082
Constants in Interface	0.032	0.019	0.013	0.013
Constructor Calls Overridable Method	0.081	0.055	0.054	0.048
Data Class	0.014	0.012	0.013	0.011
Non-Abstract Empty Method in Abstr.Class	0.047	0.070	0.081	0.076
Field Declarations Not at Start of Class	***0.237***	***0.183***	***0.158***	***0.146***
God Class	0.014	0.019	0.012	0.011
Immutable Field	0.076	0.028	0.024	0.024
Complex Boolean Expressions	0.013	0.010	0.008	0.007
Complex Boolean Returns	0.083	0.047	0.046	0.042
Singleton Class Returning New Instance	0.007	0.006	0.006	0.005
Switch Density	0.009	0.003	0.003	0.003
Switch Statements Without Default	0.039	0.039	0.051	0.046
Too Few Branches for a Switch Statement	0.029	0.010	0.010	0.009
Uncommented Empty Constructor	0.047	0.036	0.041	0.039
Uncommented Empty Method Body	0.033	0.065	0.071	0.082
Unnecessary Local Variable Before Return	0.053	0.044	0.042	0.038

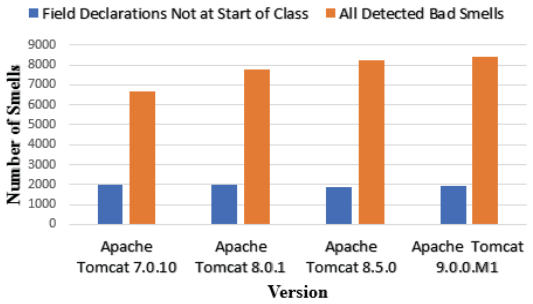

Fig.4. Number of 'not at start' declarations vs the total num. of smells

In order to avoid Field Declaration Not Start At Class code smell, we should put the field declarations like the following: • *static/instance variable field declaration* • *static/instance initializer* • *static/instance member inner class declarations* • *static/instance method declarations* • *instance constructor declarations*.

This ordering is very important because we cannot use a field in an initializer before it itself has been initialized. Such code is difficult to understand because it expects the field is declared at the start of class. In addition, there are well-identified strategies to improve code readability, code reuse, and cohesion.

Fig.5 show the impact of code smells on the different versions of Tomcat versions by vulnerability priority. The distribution remains consistent among all versions with a small drop in important vulnerabilities as the software matured. This is expected because as the software is being developed, more vulnerabilities are discovered and patched.

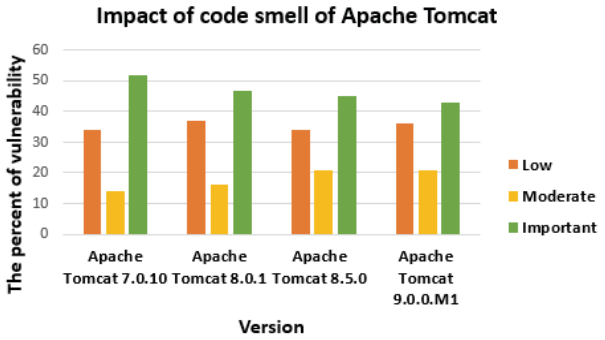

Fig.5. The impact of code smells of Apache Tomcat

In Fig.6, the total number of code smells in each vulnerable version and the corresponding fixed version are shown. It can be observed that in terms of numbers, code smells are higher in most cases rather than falling down. This can be attributed to the fact that when fixing a certain vulnerability, more code is added to address the issues found. To address this issue, more versions should be studied incrementally in order to verify the results.

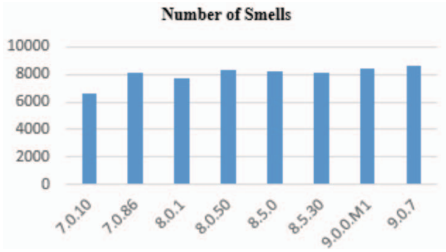

Fig.6 The number of smells found in Apache Tomcat versions.

B. *Discussion of research questions*

To answer (RQ1) on *how are the code smells related to a different number of vulnerabilities* we discover whether the dissemination of code smells are varying with the number of security vulnerabilities in each version of Tomcat. We look to find a statistically significant difference in code smell spreading and the number of vulnerabilities for each version of Tomcat that indicates high correlation to each other. According to Figs.1-2, the dissemination of code smells are varying with the number of security vulnerabilities in each version of Tomcat. On the other hand, Fig.3 shows the number of vulnerabilities recognized in each version. It is clear that Tomcat 7.0.10 and 8.0.1 are the most vulnerable versions compared to other two.

To answer (RQ2) on *how do code smells effect software vulnerabilities* we computed the correlation-coefficient r for each code smell in Tomcat dataset of versions 7.0.10, 8.0.1, 8.5.0 and 9.0.0.M1. According to Table 2, there is a small correlation between the code smells and the security vulnerabilities. However, Field Declarations Not at Start of Class code smell is highly correlated to the security vulnerabilities which indicates that they coexist together in most of the affected classes. On the other hand, Figures 4-7 show the impact of code smells on the different tested Tomcat versions by vulnerability priority. The distribution remains consistent among all versions with a small drop in important vulnerabilities as the software matured. This is expected because as the software is being developed, more vulnerabilities are discovered and patched. Also, we may not know all the vulnerabilities of

the version 10 simply it is less mature, and we only base on the data we have that are known, there was a long time to discover issues for version 7 compared to version 10.

C. Validity Threats

The study shows the relationship between the code smells and the security vulnerabilities. Next, we discuss the study outcome and generalization of results through threats to validity. **Internal validity**: To test the impact of code smells on security vulnerabilities we used an open source implementation of Tomcat technologies. We obtained the vulnerabilities list from the openly available Vulnerabilities Dataset. In our study, we do not have a complete list of vulnerabilities e.g. overflow, Gain information, execute code, etc. We focus only on a smaller number of vulnerabilities of high meaning. Our study is limited to vulnerabilities that are known so far and reported. To identify the code smells, similar to [23], the PMD tool was used as code analyzer. To minimize measurement errors, we measured the code smells twice and cross-validated our results. **External validity**: Our study considers one representative software of a real-world application. The used Tomcat controls abundant large-scale, mission-critical web applications across a diverse range of industries and organizations, moreover, it is open source. This representative software does not reflect all aspects of the software applications or all conventional development styles or approaches. To avoid single application version perspective we considered four versions. This case study serves as a demonstration of code smell impact on vulnerability..

V. CONCLUSION

In this paper, we performed an empirical study to test the relationship between code smells and security vulnerabilities. We used the Tomcat software for the case study because it is an open source and it has a well-defined vulnerability history available. We used static code analysis to identify code smells for the Tomcat source files. The results show the relationship between particular code smell and the security vulnerabilities. While most code smells, have a slight impact, one particular smell is identified to have a more considerable impact. The most notable correlation is with “Field Declarations Not at Start of Class”. For future work, we aim to conduct a study with more open source projects to examine and validate the results of this study. In particular complied code or byte code analysis may yield with other kinds of smells.

REFERENCES

[1] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference, Florence, Italy, 2015.

[2] Black, P. E., & Fong, E. (2016). Report of the Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV). NIST Special Publication, 500, 320.

[3] Apache Software Foundation, "Apache Tomcat," [Online]. Available: http://tomcat.apache.org/.

[4] "PMD Source Code Analyzer," 2018. [Online]. Available: https://pmd.github.io/. [Accessed 2018].

[5] Z. Tóth, P. Gyimesi, and R. Ferenc, "A Public Bug Database of GitHub Projects and Its Application in Bug Prediction," in Intl. Conference on Computational Science and Its Applications, Springer, Cham, 2016.

[6] M. Abbes, F. Khomh, Y.-G. Gueheneuc and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference, Oldenburg, Germany, 2011.

[7] H. Mumtaz, M. Alshayeb, S. Mahmood and M. Niazi, "An empirical study to improve software security through the application of code refactoring," Information and Software Technology, pp. 112-125, 2018.

[8] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells using Software Repository Mining," in Software Maintenance and Reengineering (CSMR), Szeged, Hungary, 2012.

[9] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia and R. Oliveto, Smells like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells., In Proceedings of the 32nd International Conference on Software, 2016.

[10] G. a. R. M. Ganea, Modeling Design Flaw Evolution Using Complex Systems, In 17th Intl. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 433-436. IEEE, 2015.

[11] W. Li and R. Shatnawi, "An investigation of bad smells in object-oriented design," in Third International Conference on Information Technology: New Generations, 2006.

[12] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato and K.-i. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software," in Eighth IEEE Symposium on Software Metrics, 2002.

[13] C. J. Kapser and M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software," Empirical Software Engineering 13, p. 645–692, 2008.

[14] M. Siponen and R. Baskerville, "A New Paradigm for Adding Security into are Development Methods," Advances in Information Security Management & Small Systems Security, vol. 72, pp. 99-111, 2001.

[15] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," IEEE Transactions on Software Engineering, 36(1), pp. 20-36, 2010.

[16] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smell in Object-Oriented Code," in Seventh International Conference on the Quality of Information and Communications Technology, Porto, 2010.

[17] M. Almorsy, J. Grundy and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in Automated Software Engineering (ASE), Essen, Germany, 2012.

[18] M. R. Islam and M. F. Zibran, "A Comparative Study on Vulnerabilities in Categories of Clones and Non-cloned Code," in Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, 2016.

[19] K. Z. Sultana, A. Deo, and B. J. Williams, "A Preliminary Study Examining Relationships Between Nano-Patterns and Software Security Vulnerabilities," in Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, 2016.

[20] H. Alves, B. Fonseca, and N. Antunes, "Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study," in Dependable Computing Conference (EDCC), Gothenburg, Sweden, 2016.

[21] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An Empirical Examination of the Relationship between Code Smells and Merge Conflicts," in Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 2017.

[22] Bland, J. M., & Altman, D. (1986). Statistical methods for assessing agreement between two methods of clinical measurement. The lancet, 327(8476), 307-310.

[23] de Sousa, D. B. C., Maia, P. H., Rocha, L. S., & Viana, W. (2018, September). Analyzing the Evolution of Exception Handling Anti-Patterns in Large-Scale Projects: A Case Study. In Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse (pp. 73-82). ACM.

[24] Gai, K., & Qiu, M. (2018). Blend arithmetic operations on tensor-based fully homomorphic encryption over real numbers. IEEE Transactions on Industrial Informatics, 14(8), 3590-3598.

[25] Gai, K., Choo, K. K. R., Qiu, M., & Zhu, L. (2018). Privacy-preserving content-oriented wireless communication in internet-of-things. IEEE Internet of Things Journal, 5(4), 3059-3067.

[26] Gai, K., Qiu, M., Xiong, Z., & Liu, M. (2018). Privacy-preserving multi-channel communication in edge-of-things. Future Generation Computer Systems, 85, 190-200.