An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics

Yonghee Shin
Department of Computer Science
North Carolina State University
Raleigh, NC 27695, U.S.A.
yonghee.shin@ncsu.edu

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695, U.S.A.
williams@csc.ncsu.edu

ABSTRACT

Complexity is often hypothesized to be the enemy of software security. If this hypothesis is true, complexity metrics may be used to predict the locale of security problems and can be used to prioritize inspection and testing efforts. We performed statistical analysis on nine complexity metrics from the JavaScript Engine in the Mozilla application framework to find differences in code metrics between vulnerable and non-vulnerable code and to predict vulnerabilities. Our initial results show that complexity metrics can predict vulnerabilities at a low false positive rate, but at a high false negative rate.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Complexity measures, Product metrics

General Terms

Measurement, Reliability, Security.

1. INTRODUCTION

One successful security attack can bring severe damages to people and organizations by allowing an attacker to access or modify confidential information or to launch a denial of service attack. Security attacks occur by exploiting vulnerabilities in a system. A software vulnerability is a weakness in a software system that allows an attacker to use the system for a malicious purpose. Buffer overflow, SQL injection, cross-site scripting are the representative vulnerabilities [4]. The weakness can come from design flaws, implementation errors, and configuration errors [7]. Early detection and mitigation of vulnerabilities in software artifacts can help to produce higher quality of software, to prioritize testing efforts, and to reduce the cost of later fixes [2] and potential damages to finances and trust.

A software fault is a latent error in code that causes a failure when the fault is executed. Faults and failures are known to be correlated with software complexity [1, 5]. Software vulnerability is also presumed to be correlated with software complexity [4]. Even though software vulnerabilities are considered a subset of software faults [7], the characteristics of vulnerabilities that differentiate them from faults in terms of complexity have not yet been investigated. Knowing the characteristics may help organizations to distribute time and resources to improve software security. While static analysis tools can detect certain patterns of vulnerabilities that those tools were designed for, complexity metrics can be used as a complementary way to find vulnerable locations in software artifacts that static analysis tools cannot detect and to direct further inspection and testing.

Our research objective is to identify the code complexity metrics that differentiate vulnerable functions from non-vulnerable functions and faulty-functions, and to investigate whether code metrics can be useful for vulnerability prediction.

We conducted a case study on the JavaScript Engine in the Mozilla application framework[1] to identify possible predictors of software vulnerabilities based on the nine complexity metrics we collected and to predict vulnerability-proneness using statistical analysis.

2. STUDY DESIGN

We pursue the answers of the following questions in this study.

Q1: Do the measures for complexity metrics for vulnerable and non-vulnerable functions demonstrate differences? If so, which complexity metrics can identify the differences?

Q2: Do the measures for complexity metrics for vulnerable and faulty-but-non-vulnerable functions demonstrate differences? If so, which complexity metrics can identify the differences?

Q3: Can we predict vulnerable functions from all functions using complexity metrics?

Q4: Can we predict vulnerable functions from faulty functions using complexity metrics?

Answering Q1 and Q3 helps prioritize the security efforts according to the measures of complexity metrics identified as best differentiating factors and the code locations predicted as being vulnerable. Answering Q2 and Q4 helps to determine whether verification and validation (V&V) teams should have different prioritization and strategy to find faults and vulnerabilities.

ESEM'08, October 9-10, 2008, Kaiserslautern, Germany.
Copyright 2008 ACM 978-1-59593-971-5/08/10...$5.00.

[1] http://www.mozilla.org

2.1 Metrics

We collected the following nine code complexity metrics using a metrics tool, Understand C++[2]. McCabe's cyclomatic complexity [3] counts the number of decision statements plus one. Modified cyclomatic complexity and strict cyclomatic complexity are the same as McCabe's cyclomatic complexity except for the differences in the way that `switch` statements and conditions in decision statements are counted. Essential cyclomatic complexity measures the structuredness of code by counting cyclomatic complexity after iteratively replacing all structured programming primitives to a single statement. Nesting complexity measures the deepest level of nested control constructs. Paths are the number of possible paths. SLOC is the number of lines of source code excluding comments. SLOC_exe is the number of SLOC excluding declarations. Stmt_exe is the number of executable statements.

2.2 Project Under Study

We chose the Mozilla Java Script Engine (JSE) open source for our case study because the source code, faults, and vulnerability information are publicly available and the amount of faults and vulnerabilities reported were enough for our initial analysis. The faults in the Mozilla JSE can be found from the Bugzilla[3], a bug tracking system. The mitigated vulnerabilities in the Mozilla JSE are posted to the Mozilla Foundation Security Advisories (MFSA)[4]. The total number of bug reports in Bugzilla was 51370 (as of 29th, February, 2008) and the number of bug reports linked from MFSAs was 458. The percentage of vulnerabilities among the total bug reports is around 0.89%. There were 106 bug reports on JSE linked from MFSAs and 15 of them were not accessible from the bug reports due to the security policy of Mozilla project, and therefore, could not be included in our analysis.

We chose six versions of JSE among 51 available versions. For the variety of our samples, we chose two minor versions from each major version; v1.0.2, v1.0.7, v1.5, v.1.5.0.8, v.2.0, and v.2.0.0.4. JSE v1.0.2 has 83 files and 78 KSLOC. JSE v1.5 has 85 files and 96 KSLOC. JSE v2.0.0.4 has 88 files and 107 KSLOC. The number of functions in the six versions of JSE is between 1352 and 1862. To identify faulty and vulnerable functions, we counted the number of functions that were changed due to faults and vulnerabilities. We know in which version a vulnerability was fixed, however, we do not know when the vulnerability was introduced. Therefore, when there is a vulnerability fix in a version (e.g. v1.5.0.8), we assumed that the same vulnerability existed in all the previous versions (e.g. v1.5.0.7, v1.5.0.6, etc.). After all, 0.8% to 17.5% of functions were changed due to identified faults, and 0.6% to 9.3% of functions due to vulnerabilities.

2.3 Analysis Methods

We compared the complexity measures of vulnerable and non-vulnerable functions to answer Q1 and compared the complexity measures of vulnerable and faulty-but-non-vulnerable functions to answer Q2. For the comparison, we used the Wilcoxon rank sum test [6], a non-parametric test that is not sensitive to outliers and does not assume any distribution of sample data.

To answer Q3 and Q4, we performed binary logistic regression analysis [6]. Binary logistic regression analysis is a way to classify data into two groups depending on the probability of an occurrence of an event for given values of independent variables. In our case, logistic regression analysis computes the probability that a function is vulnerable for given complexity measures. A function with probability of vulnerability greater than a certain cutoff point (0.5 in our case study) is classified as vulnerable.

The quality of prediction using a logistic regression model can be measured in terms of accuracy, a false positive rate (Type I error) and a false negative rate (Type II error). The accuracy measures the degree of overall correct classification. The false positive (FP) rate measures the rate of falsely classified functions as vulnerable among the non-vulnerable functions. The false negative (FN) rate measures the rate of falsely classified functions as non-vulnerable among the vulnerable functions. A high false positive rate indicates that effort may be wasted in finding vulnerabilities when there are none. A high false negative rate indicates that there is a risk of overlooking vulnerabilities. The three quality criteria are defined in the following formula:

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+TN+FN},$$

$$\text{FP rate} = \frac{FP}{FP+TN}, \quad \text{FN rate} = \frac{FN}{TP+FN}$$

To measure the efficacy of using complexity metrics in predicting vulnerabilities, we performed next version validation. Next version validation is performed by testing the version n+1 using the model trained from the version n. This way, we can measure the model's ability to predict vulnerabilities in the current version using the model built from the previous version.

3. RESULTS & CONCLUSION

To answer Q1 and Q2, we performed the Wilcoxon rank sum test. The results show that the measures of complexity for the vulnerable functions and the non-vulnerable functions in JSE are significantly different at the 0.05 significance level for all the six versions. The measures of complexity for the vulnerable functions and the faulty-but-non-vulnerable functions in the three older versions of JSE (v1.0.2, v1.0.7, and v.1.5) were also significantly different in the nine complexity metrics. However, in the later versions of JSE (v1.5.0.8, v2.0 and v.2.0.0.4), the measures of complexity for the vulnerable functions and the faulty-but-non-vulnerable functions were significantly different only in the nesting complexity. This result indicates that nesting complexity can be a differentiating factor of vulnerable functions from faulty functions in JSE. Therefore, giving more attention to highly nested functions than to other functions in security inspection could be an efficient strategy.

To answer question Q3 and Q4, we performed the following three experiments using logistic regression analysis.

- E1: Predict faulty functions from all functions.
- E2: Predict vulnerable functions from all functions.
- E3: Predict vulnerable functions from faulty functions.

[2] http://www.scitools.com/

[3] http://bugzilla.mozilla.org

[4] http://www.mozilla.org/projects/security/known-vulnerabilities.html

The dependent variables of our logistic regression models are fault-proneness (E1) and vulnerability-proneness (E2 and E3). The independent variables were chosen by the stepwise regression method [6], a method to systematically select independent variables that are highly significant. We built 15 models in total for five versions of JSE in the three experiments. A model was built for each version except the last version (v2.0.0.4) in each experiment and then the model was applied to the next version to predict vulnerabilities. The training data for the model of v1.0.2 were used to test the model because there is no previous model for v1.0.2. In all the cases, nesting complexity was consistently chosen as an independent variable. SLOC complexity was the next frequently chosen variable. Table 1 shows the accuracy, false positives rates, and false negative rates in the three experiments.

Table 1. Predictive power of the logistic regression models

		v1.0.2	v1.0.7	v1.5	v1.5.0.8	v2.0	v2.0.0.4
E1	Accuracy	84.76	85.01	83.75	89.62	90.96	98.39
	FP rates	1.15	1.15	1.30	3.43	0.42	1.03
	FN rates	85.40	85.20	86.73	80.39	93.02	78.57
E2	Accuracy	90.98	91.36	91.13	96.34	96.81	99.52
	FP rates	0.90	0.89	0.98	1.63	0.06	0.00
	FN rates	88.10	86.18	89.93	79.55	95.08	81.82
E3	Accuracy	65.49	65.92	62.59	62.75	64.53	42.86
	FP rates	48.00	48.00	58.62	36.70	1.80	0.00
	FN rates	23.81	22.76	16.78	38.64	96.72	72.73

The prediction results show that the overall accuracies are very high for E1 and E2, and fairly high for E3. The low false positive rates for E1 and E2 indicate that once our model predicts faults and vulnerabilities, those predicted ones are likely to be true vulnerabilities. Furthermore, those faults and vulnerabilities can be identified at an early development phase from code before testing. However, the high false negative rates for E1 and E2 indicate that our model can miss a large portion of faults and vulnerabilities. Therefore, the current model is useful to identify the initial locations for inspecting and testing, and should be used as a complementary way of other techniques and tools for vulnerability detection. One of the possible reasons could be because the complexity is associated with vulnerabilities only at above a certain threshold. The other possible reason is because we did not differentiate the functions changed for the direct reason of vulnerabilities and the functions changed as a secondary reason identified from vulnerabilities such as an addition of a parameter to several functions with low complexity. Giving different weights in the simple changes propagated from the main changes due to vulnerabilities might lead to more precise results. E3 showed comparatively high false positives and low false negatives. This result indicates that average complexity measures for the vulnerable functions and the faulty-but-non-vulnerable functions are different as explained previously, but the difference is not as big as the vulnerabilities can be predicted precisely.

The variations in predictability between v1.0.2, v1.0.7, and v1.5 were very small. The small variations in predictability between versions indicate that a model built from a previous version can be reliably used to predict vulnerabilities for the next version. The large variations in predictability in the later versions might be because the number of vulnerabilities was small to obtain statistically significant results. The later versions have less vulnerabilities than the older versions because we assumed that all the previous versions of a function have the same vulnerability when the function has a vulnerability. Another reason that the later versions have less vulnerabilities than the older versions might be because the vulnerabilities have not been discovered yet. However, because vulnerability prediction is for a preventive action, not for a reactive action after the vulnerabilities are found, performing V&V based on the results of our prediction model still can be helpful. Prediction of vulnerable functions from all functions provides slightly better predictability than prediction of faulty functions from all functions, showing that inspecting vulnerabilities using code complexity is as good approach as inspecting faults using code complexity.

To conclude, vulnerable functions have distinctive characteristics from non-vulnerable functions and from faulty-but-non-vulnerable functions in code complexity. Nesting complexity was the best distinguishing factor among the nine complexity metrics in JSE. Prediction of vulnerabilities from source code using complexity metrics is a feasible approach with low false positives, however, still misses many vulnerabilities. We will extend our study to reduce false negatives considering the code changes due to the main effects of vulnerabilities and the secondary effects of vulnerabilities. We will also find better metrics including design level metrics for better prediction. As noted in previous studies [5], our results might not be generalized to other projects.

4. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Grant No. 0716176. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

5. REFERENCES

[1] Basili, V. R., Briand, L. C., and Melo, W. L., "A Validation of Object-Oriented Design Metrics as Quality Indicators," IEEE Transactions on Software Engineering, vol. 22, no. 10, pp. 751 - 761, October, 1996.

[2] Boehm, B. W., Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall Inc., 1981.

[3] McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320, 1976.

[4] McGraw, G., Software Security: Building Security In. Boston, NY: Addison-Wesley, 2006.

[5] Nagappan, N., Ball, T., and Zeller, A., "Mining Metrics to Predict Component Failures," in Proceedings of Proceedings of the 28th international conference on Software engineering Shanghai, China, May 20-28, 2006, pp. 452-461.

[6] Ott, R. L. and Longnecker, M., An Introduction to Statistical Methods and Data Analysis, 5th edition: Duxbury, 2001.

[7] Viega, J. and Mcgraw, G., Building Secure Software. Boston, NY: Addison-Wesley, 2002.