Journal of Engineering Technology
Volume 6, Issue 1, Jan. 2018, PP.570-583

RESEARCH ARTICLE -ENGINEERNG TECHNOLOGY

Software vulnerabilities detection based on security metrics at the design and code levels: empirical findings

Mamdouh Alenezi and Mohammad Zarour

Department of Software Engineering, Prince Sultan University
P.O Box. 66833, Riyadh, 11586
Kingdom of Saudi Arabia

Abstract: Vulnerabilities detection using source code metrics, such as size and complexity, has been studied thoroughly in the literature. Few studies are concerned in detecting vulnerabilities during the software design phase before implementing the solution. Most of the research in the literature is interested in studying software vulnerabilities at the code level. Less research work has considered the architectural metrics to predict security vulnerabilities. Hence, in this paper, we try to consider the main metrics of software architecture in conjunction with the popular code level metrics to study software vulnerabilities and the relationship among these metrics. The literature has been reviewed to identify various security metrics to be used in studying nine common Java libraries. Understand and SonarCube tools are used to collect metrics of the specified libraries, The WEKA tool is used to apply the ReliefF algorithm, a feature selection algorithm, to decide which of the specified metrics are good predictors of software vulnerability.
We investigated whether software design metrics can be used in predicting vulnerable classes, and guide actions for code improvement, hence can help development team to prioritize validation and verification efforts. We found that metrics like Number of Children (NOC), Count of Base Classes (CBC), Response for a Class (RFC), Coupling Between Objects (CBO), Lines of Code (LOC) and McCabe's Cyclomatic Complexity are good vulnerabilities' predictors.

Keywords: Software security, Vulnerabilities, Metrics, software design

1. Introduction

Security is one of the eight characteristics of product quality defined by ISO 25010 [1]. This standard defines security as the "degree to which a product or system protects information and data so that persons or other products have the degree of data access appropriate to their types and levels of authorization" [1]. Security is concerned in maintaining confidentiality, integrity, non-repudiation, accountability and authenticity which form together the main security sub-characteristic in the ISO standard. Any weakness in maintaining any of these sub-characteristics can become security vulnerability. Security as a quality attribute can be studied as a software product attribute, computer system attribute and information system performance attribute. Hence, it can influence quality in use for primary users. Our work is concerned with security as a software product attribute where we aim to explore various software measures that have positive influence on software security mainly during software design.

Although security becomes a very important requirement in most of the software systems nowadays, its related issues are not very well addressed by traditional software metrics [2]. This has resulted in producing tremendous amount of insecure software where insecure software is eating the world [3]! Recently, Google detected more than 1000 bugs in 47 open source projects. The widely used component based development approach where COTS, like software libraries, are widely used, can introduce more vulnerabilities if the security of theses libraries is not guaranteed. Unfortunately, COTS components have been developed without a focus on robustness and security [4]. A widely used approach is assessing the software security is based on identifying pre-existing vulnerabilities [5][6][7]. Security measures can be applied either at the system high level or from a low level (i.e. the program code's level) [8][9]. Measuring security at the code level makes it hard and expensive to discover and fix vulnerabilities caused by software design errors. Design phase measures might efficiently reduce software security issues, such measures have not been considered as security indicators until recently [10][11].

In this paper, we focused on studying the security of popular Java libraries and what measures can be used at the design and code levels to measure their security. The rest of this paper is organized as follows: Section 2 discusses software vulnerabilities measurement. Section 3 presents software libraries. Section 4 describes object oriented security measures and related work. Section 5 presents measurement collection process, while section 6 discusses findings. Conclusion and future work are discussed in Section 7.

2. Measuring Software Vulnerabilities

Vulnerabilities refer to any flaw or weakness in the system's design or software. The amount of vulnerabilities and their severity level are significant indicators of software security and trustworthiness; discovering vulnerabilities in a software product will negatively affect the level of trustworthiness of that product.

Unfortunately, the literature does not specify clearly which metrics are good predictors of vulnerabilities. The literature possess certain level of uncertainty in this regard; for instance, A positive correlation between vulnerabilities and code metrics, such as complexity and size, have been recorded in [12] and [13], while shin and William [14] and Morrison et. al. [15] found weak correlation between complexity and vulnerabilities! Sin and Williams [16] found that complexity metrics can predict vulnerabilities at a low false positive rate, but at a high false negative rate.

Zimmermann et al. [17] observed that "classical software measures predict vulnerabilities with a high precision but low recall values. The actual dependencies, however, predict vulnerabilities with a lower precision but substantially higher recall". Experimental results of Chowdhury and Zulkernine [17] indicate that metrics such as complexity, coupling, and cohesion are useful in vulnerability prediction. Neuhaus et. al. [15] developed a tool that is useful in predicting vulnerable components using mining techniques. Similar mining techniques and N-Gram analysis are used in [18], [19] for the same purpose. Kamongi et. al. [20] developed a predictive model that represents product maturity in terms of source-code base growth and vulnerability disclosure history to predict new possible vulnerabilities.

Accordingly, most of the research in the literature is interested in studying software vulnerabilities at the code level. Less research work has considered the architectural metrics to predict vulnerabilities. Hence, in this study, we try to consider the main metrics of software architecture in conjunction with the popular code level metrics to study software vulnerabilities and the relationship among these metrics.

3. Software Libraries

A software library is a collection of software reusable, extendable and generic components to support programming through a well-established clear interface [21]. Libraries include specified rules that should be adhered to when accessing their functionalities. Clients should be able to extend their functionalities, reuse available ones, customize or specialize some of their generic functionalities. According to these characteristic, libraries should be well-designed pieces of code, which adhere to software design principles. This assumption is based on the belief that libraries are envisioned to support several clients and software. This also entails having these libraries to follow specific constraints on the internal development practices leading to robust software architecture [22].

Using security measures of [6][9], the software or library is considered secure if no vulnerabilities are introduced otherwise it is considered insecure. It is black and white approach (binary) that does not quantify how secure a code is. Therefore, there is still neccasity to measure the library security using well-establsihed metrics [23]. With regard to the libraries' quality design, a study showed that libraries exhibit higher quality compared to regular software systems [24]. Nevertheless, to the best of our knowledge, studies that have been done to evaluate the security of libraries by means of software metrics at the design level are rare. In this paper, we aim to measure software security metrics at the design level of object oriented software. We are doing this using the nine most popular java libraries. Studying the security of such libraries is pivotal due to their popularity. Some general measures are given in Table-1 for the selected libraries that include:

1. Apache Log4j is a popular Java library for enabling logging without modifying the application binary. It allows the developer to control which log statements are output with arbitrary granularity by using external configuration files. Log4j is designed to be reliable, fast and extensible.
2. Apache Commons IO is a library of utilities to assist with developing IO functionality by performing various input/output operations. It includes six main areas; utility classes, input, output, filters, comparators, and file monitor.
3. Apache Commons Lang provides a host of helper utilities for the java, lang API, String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties.
4. Gson can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. It is not necessary to have the source code of java objects in order to convert them to JSON.
5. Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API. This library does not need any configuration file to be used and is mostly used by calling methods and instantiating classes.
6. JUnit is an open source java library designed for the purpose of writing and running tests and test cases. JUnit is an important tool in the evolution of test-driven development.
7. Logback is conceptually very similar to its ancestor log4j, but brings many improvements over log4j. Designed of three main modules logback-core, logback-classic and logback-access. Logback-classic module can be thought of as advanced log4j. By natively implementing slf4j API, it is very easy to switch between logging frameworks. Integration with servlet containers like Tomcat or Jetty is handled by logback-access module. Finally, as name says logback-core provides main support for previously named two modules.
8. Mockito is a mocking framework that lets you write tests with a clean & simple API. Mockito promises very readable tests with clean verification errors.

9. The Simple Logging Facade for Java (SLF4J) is a simple facade for different logging frameworks (e.g. java.util.logging, logback, log4j) to allow end users to plug in the desired logging framework at deployment time.

   Table 1 summarizes some common measures that correspond to the selected libraries. These measures are mainly the number of classes per library and the number of vulnerabilities as well.

4. Object-Oriented Security Measures

Security measures at the design level includes: composition, coupling, extensibility, inheritance, and the design size of a given object-oriented, multi-class program from the point of view of potential information flow [8]. Measures to assess the security of a single object-oriented class include Data Encapsulation and Cohesion for a given class [25]. These measures would allow developers to compare the level of security of different designs and implementations. Although Complexity measures are found to have weak correlation with security problems [26][16], Shine et. al. [27] used complexity metrics, code churn, and developer activity metrics to build vulnerability prediction models.

Table-1: Common Measures of the Selected Libraries

Library Name	Version	Number of Classes	Number of vulnerabilities
Apache Log4j	2.7	1912	282
Apache Commons IO	2.5	297	66
Apache Commons Lang	3.5	596	46
Gson	2.8.0	758	24
Joda-Time	2.9.7	528	46
Junit	r5.0.0-M3	753	6
Logback	1.1.8	1166	130
Mockito	1.10.19	557	84
slf4j	1.7.22	243	40

Zimmermann et al. [28] studied the relationship between vulnerabilities and some software metrics (e.g., complexity, churn, coverage, dependency measures, and organizational structure). They employed Logistic Regression to predict vulnerabilities based on the metrics. Their experiments on Windows Vista indicated that these metrics can be used to get some insights about software vulnerabilities.

Agrawal et al. [29] proposed a class measurement for vulnerabilities. The measurement calculates how much vulnerable classes are there in a design. A class is considered as vulnerable if it has sensitive data members or methods. Argawal and Khan [30] studied the effect of inheritance on the security level of object-oriented design. They proposed an algorithm to measure vulnerability propagation for an object-oriented design that calculates the Attribute Vulnerability Ratio (AVR). Alenezi and Abunadi [31] considered several metrics in their pursuit of predicting security issues in PHP web applications. They used size, volume, coupling, and complexity metrics and found out that some metrics are discriminative and predictive of security vulnerabilities.

Table-2 summarizes metrics found in the literature that have been explicitly used to measure software security at the design level.

Table-2: Security Measures

Reference	Security Measures	Description
[32]	Complexity	Coupling, SumFanIn, SumFanOut, MaxFanIn, MaxFanOut
[33]	Coupling Induced Vulnerability Propagation (CIVP)	calculates relative vulnerability of two object oriented designs of same software as well as designs of different object-oriented software.
[34]	Design-level Complexity Metrics Design-level Coupling Metrics Design-level Cohesion Metrics	WMC, DIT, NOC, CBC DIT, NOC, CBC, RFC, CBO LCOM,
[35]	Size Complexity	Lines of code Cyclomatic complexity, Maximum nesting complexity Fan-in, Fan-out
[36]	Size Complexity Coupling	LOC McCabe's Cyclomatic Complexity Fan-in, Fan-out

Accordingly, Table-3 summarizes the metrics to be used to measure the security vulnerabilities.

Table-3: Metrics to be Used to Measure the Security Vulnerabilities

No.	**Complexity Metrics**
M1	SumFanIn
M2	SumFanOut
M3	MaxFanIn
M4	MaxFanOut
M5	Weighted Methods per Class (WMC)
M6	Depth of Inheritance Tree (DIT)
M7	Number of Children (NOC)
M8	Count of Base Classes (CBC)
M9	Response for a Class (RFC)
M10	Coupling Between Objects (CBO)
M11	Lack of Cohesion of Methods (LCOM)
M12	Lines of Code (LOC)
M13	McCabe's Cyclomatic Complexity
M14	Maximum nesting complexity

5. Measurements Collection

In order to collect and analyze the security issues of the selected libraries, we used two main tools: Understand tool [37] and SonarQube software [38].

A. Understand Tool [37]:
Understand is a static code analysis tool that helps programmers understand and visualize complex code. It is vital to understand legacy code with poor documentation. This tool is used to collect metrics depicted in Table 3.

B. SonarQube software [38]: SonarQube is an open source software aims to be the source code's quality management platform and is developed under LGPL v3 license. It provides control over a large number of software metrics by the development team. SonarQube helps identifying how the code is evolving, and illustrates the behavior of the multiple quality measures, while pointing possible software bugs. it visualizes findings and provide an overview of the overall health of the source code. SonarQube supports many languages besides Java and can be integrated with IDEs and other external tools. The SonarQube tool manages to keep the number of false positives vulnerabilities very low which keeps the number of issues reported by the tool low. This makes it a very viable tool to include in a development process [39].
SonarQube adopts two types of rules: standard rules and security related rules. Standard rules should not produce any false positive issues where as security related rules can produce some false positive issues. Every rule represents a single-issue type in the code, as exception should be caught instead of thrown. Rules can have tags defined, which makes it easier to categorize rules, tags can be something like security, Common Weakness Enumeration (CWE) or convention [40]. The Java Plugin itself contains more than 390 rules for analyzing Java source code. There are rules for coding conventions, bug detection and security problems. Security related rules contains checks for some CERT and CWE weaknesses as for some SANS top 25 most dangerous software errors and some OWASP top 10 weaknesses. All rules are CWE compatible so it is possible to search rules by CWE identifier [41].

Table 4 summarizes the SonarQube two security metrics used in our case study that relate to vulnerabilities, namely, the number of vulnerabilities and security rating. The overall security rating has five levels as follows:

- A: when there is no vulnerability
- B: when there is at least one minor vulnerability
- C: when there is at least one major vulnerability
- D: when there is at least one critical vulnerability
- E: when there is at least one blocker vulnerability

Table-4: SonarQube Security Metrics

Name	**Key**	**Description**
Vulnerabilities	vulnerabilities	Number of vulnerabilities.

Security Rating	security_rating	A = 0 Vulnerability
		B = at least 1 Minor Vulnerability
		C = at least 1 Major Vulnerability
		D = at least 1 Critical Vulnerability
		E = at least 1 Blocker Vulnerability

The number of vulnerabilities, overall security rating and the total number of classes per Java library have been measured and documented in Table 5. These metrics will be used to calculate the correlation between different metrics and vulnerabilities. As can be seen from Table 5, seven Java libraries are found to have high security rating with at least one minor vulnerability while two others are found to have serious vulnerabilities rated as (E).

Table-5: Common Metrics of the Selected Libraries

Library Name	Number of Classes	Number of Vulnerabilities	Overall Security Rating
Apache Log4j	1912	282	E
Apache Commons IO	297	66	B
Apache Commons Lang	596	46	B
Gson	758	24	B
Joda-Time	528	46	B
Junit	753	6	B
Logback	1166	130	E
Mockito	557	84	B
slf4j	243	40	B

The Understand tool has been used to measure the Java libraries based on the metrics shown in Table 3. The collected metrics are shown in Table 6 and visualized in Figure 1.

Table 6: Collected Metrics for the specified Java Libraries

Java Library	**M 1**	**M 2**	**M 3**	**M 4**	**M 5**	**M 6**	**M 7**	**M 8**	**M 9**	**M 10**	**M 11**	**M 12**	**M 13**	**M 14**
apache-log4j-2.7	3.2617	9.3828	51	105	9.3263	1.5120	0.2673	1.3284	13.0622	4.0319	34.4189	57.5842	1.3996	0.8902
commons-io-2.5	1.5545	8.2091	19	49	15.3872	1.6431	0.3872	1.3367	11.6296	1.5892	36.7575	95.4949	1.3232	0.9832
commons-lang3-3.5	1.7059	7.8897	17	43	19.4127	1.2718	0.2701	1.3640	14.1007	1.6560	35.0520	124.7919	1.1779	0.7248
gson-gson-parent-2.8.0	4.1032	8.7226	64	60	5.5738	1.2652	0.3667	1.5118	3.8681	2.2454	10.9947	36.5198	0.9934	0.4037
joda-time-2.9.7	7.0043	11.8017	105	51	27.4394	2.0397	0.4223	1.4602	41.6988	6.8617	50.6042	172.8787	1.3220	1.01326
junit5-r5.0.0-M3	3.1033	6.7975	40	31	5.65867	1.1939	0.1420	1.2775	5.4807	3.6082	17.5817	38.8672	0.8805	0.2749
logback-1.1.8	3.4765	7.8758	86	30	8.3045	1.9117	0.3687	1.2281	13.0960	3.6732	31.6801	43.7169	1.3379	0.8508
mockito-all-1.10.19	3.7135	7.2865	82	52	12.9408	1.4847	0.2962	1.6050	9.0179	3.6320	35.5009	58.4416	1.6140	0.9659
slf4j-1.7.22	1.4762	6	7	17	10.1770	1.2469	0.1028	1.3497	8.3210	2.1481	39.7654	45.7037	1.2592	0.6996

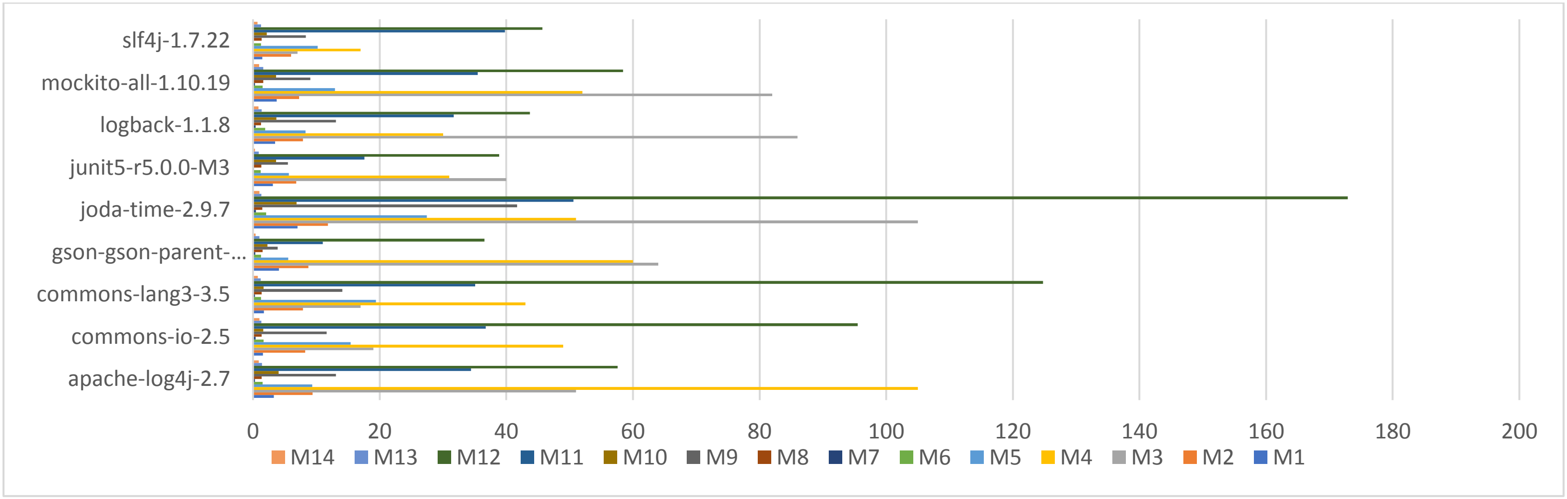

Figure 1: Metrics' values for the selected Java Libraries

Table 7: Correlation table among the collected Metrics

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14
M1	1													
M2	**0.7836**	1												
M3	**0.8939**	**0.6140**	1											
M4	0.2427	**0.5351**	0.2163	1										
M5	0.3952	**0.6287**	0.2194	0.0089	1									
M6	**0.6002**	**0.6770**	**0.6847**	0.1029	**0.5397**	1								
M7	**0.5313**	**0.7323**	**0.6097**	0.3239	0.4375	**0.7300**	1							
M8	0.3954	0.2268	0.3662	0.2167	0.2606	-0.0593	0.2663	1						
M9	**0.7029**	**0.8175**	0.5189	0.1005	**0.8674**	**0.7584**	0.4962	0.1097	1					
M10	**0.8918**	**0.6871**	**0.7923**	0.2169	0.4258	**0.6579**	0.3055	0.1402	**0.7696**	1				
M11	0.1794	0.3613	0.1174	-0.020	**0.7794**	**0.5963**	0.1811	0.0286	**0.7338**	0.4224	1			
M12	0.409	**0.694**	0.194	0.066	**0.980**	0.513	0.470	0.180	**0.867**	0.416	**0.682**	1		
M13	0.088	0.150	0.307	0.272	0.336	0.4993	0.297	0.315	0.275	0.236	**0.677**	0.19	1	
M14	0.171	0.412	0.283	0.258	**0.60**	**0.7108**	**0.506**	0.153	**0.556**	0.316	**0.841**	**0.506**	**0.90**	1

6. Findings

Table 7 presents the correlation matrix among the collected metrics that we are studying. The correlation is considered high if the coloration r>=0.7 and is considered moderate if the correlation is 0.5 <= r < 0.7 otherwise the correlation is considered weak. We find, from Table 7, that there is high correlation between

1. SumFanIn (M1) and SumFanOut (M2), MaxFanIn (M3), Reponses for class (M9) and Coupling between objects (M10). With reference to metrics' categories discussed in [42], we can say that there

is strong correlation among the software complexity measures (M1, M2 and M3), Dependency among objects (M10) and size (M9).

2. SumFanIn (M2) and Number of children (M7), response for a class (M9), i.e. there is strong correlation between complexity (M2 and M7) and effort (M9). This supports the first finding.
3. MaxFanIn (M3) and Coupling of Objects (M10), i.e. there is strong correlation between complexity (M3) and dependency (M10). This supports the first finding.
4. Weighted Methods per Class (M5) and Reponses for a class (M9), Lack of cohesion of methods (M11) and Lines of code (M12). This means that there is strong correlation between complexity (M5, M12) and size (M9, M12). This supports the first finding.
5. Depth of inheritance tree (M6) and Number of children (M7), Response for a class (M9), Maximum nesting complexity (M14). This means that there is strong correlation between dependency (M6), complexity (M7, M14) and size (M9). This supports the first finding.
6. Response for a class (M9) and Coupling between objects (M10), Lack of Cohesion of Methods (M11) and Lines of code (M12). This means that there is strong correlation between dependency (M10) complexity (M11) and size (M9, M12).
7. Lack of Cohesion of Methods (M11) and Maximum nesting complexity (M14) and both are complexity metrics.
8. McCabe’s Cyclomatic Complexity (M13) and Maximum nesting complexity (M14) and both are complexity metrics.

Overall, twelve metrics (M1, M2, M3, M5, M6, M7, M9, M10, M11, M12, M13 and M14) are found to correlate among each other while one metric (M4) is found to have moderate correlation with (M2). Another metric M8 is found to have weak correlation with other metrics. The correlated methods correspond to complexity, size, dependency and effort.

Moreover, and to identify which metrics can be considered as good predictors of the software vulnerabilities, the WEKA tool [44] is used to run a feature selection algorithm. Feature selection algorithms can evaluate individual attributes that identify which metric is able to serve as discriminatory attribute for indicating an external quality. WEKA tool implements a feature selection algorithm called ReliefF which evaluates the worth of an attribute by repeatedly sampling an instance and considering the value of the given attribute for the nearest instance of the same and different class [43]. ReliefF is an extension of the Relief algorithm that can handle noise and multiclass data sets. After running the algorithm in the 9 libraries, six metrics were found valuable in indicating software security (see Table 8), hence how vulnerable the software is.

Table 8: Good predictors of Software vulnerabilities and their categories

Metric	Metric Category [42]
M7: Number of Children (NOC),	Complexity
M8: Count of Base Classes (CBC)	Size
M9: Response for a Class (RFC)	Size
M10: Coupling Between Objects (CBO)	Dependency
M12: Lines of Code (LOC)	Size
M13: McCabe’s Cyclomatic Complexity	Complexity

7. Conclusion and Future Work

In this work, we studied the top used open source Java libraries with regard to their design and complexity metrics. These metrics are studied to identify which of them can predict possible software vulnerabilities. We studied these libraries with regard to size, coupling, cohesion (as dependency metrics), and complexity. We investigated whether software design metrics can be used in predicting vulnerable classes, and guide actions for code improvement to help development team in prioritizing validation and verification efforts.

We found that metrics like Number of Children (NOC), Count of Base Classes (CBC), Response for a Class (RFC), Coupling Between Objects (CBO), Lines of Code (LOC) and McCabe's Cyclomatic Complexity are good vulnerabilities' predictors. Our future work will focus on conducting similar studies with a broader set of libraries and systems, including both commercial and open source applications to investigate more the influencing metrics in detecting software vulnerabilities.

References

[1] ISO/IEC JTC 1/SC 7, "ISO/IEC 25010:2011. Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models," 2011.

[2] C. Banerjee, A. Banerjee, and P. D. Murarka, "Evaluating the Relevance of Prevailing Software Metrics to Address Issue of Security Implementation in SDLC," Int. J. Adv. Stud. Comput. Sci. Eng. IJASCSE, vol. 3, no. 3, 2014.

[3] J. Daley, "Insecure software is eating the world: promoting cybersecurity in an age of ubiquitous software-embedded systems," Stanford Technol. Law Rev., vol. 19, 2016.

[4] M. Susskraut and C. Fetzer, "Robustness and Security Hardening of COTS Software Libraries," in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), 2007, pp. 61–71.

[5] B. Chess and J. West, Secure programming with static analysis. Addison-Wesley, 2007.

[6] K. Maruyama, "SECURE REFACTORING Improving the Security Level of Existing Code," in Proceedings of the Second International Conference on Software and Data Technologies, 2007.

[7] J. Viega, G. McGraw, T. Mutdosch, and E. W. Felten, "Statically Scanning Java Code: Finding Security Vulnerabilities," IEEE Softw., vol. 17, no. 5, pp. 68–74, Sep. 2000.

[8] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Designs," in 2010 21st Australian Software Engineering Conference, 2010, pp. 55–64.

[9] M. Howard and D. LeBlanc, Writing secure code. Microsoft Press, 2003.

[10] A. Sachitano, R. O. Chapman, and J. A. Hamilton, "Security in software architecture: a case study," in Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004., pp. 370–376.

[11] E. A. Schneider and E. A., "Security architecture-based system design," in Proceedings of the 1999 workshop on New security paradigms - NSPW '99, 2000, pp. 25–31.

[12] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," Comput. Fraud Secur., vol. 2013, no. 5, pp. 8–17, May 2013.

[13] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," IEEE Trans. Softw. Eng., vol. 37, no. 6, pp. 772–787, Nov. 2011.

[14] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in Proceedings of the 4th ACM workshop on Quality of protection - QoP '08, 2008, p. 47.

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

[16] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08, 2008, p. 315.

[17] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 421–428.

[18] Y. Pang, X. Xue, and A. S. Namin, "Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection," in 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), 2015, pp. 543–548.

[19] 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. 993–1006, Oct. 2014.

[20] P. Kamongi, K. Kavi, and G. Mahadevan, "Predicting Unknown Vulnerabilities using Software Metrics and Maturity Models," in The Eleventh International Conference on Software Engineering Advances (ICSEA 2016), 2016, pp. 311–317.

[21] T. Chaikalis, A. Chatzigeorgiou, A. Ampatzoglou, and I. Deligiannis, "Assessing the Evolution of Quality in Java Libraries," in Proceedings of the 7th Balkan Conference on Informatics Conference - BCI '15, 2015, pp. 1–4.

[22] J. Tulach, Professional API design : confessions of a Java framework architect. Apress, 2008.

[23] G. McGraw, Software Secuirty: Building Security. Addison-Wesley, 2006.

[24] A. Chatzigeorgiou and E. Stiakakis, "Benchmarking library and application software with Data Envelopment Analysis," Softw. Qual. J., vol. 19, no. 3, pp. 553–578, Sep. 2011.

[25] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-Oriented Class Designs," in 2009 Ninth International Conference on Quality Software, 2009, pp. 11–20.

[26] Y. Shin and L. Williams, "Is complexity really the enemy of software security?," in Proceedings of the 4th ACM workshop on Quality of protection - QoP '08, 2008, p. 47.

[27] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," IEEE Trans. Softw. Eng., vol. 37, no. 6, pp. 772–787, Nov. 2011.

[28] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 421–428.

[29] A. Agrawal, S. Chandra, and R. A. Khan, "An Efficient Measurement of Object Oriented Design

Vulnerability," in 2009 International Conference on Availability, Reliability and Security, 2009, pp. 618–623.

[30] A. Agrawal and R. A. Khan, "Impact of inheritance on vulnerability propagation at design phase," ACM SIGSOFT Softw. Eng. Notes, vol. 34, no. 4, p. 1, Jul. 2009.

[31] M. Alenezi and I. Abunadi, "Evaluating Software Metrics as Predictors of Software Vulnerabilities," Int. J. Secur. Its Appl., vol. 9, no. 10, pp. 231–240, 2015.

[32] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," IEEE Trans. Softw. Eng., vol. 37, no. 6, pp. 772–787, Nov. 2011.

[33] A. Agrawal and R. A. Khan, "A Vulnerability Metric for the Design Phase of Object Oriented Software," Springer Berlin Heidelberg, 2010, pp. 328–339.

[34] I. Chowdhury, "USING COMPLEXITY, COUPLING, AND COHESION METRICS AS EARLY INDICATORS OF VULNERABILITIES," 2009.

[35] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," in 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 23–33.

[36] P. Morrison, D. Moye, and L. A. Williams, "Mapping the Field of Software Security Metrics," North Carolina State University. Dept. of Computer Science, 2014.

[37] S. Toolworks, "Understand Tool." [Online]. Available: https://scitools.com/. [Accessed: 08-Mar-2017].

[38] SonarSource, "SonarQube Platform." [Online]. Available: https://www.sonarqube.org/. [Accessed: 08-Mar-2017].

[39] A. Ramos, "Evaluating the ability of static code analysis tools to detect injection vulnerabilities," Umeå University, 2016.

[40] A. Campbell, "Rules - SonarQube Documentation - SonarQube," 2016. [Online]. Available: https://docs.sonarqube.org/display/SONAR/Rules#Rules-RuleDetails. [Accessed: 15-Mar-2017].

[41] D. Racodon, "SonarJava | SonarSource," 2016. [Online]. Available: https://www.sonarsource.com/why-us/products/codeanalyzers/sonarjava.html. [Accessed: 15-Mar-2017].

[42] P. Morrison, D. Moye, and L. A. Williams, "Mapping the Field of Software Security Metrics," North Carolina State University. Dept. of Computer Science, 2014.

[43] M. Robnik-Sikonja and I. Kononenko, "An adaptation of Relief for attribute estimation in regression," in Fourteenth International Conference on Machine Learning, 1997, pp. 296–304.

[44] I. H. (Ian H. . Witten, E. Frank, M. A. (Mark A. Hall, and C. J. Pal, Data mining : practical machine learning tools and techniques. 2016.