Exploring Complexity Metrics as Indicators of Software Vulnerability

Yonghee Shin
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
yonghee.shin@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. Our goal is to investigate the effect of software complexity to security problems and to identify metrics that represent the complexity that lead to security problems. We will use statistical analysis and machine learning to build models to capture the relationship between complexity and security problems using open source and industrial projects. The initial case study on the JavaScript Engine in the Mozilla application framework shows that our initial modeling and complexity metrics need to be improved, but our approach is a feasible means of identifying vulnerable areas of code.

Categories and Subject Descriptors

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

General Terms

Measurement, Reliability, Security.

Keywords

Software metrics, security metrics, complexity, reliability, fault prediction, vulnerability prediction

1. INTRODUCTION

The wisdom of security experts is that complexity leads to software security problems [9, 14, 20, 32]. Geer [9] stated that that "complexity provides both opportunity and hiding places for attackers" and "security failures come from it [complexity] as surely as dawn comes from the east" in a hearing at a subcommittee of the Committee on Homeland Security on 23rd, April, 2007. McGraw [20] also argued that the three major causes of security problems are complexity, connectivity, and extensibility. The wisdom of these experts, though, has not been substantiated by empirical evidence in the sense that complexity has not been measured in an elaborated way to identify the vulnerable locations and appropriate resource distribution.

Conference'04, Month 1–2, 2004, City, State, Country.

The study of the relationship between complexity and vulnerabilities is similar to the study of the relationship between complexity and faults. A fault is an accidental condition that causes a functional unit to fail to perform its required function [13]. A software vulnerability is an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [18]. Previous studies have shown complexity is related with software faults [15-17, 26, 34, 37, 38]. However, other studies argue that the currently-known complexity metrics seem not to be good indicators of faults [5, 7, 33]. Jaquith [14] agreed that "the relationship of complexity metrics to security is hypothetical rather than proven". He, however, recommends to use code complexity combined with additional environment metrics to obtain a true picture of risk. Even though it is early to conclude that complexity is a direct cause of faults, software complexity affects cognitive complexity, and cognitive complexity is a direct cause of faults [3] by making understanding, testing and maintaining code difficult. The previous studies [15-17, 26, 34, 37, 38] can be replicated in the context of vulnerabilities. However, the results might not be necessarily the same as fault prediction, because even though vulnerabilities are a subset of faults [35], the differences in the characteristics of vulnerable code and faulty code have not been investigated yet.

Our research objective is to determine if complex code is less secure and, if so, what metrics can be used to embody the type of complexity that tends to lead to decreased security.

We will use statistical analysis and machine learning methods to predict vulnerabilities on the available code level and design level metrics. We will also devise new metrics including architectural level metrics, as necessary.

We conducted a case study on the JavaScript Engine of the Mozilla application framework[1] to identify possible predictors of software vulnerabilities from nine complexity metrics. Our initial results show that complexity metrics can predict vulnerabilities at a low false positive rate (e.g. 0.9% for JSE v1.0.2), but at a high false negative rate (e.g. 88.0% for JSE v1.0.2). For more evidence about the relationship between complexity and security, we will perform more analysis using various models and metrics.

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

2. BACKGROUND AND RELATED WORK

This section defines the terms that we will use in this paper, provides background on complexity metrics, and introduces prior studies on fault and vulnerability prediction.

2.1 Definitions

The literature has used the definitions of terms related with faults in a similar but inconsistent way [8, 13, 20]. The situation is the same for the definition of vulnerabilities [2, 12, 20]. To avoid confusion, we provide the definitions of fault and vulnerability related terms that will be used in this paper. An *error* is a human mistake that causes a fault in software [8, 13]. A *fault* is an encoded human error in software artifacts that causes a failure when it is executed [8, 13]. A *bug* is a fault in code. A *failure* is a deviation of a system from its required behavior [8, 13]. A fault causes a failure if the fault is executed.

Security is defined as "the quality or state of being secure" or "freedom from danger"[2]. A *security policy* is "a statement of what is, and what is not, allowed" [2]. Computer security has three aspects to be secured: confidentiality, integrity, and availability [2]. *Security violation* occurs when confidentiality, integrity, and availability are damaged. A *threat* is a potential violation of security [2]. An *attack* is a violation of security [2]. A *vulnerability* is **"**a weakness that makes it possible for a threat to occur" [2]. In other words, an attack is an exercised threat exploiting vulnerabilities. In the context of software systems, a software vulnerability is an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [18].

2.2 Complexity Metrics

Software complexity can be measured in four aspects: problem complexity, algorithmic complexity, cognitive complexity, and structural complexity [8]. Even though the four complexities are interrelated, the most interested complexity in this research is structural complexity that we can mine from software artifacts such as code. Many structural complexity metrics have been proposed in code and design levels. The representative examples of code level complexity metrics are Halstead's software science [11] and McCabe's cyclomatic metrics [19] that measure complexity in terms of lexical tokens and control constructs, respectively. The design level complexity metrics include cohesion and coupling. Cohesion metrics measure the relationships among the elements within a single module [22]. Coupling metrics measure the relationships between modules [22]. Object-oriented metrics [4] include a set of metrics for object-oriented systems including code level complexity, cohesion, coupling, and inheritance complexity.

2.3 Predicting Failures using Complexity Metrics

Zhao et al. [37] compared the failure prediction ability of software design and code metrics that count the number of constructs including the number of variables and the number of if-statements. Their case study on an industrial project show that "design metrics are as good as code metrics" and can be used as an early indicator of failures.

Subramanyam and Krishnan [34] investigated the effects of object-oriented design complexity metrics on software defects. In their study, class size, WMC (Weighted Methods per Class), and DIT (Depth of Inheritance) are positively correlated with defects in both C++ and Java programs. However, increase of defects in increase of CBO (Coupling between Objects) was shown only in the C++ programs, not in the Java programs.

Nagappan et al. [26] performed a study on finding a subset of complexity metrics that predict post-release failures. They chose combined metrics that best predict failures in five major Microsoft project components including the Internet Explorer 6 using principal component analysis. They found combined sets of complexity metrics to predict failures in the projects, and also found that no single metric that can be applied to all projects to predict failures.

Zimmermann et al. [38] performed a study on predicting defects in Eclipse[3] using complexity metrics at the file and package levels. They performed logistic regression on their data. In the study, 20% to 40% of defect-prone files were correctly identified and around 60% of defect-prone files predicted were true positives.

2.4 Predicting Security Vulnerabilities

Failure prediction has been the subject of much research in software engineering, however, security vulnerability prediction is a fairly new area. Some recent research on vulnerability prediction has been conducted.

Gegick et al. [10] investigated the relationship between attack-proneness and internal metrics including code churn, SLOC and alerts from automated static analysis (ASA) on a large commercial telecommunication software system. Attack-proneness is the likelihood that a software component can be exploited. They built a model to predict attack-prone components using recursive partitioning and logistic regression analysis. The results show that the combined usage of the churn and ASA alerts provides high predictive power with 8% of false positive rate and 0 % of false negative rate, however, each metric could not predict attack-prone components by itself with enough statistical significance.

Alhazmi, et al. [1] investigated whether the number of vulnerabilities latent in a software system can be predicted or not. They analyzed the Windows and Red Hat Linux operating systems and observed that the ratio of vulnerabilities to the total number of defects falls in the range of 1% to 5% and the percentage coincides with the data from other literature. This indicates that known vulnerability densities can be a useful predictor of unknown vulnerabilities, especially for similar products. Their approach can be useful for estimating the effort required to identify and correct undiscovered security vulnerabilities, but cannot identify the location of the vulnerabilities in the source code.

Neuhaus et al. [28] introduces a tool to automatically map vulnerability reports to vulnerable components in open source Mozilla code using vulnerability reports, bug identifiers in the vulnerability reports, change history in version archive. They

[2] http://www.merriam-webster.com/

[3] http://www.eclipse.org/

observed common patterns of imports (`#include`) and function calls in vulnerable components using pattern mining techniques. Their tool identified 45% of vulnerable components of all vulnerable components (45% recall) using import analysis, and 70% of all identified vulnerable components were true vulnerable components (70% precision) using function call analysis. They performed the analysis at the component level, while our study is at the function level using complexity metrics.

3. HYPOTHESES

This section describes our research questions and the hypotheses.

3.1 Research Questions

We pursue to answer the following questions in this research.

Q1: Is high complexity a cause of software vulnerability? If so, we need to find proper measures for complexity that embody this relationship.

Q2: What metrics can be used to capture the type of complexity that leads to vulnerabilities early in the development lifecycle? If a metric is available early in the development lifecycle such as design or coding phase, we can prioritize vulnerability inspection as soon as design or code is available.

Q3: Do vulnerability fixes induce more complexity? If so, it seems a self-contradiction because complexity associated with a vulnerability is increased again by fixing the vulnerability.

3.2 Research Hypotheses

From the research questions in Section 3.1, we formulated the following research hypotheses based on literature and the author's experience.

H1: More complex programs have more vulnerabilities.

H2: Complexity metrics can predict vulnerabilities at a statistically significant level.

H3: Modules with vulnerabilities have different complexity from modules with faults at a statistically significant level. Therefore, complexity metrics that can explain faults are not necessarily can also explain vulnerabilities at a statistically significant level.

H4: Vulnerability fixes induce more complexity.

4. METHODOLOGY

This section describes the data we need, the methods to collect the data, and the modeling and evaluation methods that will be used in this research.

4.1 Data Gathering

For this research, we need four types of data: fault reports; vulnerability reports; source code change history; and traceability from vulnerabilities to the original and changed code of an *entity*. An entity can be a function, a file, a class, or a component.

A representative fault reporting system is the Bugzilla[4]. The Bugzilla includes description on identified faults, related components, developers, and current status of fault fixes and verification. The Bugzilla also provides a link to modified code for each fault fix. Two representative databases for publicly-known vulnerabilities are the Common Vulnerabilities and Exposures (CVE)[5] and the National Vulnerability Database[6]. Some projects such as Mozilla have their own vulnerability reports[7]. A commonly used system to manage versions and code changes is the Concurrent Versions System (CVS)[8]. The CVS system also allows developers to record additional information such as developer names and the reasons of code changes. We identify the modules with faults and vulnerabilities by tracking the code changed due to faults and vulnerabilities. For the repeated process, all of the process should be automated. We will develop a tool to collect the necessary information from fault reports, vulnerability reports and source version control systems.

For complexity metrics, we will gather complexity measures using existing tools, such as Understand C++[9] and McCabe IQ[10]. We will also devise new metrics, if necessary. Our initial interests are in code and design level complexity. Each level of complexity (code, design, or architectural level) has different perspective and might have a complementary role in predicting vulnerabilities. However, the complexity measures obtainable earlier help organizations start earlier security inspection.

We will perform our analysis on various granularities including function, file, class, and component levels. Zimmermann et al.'s study [38] reveals that the degree of correlation between faults and complexity measures is different depending on the granularity. Therefore, using various granularities might reveal more information on the relationship between complexity and vulnerabilities.

4.2 Model Building

We are interested in vulnerability-proneness, the predicted ranking of the entities from most vulnerable to least vulnerable, and the differences in complexity between vulnerable and faulty entities.

To predict vulnerability-proneness, we will use statistical analysis and machine learning methods. A statistical analysis method that is often used for fault classification is logistic regression. Logistic regression [31] 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 an entity is vulnerable for given complexity measures. An entity with probability of vulnerability greater than a certain cutoff point (e.g. 0.5) is classified as vulnerable.

The machine learning methods that have been used for fault-proneness include decision tree methods, bagging, boosting, Naïve Bayes, and Bayesian networks [6, 15, 21, 27].

In decision tree methods, each node has a rule that classifies entities based on the measures of a metric. An example rule is "if cyclomatic complexity is greater than 10, an entity is vulnerable". The rule to be used at each node of a decision tree is determined in

[4] http://www.bugzilla.org

[5] http://cve.mitre.org/

[6] http://nvd.nist.gov

[7] http://www.mozilla.org/projects/security/knownvulnerabilities.html

[8] http://www.nongnu.org/cvs/

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

[10] http://www.mccabe.com

a way that the vulnerable and non-vulnerable entities can be classified as soon as possible from the root of the decision tree [36].

Because decision trees can be unstable in terms that slight change in training data may result in a different model, bagging and boosting [36] can be used. Bagging and boosting combine output from different models and determine the vulnerability-proneness by voting. When the number of models that predict an entity to be vulnerable is larger than the number of models that predict the entity to be non-vulnerable, the entity is finally predicted to be vulnerable. The difference in bagging and boosting is whether models are built separately (bagging) or a new model is influenced by the performance of previous models (boosting).

A Bayesian network [36] is a probabilistic graphical model for representing the interaction between variables. Each node corresponds to a variable and the influence from variable X to variable Y is represented as a conditional probability P(Y|X). In our case, a Bayesian network can be used to compute the probability that an entity will be vulnerable given measures of complexity metrics.

To predict the ranking, multiple linear regression [17, 23-26] and negative binomial regression [30] have been used to predict the number of faults. We will use these methods for vulnerability prediction. Bagging and boosting also can predict the number of vulnerabilities by calculating average of complexity measures instead of taking a vote [36].

To compare the difference in complexity between vulnerable and faulty entities, we will use Wilcoxon rank sum test [31], a non-parametric test that is not sensitive to outliers and does not assume any distribution of sample data.

4.3 Evaluation

The performance of a fault prediction model can be measured in several ways. Most frequently used measures are Accuracy, Recall, Precision, a false positive rate (Type I misclassifications), a false negative rate (Type II misclassifications), and the percentage of faults contained in the top-ranked entities (TP rate in top-ranks). Among these measures, Ostrand and Weyuker [29] argue that Type II misclassifications and a TP rate in top-ranks are the most important measures. These measures are also effective for vulnerability prediction models.

The false positives (FP) are the entities predicted as vulnerable when they are not vulnerable. The false negatives (FN) are the entities predicted as non-vulnerable when they are vulnerable. The Accuracy measures the overall correct classification rate. The Recall measures the rate of true positives correctly classified by a prediction model among all the actual positives. The Precision measures the rate of true positives classified correctly by a prediction model among all the positives classified by the model. The false positive rate measures the rate of falsely classified positives among the actual negatives. The false negative rate measures the rate of falsely classified negatives among the actual positives. 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 measures are defined in the following formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \quad (1)$$

$$\text{Precision} = \frac{TP}{TP + FP} \quad (2)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3)$$

$$\text{FP rate} = \frac{FP}{FP + TN} \quad (4)$$

$$\text{FN rate} = \frac{FN}{TP + FN} \quad (5)$$

$$\text{TP rate in top-ranks} = \frac{TP(x)}{TP + FN} \quad (6)$$

, where TP(x) is the number of true positives in the top x % of entities ranked in the order of predicted vulnerabilities.

The measures from (1)-(5) treat the entities with vulnerabilities the same regardless how many vulnerabilities exist in each entity. On the other hand, a TP rate in top-ranks can provide additional information about the effectiveness of a model by considering the number of predicted vulnerabilities in each entity.

To validate the model's efficacy, we will perform cross-validation and next release validation. The cross-validation is performed when only a single data set is available by splitting data into several bins and by using one of the bins as a testing data set and the remaining bins as training data sets for a model. Then the model is applied to the testing data. This way, the model's performance can be evaluated on the data set that was not used to train the model and can simulate the vulnerability prediction on a random population. When several releases are available, we will perform next release validation. Next release validation is performed by testing the Release N using the model trained from the Release N-1. If the model's performance is consistently good throughout past several releases, the model may also provide good performance for the near future releases.

4.4 Threats to Research

Our research relies on discovered and reported vulnerabilities for the validation of our approach rather. Vulnerabilities that have not yet been discovered or have not been not publicly announced cannot be used for our study even though the information is very important for precise modeling. Therefore, our research can provide only a partial view about the effects of complexity on security. Even in the case we are not able to find an appropriate model to capture the relationships between vulnerabilities and complexity, our experience will add knowledge to research bodies and provide a foundation for further research.

5. A CASE STUDY

We performed a case study on six versions of JavaScript Engine (JSE) in the Mozilla application framework. The faults in the Mozilla JSE can be found from the Bugzilla[11], a bug tracking system. The mitigated vulnerabilities in the Mozilla JSE are

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

posted to the Mozilla Foundation Security Advisories (MFSA)[12]. Each MFSA has a link to one or more bug reports, each of which has detailed information on a vulnerability. The total number of bug reports in Bugzilla was 51,370 (as of 29th, February, 2008). Among them, 106 bug reports were for vulnerabilities in JSE.

To answer Q2, we performed logistic regression analysis [31] on nine code complexity metrics from JSE. The nine code complexity metrics include McCabe's cyclomatic complexity [19], modified cyclomatic complexity, strict cyclomatic complexity, essential cyclomatic complexity, nesting complexity (depth of control constructs), SLOC (source lines of code), executable SLOC, and the number of statements. The independent variables were selected by using stepwise regression method [31], a method to systematically select independent variables that are highly significant. Our initial results show that the nesting complexity is the best predictor in JSE. In all models for all versions, nesting complexity has been constantly selected as a significant independent variable. The models predicted vulnerabilities at low false positive rates (e.g. 0.9% for JSE v1.0.2), but at high false negative rates (e.g. 88.0% for JSE v1.0.2). This results indicates that we need to find better metrics than the ones used in this case study that can capture the relationship between vulnerabilities and complexity and use other models if necessary.

6. SUMMARY AND RESEARCH PLAN

This paper provided three research questions and six hypotheses on explaining the influence of software complexity to vulnerabilities. We provided the methodology and evaluation methods, and a case study we have conducted.

This research will be performed on open source and industrial projects using the following steps:

Step 1. Complexity metrics will be gathered on each project using existing metrics tools described in Section 4.1.

Step 2. Vulnerabilities and code changed to fix vulnerabilities for the project will be gathered.

Step 3. Vulnerability prediction models will be built using statistical analysis and machine learning methods described in Section 4.2 to predict vulnerability-proneness, ranking, and the difference in complexity between faulty and vulnerable entities.

Step 4. The results will be evaluated according to the evaluation criteria described in Section 4.3.

If necessary, we will devise new metrics and repeat the Step 3 and 4 on the new metrics.

The expected contributions of this research are:

- To show the empirical evidence whether software complexity is the enemy of software security or not and, if so, to provide the models to predict vulnerabilities.
- To collect complexity measures and the links from vulnerabilities to code changed due to the vulnerabilities, and provide them for public use (in the case of open source).
- To create new metrics for vulnerability prediction.
- To provide a tool for mining source code and vulnerability database repository to collect metrics and code changed to fix vulnerabilities.

Nagappan et al. [26] states that "there is no single set of metrics that fits all projects" and "predictors are accurate only when obtained from the same or similar projects". Therefore, the results of our research might not be able to be generalized beyond the projects we use. However, we will collect metrics from as many projects as possible and provide the results.

We only consider vulnerabilities reported at the time that the models were built. Therefore, our models may not reflect all the lurking vulnerabilities at the time of analysis that could be found in the future. However, this limitation is the same for all the fault prediction models, which based only on identified faults. Therefore, our models should be interpreted as one of the indicators of vulnerabilities rather than used as a sole guidance.

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

7. 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.

8. REFERENCES

[1] Alhazmi, O. H., Malaiya, Y. K., and Ray, I., "Measuring, Analyzing and Predicting Security Vulnerabilities in Software Systems," *Computers & Security*, vol. 26, no. 3, pp. 219-228, May, 2007.

[2] Bishop, M., *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.

[3] Briand, L. C., Wüst, J., Ikonomovski, S., and Lounis, H., "A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: an Industrial Case Study," International Software Engineering Research Network, 1998.

[4] Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object-Oriented Design," *IEEE Transactions in Software Engineering*, vol. 20, no. 6, pp. 476-493, June, 1994.

[5] Emam, K. E., Benlarbi, S., Goel, N., and Rai, S. N., "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630 - 650, July, 2001.

[6] Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., and Mishra, R., "Predicting Software Defects in Varying Development Lifecycles using Bayesian Nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32-43, January, 2007.

[7] Fenton, N. E. and Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Transactions in Software Engineering*, vol. 26, no. 8, pp. 797 - 814, August, 2000.

[8] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*, 1997.

[9] Geer, D. E., "A witness testimony in the Hearing, Wednesday 25 April 07, entitled Addressing the Nation's Cybersecurity Challenges: Reducing Vulnerabilities Requires Strategic Investment and Immediate Action, submitted to the Subcommittee on Emerging Threats, Cybersecurity, and Science and Technology," 2007.

[10] Gegick, M., Williams, L., and Osborne, J., "Predicting Attack-prone Components with Internal Metrics," NCSU CSC TR 2008-8.
[11] Halstead, M. H., *Elements of Software Science*. New York: Elsevier, 1977.
[12] Howard, M. and LeBlanc, D., *Writing Secure Code*, 2nd ed.: Microsoft Process, 2003.
[13] IEEE, "IEEE Std 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June 9, 1988.
[14] Jaquith, A., *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Upper Saddle River, NJ: Pearson Education, Inc, 2007.
[15] Jiang, Y., Cukic, B., Menzies, T., and Bartlow, N., "Comparing Design and Code Metrics for Software Quality Prediction," in Proceedings of *the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, Leipzig, Germany, May 12–13, 2008, pp. 11-18.
[16] Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., and Goel, N., "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, vol. 13, no. 1 pp. 65 - 71, Jan.,1996.
[17] Khoshgoftaar, T. M. and Munson, J. C., "Predicting Software Development Errors Using Software Complexity Metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253 - 261, Feburary, 1990.
[18] Krsul, I. V., *Software Vulnerability Analysis*, PhD Thesis, West Lafayette, Purdue University, 1998.
[19] McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, 1976.
[20] McGraw, G., *Software Security: Building Security In*. Boston, NY: Addison-Wesley, 2006.
[21] Menzies, T., Greenwald, J., and Frank, A., "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions in Software Engineering*, vol. 33, no. 1, pp. 2 - 13, January, 2007.
[22] Myers, G. J., *Composite/Structured Design*. New York: Van Nostrand Reinhold Company, 1978.
[23] Nagappan, N. and Ball, T., "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," in Proceedings of *the 27th International Conference on Software Engineering*, St. Louis, MO. USA, 2005, pp. 580 - 586.
[24] Nagappan, N. and Ball, T., "Use of Relative Code Churn Measures to Predict System Defect Density," in Proceedings of *the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, USA, May 15-21, 2005, pp. 284 - 292.
[25] Nagappan, N., Ball, T., and Murphy, B., "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures," in Proceedings of *the 17th International Symposium on Software Reliability Engineering*, Raleigh, NC, U.S.A., November 7-10, 2006, pp. 62-74.
[26] Nagappan, N., Ball, T., and Zeller, A., "Mining Metrics to Predict Component Failures," in Proceedings of *the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 20-28, 2006, pp. 452-461.
[27] Neil, M. and Fenton, N. E., "Predicting Software Quality Using Bayesian Belief Networks," *21st Ann. Software Eng. Workshop*,December, 1996.
[28] Neuhaus, S., Zimmermann, T., and Zeller, A., "Predicting Vulnerable Software Components," in Proceedings of *the 14th ACM Conference on Computer and Communications Security (CCS'07)*, Alexandria, Virginia, USA, October 29–November 2, 2007, pp. 529 - 540.
[29] Ostrand, T. J. and Weyuker, E. J., "How to Measure Success of Fault Prediction Models," in Proceedings of *the Fourth International Workshop on Software Quality Assurance (SOQUA '07): in Conjunction with the 6th ESEC/FSE Joint Meeting*, Dubrovnik, Croatia, 2007.
[30] Ostrand, T. J., Weyuker, E. J., and Bell, R. M., "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions in Software Engineering*, vol. 31, no. 4, pp. 340 - 355, 2005.
[31] Ott, R. L. and Longnecker, M., *An Introduction to Statistical Methods and Data Analysis*, 5th ed. Pacific Grove: Duxbury, 2001.
[32] Perrow, C. B., "Complexity, Catastrophe, and Modularity," *Sociological Inquiry*, vol. 78, no. 2, pp. 162-173, May, 2008.
[33] Shepperd, M. and Ince, D. C., "A Crique of Three Metrics," *Journal of Systems Software*, vol. 26, no. 3, pp. 197-210, September, 1994.
[34] Subramanyam, R. and Krishnan, M. S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297-310, April, 2003.
[35] Viega, J. and Mcgraw, G., *Building Secure Software*. Boston, NY: Addison-Wesley, 2002.
[36] Witten, I. H. and Frank, E., *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Boston: Morgan Kaufmann Publishers, 2005.
[37] Zhao, M., Wohlin, C., Ohlsson, N., and Xie, M., "A Comparison between Software Design and Code metrics for the Prediction of Software Fault Content," *Information and Software Technology*, vol. 40, no. 14, pp. 801-809, 1998.
[38] Zimmermann, T. and Zeller, R. P. A., "Predicting Defects for Eclipse," in Proceedings of *the Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)*, Minneapolis, MN, 2007, pp. 9-15.