Check for updates

Empirical Analysis of Static Code Metrics for Predicting Risk Scores in Android Applications

Mamdouh Alenezi(✉) and Iman Almomani

College of Computer and Information Sciences, Prince Sultan University, Riyadh, Saudi Arabia
{malenezi,imomani}@psu.edu.sa

Abstract. Recently, with the purpose of helping developers reduce the needed effort to build highly secure software, researchers have proposed a number of vulnerable source code prediction models that are built on different kinds of features. Identifying security vulnerabilities along with differentiating non-vulnerable from a vulnerable code is not an easy task. Commonly, security vulnerabilities remain dormant until they are exploited. Software metrics have been widely used to predict and indicate several quality characteristics about software, but the question at hand is whether they can recognize vulnerable code from non-vulnerable ones. In this work, we conduct a study on static code metrics, their interdependency, and their relationship with security vulnerabilities in Android applications. The aim of the study is to understand: (i) the correlation between static software metrics; (ii) the ability of these metrics to predict security vulnerabilities, and (iii) which are the most informative and discriminative metrics that allow identifying vulnerable units of code.

Keywords: Static code metrics · Risk scores · Android
Security prediction models

1 Introduction

Several software systems bump into security issues during their lifetime. To ensure building highly secure software systems, developers need to invest the right amount of time in testing and debugging security issues. Nonetheless, due to limited resources, it is usually not possible to thoroughly check every file in a software system. Prioritization and different level of focus are needed to check and test more parts of the system that are more prone to be vulnerable. Regrettably, identifying these vulnerable parts is not an easy task since there are many parts in a software system, and only a few of them are vulnerable.

Software security vulnerabilities are a continuous danger to software businesses and their clients. Evaluating the security of a software system requires prioritizing resources and minimizing risks. Several available techniques can be used to identify security vulnerabilities before releasing the software such as manual inspection, static and dynamic analysis. However, these techniques were found to be error-prone and resource-intensive activities. This research aims to improve the detection of security

M. Alenezi and B. Qureshi (Eds.): *5th International Symposium on Data Mining Applications*, pp. 84–94, 2018.
https://doi.org/10.1007/978-3-319-78753-4_8

issues by building a predictive model that will enable engineers to focus their activities on non-secure apps. Basili et al. [1] suggested that prediction models can support software project planning, scheduling, and decision-making. They enable software teams to properly allocate needed resources to modules that are more likely to be defect-prone. Generally, security vulnerabilities are a subset of defects.

In this work, we study which software static metrics have more relation with security vulnerabilities in the source code. We hypothesize that some metrics can be employed to distinguish between vulnerable or non-vulnerable code. A dataset including 1407 Android apps with different static code metrics was analyzed. The purpose of this research is to design an empirical study that aims to find the correlations among these metrics, their impact in detecting vulnerabilities in source codes and to define set of metrics which highly contributing to the prediction of security vulnerabilities. The results of this empirical study showed a strong correlation among some of the static metrics. Moreover, by applying machine learning algorithm on the complete dataset with 21 static metrics, the ability of these metrics to predict security vulnerabilities was observed with accuracy reached 94.4% in some classes. To find out the set of metrics which have more influence in detecting the insecure codes, a feature selection algorithm was also implemented. This algorithm succeeded to choose 9 out of 21 metrics which reduced the complexity of the prediction model without affecting its accuracy. As a result, this research has mainly introduced an efficient risk score prediction model for Android applications.

The rest of paper is organized as follows: Sect. 2 presents recent related work. Section 3 introduces the empirical study setup. The experiments and their results are detailed in Sect. 4. Then a discussion section is followed. Section 6 concludes the paper and presents possible future work.

2 Related Work

Software security research has been going for a long period of time, several topics were discussed by researchers, including security protocols and patterns to build secure systems [2], software security testing [3], vulnerability detection [4], attack prediction [5], and intrusion detection systems [6], just to name some. This shows that building software without security vulnerabilities, despite the huge advances in software development processes, is still very difficult, if not impossible [7].

Majority of similar studies concentrated on detecting and categorizing malicious Android apps through the use of permissions [8], dynamic analysis, and machine learning techniques [9]. Rahman et al. [10] investigated how effectively static code can be used to predict security risk of Android applications. Based on 21 static code metrics of 1,407 Android applications, and using radial-based support vector machine (r-SVM), they got a precision of 0.83. Syer et al. [11] examined the relationship between files defect-proneness and platform dependence in Android apps. They found that source code files that are defect-prone have a higher dependence on the platform than defect-free files. Previous studies also investigated the undesirable effects of Android apps low-quality source code. Corral and Fronza [12] examined how market success is dependent on source code quality.

Software metrics have been widely-used to build prediction models to predict faults [13, 14]. We also believe that software metrics can be used to build prediction models to predict security vulnerabilities. Several vulnerability detection techniques and tools are implemented by both commercial and open source to detect software vulnerabilities. Generally, these techniques can be categorized broadly into three main categories: static code analysis [15], emulation of security attacks (i.e., also known as penetration testing) [16], and runtime monitoring of the system behavior [17].

While using software metrics to detect and predict security vulnerabilities is not that common, we still can find quite a few studies in the literature to show some relationships between the software internal quality attributes and its security (as an external quality attribute). The work in [18], is one of the first attempts to show that there is a strong correlation between attackability (i.e., likelihood of an attack to succeed on a software system [19]) and coupling metrics (e.g., Coupling Between Objects).

A different effort was focused on predicting software security vulnerabilities using quantitative metrics [20]. A new metric called vulnerability density (i.e., number of vulnerabilities per unit of code) was proposed to be used for comparing software systems within the same category in terms of functionality. They also investigated the possibility of predicting the number of vulnerabilities. In a different study [21], the authors tried to understand where the majority of the vulnerabilities occur in a software. They were not successful in finding a correlation between complexity or buffer usage and the number of vulnerabilities. The authors in [22] focused more on complexity metrics to predict failures and security vulnerabilities in software. They employed machine learning approach to build a model using nine complexity metrics to predict vulnerabilities. They showed that complexity metrics (e.g., Cyclomatic Complexity) can predict vulnerabilities, but with a very high false negative rate. In [23], the authors used nine function-level complexity metrics to build a framework to automatically predict vulnerabilities (in addition to other coupling and cohesion file-level metrics, in a total of 17). Several file-level metrics and development activity metrics (a total of 28) were used to distinguish between vulnerable and neutral files in [24]. The results showed their effectiveness in discriminating and predicting vulnerable files.

Recently, the authors in [25] used 27 function-level metrics to examine the correlation between the software internal quality and security vulnerabilities. Although they did not find a strong correlation between these metrics and the number of vulnerabilities, they found that software metrics can be used to discriminate vulnerable functions from non-vulnerable ones.

The following sections present the structure of an empirical study, its implementation, results' discussions and analysis. This study aims to investigate the impact of static code metrics in detecting insecure software code, which metrics and to what extent they can contribute to propose high-performance prediction models.

3 Empirical Study Setup

This section presents the empirical study conducted to examine the impact of static code metrics in predicting the level of vulnerability in Android apps. Figure 1 shows the empirical study structure.

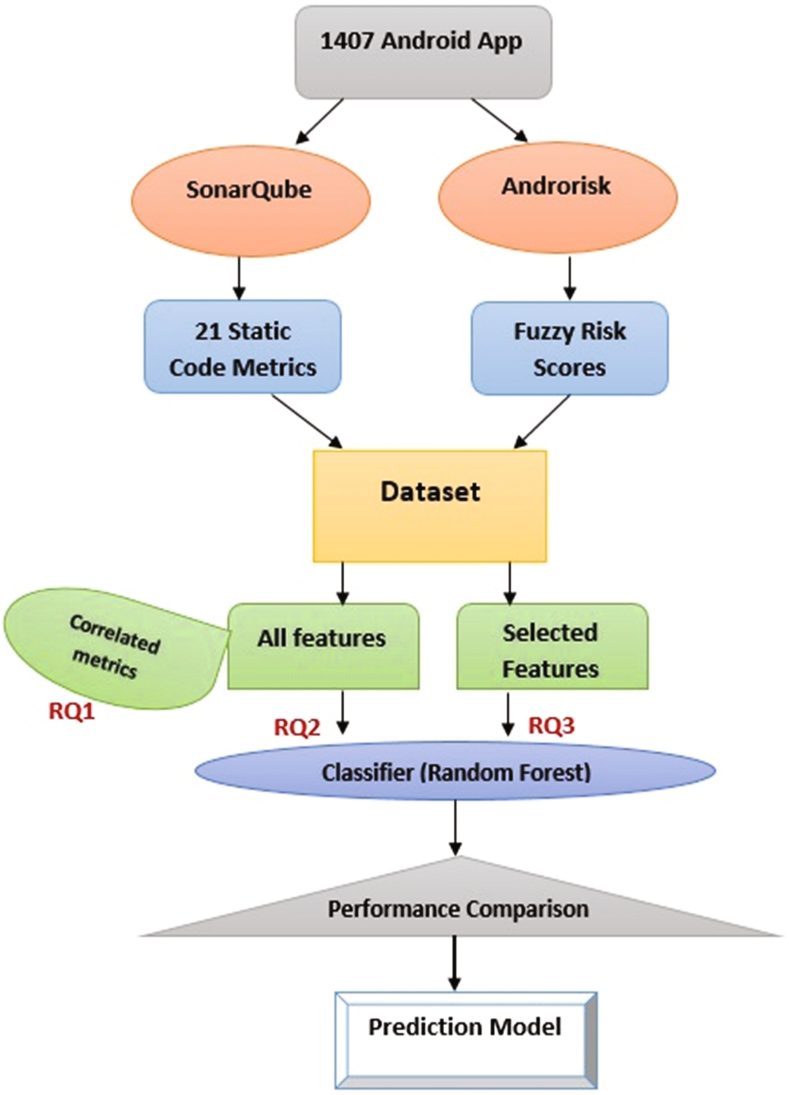

Fig. 1. Empirical study structure

This study has used a dataset of 1407 Android apps obtained from the authors of the work presented in [10]. SonarQube [26] analyzed these apps to extract 21 static code metrics. SonarQube is a well-known tool that uses source code static analysis and produces metrics. Since Android applications are built using Java, SonarQube was used to extract the static code metrics from the Android Java source code files. The 21 static code metrics are classified as follows:

- **Object-oriented:** Class complexity, Comment lines, Complexity, Density of comment lines, Files, File complexity, Function complexity, Lines, Lines of code, Methods, Number of classes, Percentage of comments, Percentage of duplicated lines. The
- **Bad Coding Practice:** Blocker practices, Critical practices, Major practices, Minor practices, Total bad coding practices
- **Duplication:** Duplicated blocks, Duplicated files, Duplicated lines

Moreover, Androrisk [27] tool was used to give a risk score to the Android applications using fuzzy logic. This risk score is an approximation of the amount of security and privacy risk for the Android application. Androrisk calculates a risk score between 0 and 100 for each app based on different permissions and settings used by the application. Each permission has a weight depending on its sensitivity and risk (i.e. access to the Internet, SMS messages, or payment systems). The presence of more dangerous functionality in the app (i.e. a shared library, use of cryptographic functions, the reflection API) also has an impact on the risk score. Androrisk reports the security risk score for each application. Androrisk is freely available, open-source, and has the ability to quickly process a large number of apps.

The complete Dataset used by this research's experiments includes the values of the 21 static metrics for each one of the 1407 apps. Also, the dataset categorizes the risk scores into No, Low, Medium, and High based on their statistical distribution to ease the prediction. Table 1 describes the classes/labels and their frequencies in the dataset.

Table 1. Classes categories in the dataset

Class	Number
High	767
Low	28
Medium	22
No	590
Total	1407

Using this dataset, we designed our experiments to answer the following research questions:

- RQ1: Is there any correlations among the static code metrics?
- RQ2: Are static code metrics able to predict security vulnerabilities?
- RQ3: Can static code metrics be ranked according to their contributions in predicting security vulnerabilities?

The following section illustrates how the conducted empirical study answers these research questions.

4 Experiments and Results

To answer the first research question (RQ1), Spearman's Rank Correlation Coefficient was performed. Spearman's correlation can be used to evaluate the statistical dependency between two metrics [28]. One of the strongest points of Spearman is the fact that it does not need a normal distribution of data. Table 2 presents the results obtained from the analysis. Two variables are considered strongly correlated if the value is higher than 0.9 [28]. Since the dataset contains static code metrics, some of these metrics might have similar definitions that could cause redundancy.

Table 2. Strongly correlated static code metrics

	LOC	Functions	Complexity	Files	Violations	Lines
Classes	0.96	0.97	0.97	0.93		0.95
LOC		0.98	0.99	0.94	0.93	0.99
Functions			0.98	0.93	0.9	0.98
Complexity				0.92	0.92	0.97

To answer the second research question (RQ2), a prediction model can be built on the available dataset. The problem is formulated as a classification problem where the static metrics are considered features and the risk level is considered class label.

A classic software security prediction model is trained using static code metrics and security data that have been collected from already developed software. After creating the prediction model, it can be used in new programs that do not have security information.

Based on previous studies [24, 29], we chose Random Forest as our machine learning algorithm to build this prediction model. Random Forests is an ensemble learning method, which produces several decision trees at training time. Each tree gives a class label. The Random Forests classifier selects the class label that has the mode of the classes output by individual trees. The algorithm combines several decision tree classifiers, each one fitted on a random sub-sample of a dataset, making it more accurate and robust to outliers and noise than a single classifier. The Random Forest classifier was implemented in WEKA tool with default parameter settings specified in WEKA.

To get the classification results, we performed a 10-fold cross-validation, which randomly partitions the data into 10 folds, with each fold being the held-out test fold exactly once. The metrics used to evaluate the performance of the classifier are namely Precision, Recall, and ROC (Receiver Operating Characteristic). The authors of [30] argue that ROC is the best measure to report the classification accuracy. In the collected data set, the number of apps with security problems is much lower than the number of apps with no security problems. The Area Under the ROC Curve (AUC) is a preferred measure since it considers the ability of a classifier to differentiate between the two classes. AUC is sound more than other performance metrics since it has lower variance. The AUC value ranges from 0 to 1. The ROC curve characterizes the trade-off between true positives and false positives. The goodness of a classifier is judged based on how large an area under the curve is. Precision measures how many of the vulnerable instances returned by a model are actually vulnerable. The higher the precision is, the fewer false positives exist. Recall measures how many of the vulnerable instances are actually returned by a model. The higher the recall is the fewer false negatives exist.

Table 3 shows the performance of the random forest-based prediction model, where Table 4 displays the confusion matrix in case of considering all features (21 static code metrics). Overall, the best prediction is observed in case of High and No classes.

Table 3. Random forest classification with all features

Precision	Recall	ROC Area	Class
0.854	0.846	0.944	No
0.941	0.571	0.786	Low
0.909	0.455	0.870	Medium
0.867	0.898	0.938	High
0.864	**0.863**	**0.936**	Weighted avg.

Table 4. Confusion matrix for all features

High	No	Low	Medium	
689	76	1	1	**High**
91	499	0	0	**No**
6	6	16	0	**Low**
9	3	0	10	**Medium**

Looking at the problem from a practitioner's point of view, it is preferred to have a model with a small set of metrics. Usually, some of these metrics provide redundant or no new knowledge. In this work and to answer the third research question (RQ3), we did a feature selection to find the best subset of the 21 metrics and to rank them with regard to their contribution in predicting security vulnerabilities in order to build a reliable prediction model.

Hall and Holmes [31] categorized feature selection algorithms to (1) algorithms that evaluate individual attributes and (2) algorithms that evaluate a subset of attributes. Since we are interested to find the best subset of features, we used the second category. Correlation-based feature selection (CFS) is an automatic filter algorithm that does not need user-defined parameters. Features selection implies removing irrelevant and redundant features. CFS calculates a heuristic measure of the merit of a feature subset from pair-wise feature correlations and a formula adapted from test theory. The subset with the highest merit found during the search is reported. After running CFS, the selected features are classes, Density of comment lines, files, directories, File complexity, violations, duplicated blocks, lines, and critical violations.

Table 5 shows the performance of the prediction model after implementing the features selection which reduces around 43% of the features. The results reveal maintaining high accuracy in comparison to the model built based on all features. Similar observations regarding the confusion matrices as can be seen in Table 6.

It is noteworthy knowing the most influential static code metrics that contribute to predicting security vulnerabilities. The most influential metrics can be computed using gain ratio [31]. The gain ratio provides a normalized measure of the contribution of each feature to the classification. Table 7 reports the rank of the selected metrics. The higher the gain ratio, the more important the feature to predict security vulnerabilities.

Table 5. Random forest classification after feature selection

Precision	Recall	ROC Area	Class
0.844	0.846	0.941	**No**
0.941	0.571	0.802	**Low**
0.909	0.455	0.879	**Medium**
0.862	0.885	0.934	**High**
0.857	**0.856**	**0.934**	**Weighted avg.**

Table 6. Confusion matrix after features selection

High	No	Low	Medium	
679	86	1	1	**High**
91	499	0	0	**No**
8	4	16	0	**Low**
10	2	0	10	**Medium**

Table 7. Features with high gain ratio values

Metric	Gain ratio
comment_lines_density	0.1078
Lines	0.0973
Files	0.0839
Violations	0.0818
duplicated_blocks	0.0787
Classes	0.0763
Directories	0.071
critical_violations	0.0701
file_complexity	0.0318

5 Discussion

All the metrics used in this study are static which means developers and organizations can calculate them very easily. The features selected by the feature selection methodology are classes, Density of comment lines, files, directories, File complexity, violations, duplicated blocks, lines, and critical violations. The number of classes determines the size of the app, which plays an important role in determining the security of the app. More classes mean more attack surface. For Density of comment lines, it is very common in the software engineering community that if you have a very complex code, developers tend to write more comments to explain it. Files, directories, and lines have the same philosophy of the classes. For the file complexity, complexity is always the enemy of security [32]. Regarding duplicated blocks and critical violations, they are patterns found to be unhealthy for software systems especially security and privacy.

6 Conclusion

This papers presented an empirical study to examine the impact of static code metrics in predicting security vulnerabilities in Android applications. Three main research questions have been raised and answered in this research. The study results showed different levels of correlation among the code metrics. A direct influence for the code metrics in predicting security vulnerabilities was also observed. Moreover, these metrics could be ranked according to their contributions to providing high prediction rate. Feature selection algorithm played an important role in selecting the most influential metrics. The algorithm reduced around 43% of the static metrics which resulted in producing a lightweight, reliable prediction model.

As for future work, other static metrics could be considered, other tools with different ranking philosophies could be examined to enhance the ranking system and introduce prediction models with even higher accuracy.

References

1. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
2. Fernandez-Buglioni, E.: Security Patterns in Practice: Designing Secure Architectures Using Software Patterns. Wiley, Chichester (2013)
3. Wysopal, C., Nelson, L., Dustin, E., Zovi, D.D.: The Art of Software Security Testing: Identifying Software Security Flaws. Pearson Education (2006)
4. Antunes, N., Vieira, M.: Benchmarking vulnerability detection tools for web services. In: 2010 IEEE International Conference on Web Services (ICWS), pp. 203–210. IEEE (2010)
5. Abdlhamed, M., Kifayat, K., Shi, Q., Hurst, W.: Intrusion prediction systems. In: Information Fusion for Cyber-Security Analytics, pp. 155–174. Springer, Cham (2017)
6. Messier, R.: Intrusion detection systems. In: Network Forensics, pp. 187–209 (2017)
7. Cusumano, M.A.: Who is liable for bugs and security flaws in software? Commun. ACM **47**(3), 25–27 (2004)
8. Gorla, A., Tavecchia, I., Gross, F., Zeller, A.: Checking app behavior against app descriptions. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1025–1035. ACM (2014)

9. Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Using probabilistic generative models for ranking risks of Android apps. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 241–252. ACM (2012)
10. Rahman, A., Pradhan, P., Partho, A., Williams, L.: Predicting Android application security and privacy risk with static code metrics. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, pp. 149–153. IEEE Press (2017)
11. Syer, M.D., Nagappan, M., Adams, B., Hassan, A.E.: Studying the relationship between source code quality and mobile platform dependence. Softw. Qual. J. **23**(3), 485–508 (2015)
12. Corral, L., Fronza, I.: Better code for better apps: a study on source code quality and market success of Android applications. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, pp. 22–32. IEEE Press (2015)
13. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng. **31**(4), 340–355 (2005)
14. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. **33**(1), 2–13 (2007)
15. Chess, B., McGraw, G.: Static analysis for security. IEEE Secur. Priv. **2**(6), 76–79 (2004)
16. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Secur. Priv. **3**(1), 84–87 (2005)
17. Ko, C., Ruschitzka, M., Levitt, K.: Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In: IEEE Symposium on Security and Privacy, Proceedings, pp. 175–187. IEEE (1997)
18. Liu, M.Y., Traore, I.: Empirical relation between coupling and attackability in software systems: a case study on DOS. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, pp. 57–64. ACM (2006)
19. Howard, M., Pincus, J., Wing, J.M.: Measuring relative attack surfaces. In: Computer Security in the 21st Century, pp. 109–137. Springer, Heidelberg (2005)
20. Alhazmi, O.H., Malaiya, Y.K., Ray, I.: Measuring, analyzing and predicting security vulnerabilities in software systems. Comput. Secur. **26**(3), 219–228 (2007)
21. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 529–540. ACM (2007)
22. Shin, Y.: Exploring complexity metrics as indicators of software vulnerability. In: Proceedings of the 3rd International Doctoral Symposium on Empirical Software Engineering, Kaiserslautem, Germany (2008)
23. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Architect. **57**(3), 294–313 (2011)
24. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (2011)
25. Alves, H., Fonseca, B., Antunes, N.: Software metrics and security vulnerabilities: dataset and exploratory study. In: 2016 12th European Dependable Computing Conference (EDCC), pp. 37–44. IEEE (2016)
26. Campbell, G., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co., Greenwich (2013)
27. Dunham, K., Hartman, S., Quintans, M., Morales, J.A., Strazzere, T.: Android Malware and Analysis. CRC Press, Boca Raton (2014)
28. Montgomery, D.C., Runger, G.C.: Applied Statistics and Probability for Engineers. John Wiley & Sons, New York (2010)

29. Abunadi, I., Alenezi, M.: An empirical investigation of security vulnerabilities within web applications. J. UCS **22**(4), 537–551 (2016)
30. Czibula, G., Marian, Z., Czibula, I.G.: Software defect prediction using relational association rule mining. Inf. Sci. **264**, 260–278 (2014)
31. Hall, M.A., Holmes, G.: Benchmarking attribute selection techniques for discrete class data mining. IEEE Trans. Knowl. Data Eng. **15**(6), 1437–1447 (2003)
32. Lagerström, R., Baldwin, C., MacCormack, A., Sturtevant, D., Doolan, L.: Exploring the relationship between architecture coupling and software vulnerabilities. In: International Symposium on Engineering Secure Software and Systems, pp. 53–69. Springer, Heidelberg (2017)