Toward the Use of Automated Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components

Michael Gegick and Laurie Williams
Department of Computer Science, North Carolina State University
{mcgegick, lawilli3}@ncsu.edu

Abstract

Extensive research has shown that software metrics can be used to identify fault- and failure-prone components. These metrics can also give early indications of overall software quality. We seek to parallel the identification and prediction of fault- and failure-prone components in the reliability context with vulnerability- and attack-prone components in the security context. Our research will correlate the quantity and severity of alerts generated by source code static analyzers to vulnerabilities discovered by manual analyses and testing. A strong correlation may indicate that automated static analyzers (ASA), a potentially early technique for vulnerability identification in the development phase, can identify high risk areas in the software system. Based on the alerts, we may be able to predict the presence of more complex and abstract vulnerabilities involved with the design and operation of the software system. An early knowledge of vulnerability can allow software engineers to make informed risk management decisions and prioritize redesign, inspection, and testing efforts. This paper presents our research objective and methodology.

1. Introduction

If you cannot measure it, you cannot improve it.
-- Lord Kelvin (1824-1907)

CERT/CC[1] reports an overall 450% increase in the number of vulnerabilities reported between 2000 and 2005. Despite increased focus on new techniques for computer system security, the number of vulnerabilities reported increased 35% from 2005 to 2006. Rather than gaining a foothold on the security problem through our increased focus, we continue to lose ground. We must build better software.

[1] http://www.cert.org/stats/cert_stats.html

Extensive research, including [29, 31-37, 42], has shown that software metrics can be used to identify fault- and failure-prone components and to predict the overall quality of a system early in the software development lifecycle (for example [1, 3, 10, 11, 15, 16, 24, 27, 28, 31, 36, 38, 39, 42]). In the fault tolerance discipline, a fault is an incorrect step, process, or data definition in a computer program [17] whereas a failure is the inability of a software system or component to perform its required function [17]. A fault is latent within a software product until it is revealed, if ever, as a failure via execution by a tester or customer.

The longer a fault remains in a pre-release product, the more time and resources will be required to find and remove the fault [2]. Limited resources preclude software engineers from identifying and fortifying all security risks. Given early, objective, and quantifiable indicators and predictors of software vulnerability, software engineers can strategically drive security efforts into the software development lifecycle (SDLC). Starting security early will afford developers enough time and resources to effectively improve security problems.

There are many methods (automated and manual) of vulnerability identification such as automated static analyses (ASA) of byte/source code, architectural risk analyses, risk-based security testing, functional security testing, penetration testing, and code audits. If ASA can represent or predict the results of the other vulnerability reporting methods, then an objective, repeatable, and automated means of accurate vulnerability identification is possible early in the SDLC. *Our research objective is to create and validate a model that uses ASA alerts to identify vulnerability-prone and attack-prone components in a software product. This model can be used to inform risk management, prioritize re-design and direct verification and validation (V&V) efforts in the later phases of the SDLC.*

A vulnerability is "an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate the [implicit or explicit] security policy" [25]. We seek to parallel the identification and prediction of fault- and failure-prone components in the quality context to vulnerability- and attack-prone components in the security context. We define vulnerability-prone as a component that has a high probability that internal V&V efforts will find security vulnerabilities prior to release. A vulnerability-prone component relates to a fault-prone component in that the focus is on the existence of problems before release. An attack occurs when an attacker has a motive or reason to attack and takes advantage of a vulnerability to threaten an asset [14]. Therefore, we define a component as being attack-prone if it has a high probability that security vulnerabilities will be attacked after release. An attack-prone component maps to a failure-prone component because the focus is on failures that occur in the field.

The rest of the paper is organized as follows: Section 2 provides an overview of prior research on fault- and failure-prone components. Section 3 presents our basis for using ASA, and Section 4 details our research methodology. Finally, Section 5 summarizes this paper.

2. Background and related work

This section is organized as follows. Section 2.1 explains our definition of vulnerability, Section 2.2 provides information on ASA tools and Section 2.3 presents prior work on analyzing and predicting reliability.

2.1. Defining vulnerability

As mentioned in Section 1, a software vulnerability is an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate the [implicit or explicit] security policy" [25]. The following are IEEE definitions [17] that serve as a basis for our definition and use of vulnerability.

Component - one of the parts that make up a system. [It] may be a hardware or software and may be subdivided into other components.

Architectural design - (1) The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system. (2) The result of the process in (1).

Error - The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Mistake - A human action that produces an incorrect result. *Note:* The fault tolerance discipline distinguishes between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

Krsul [25] originally stated that a vulnerability is an instance of an error. Since we are relating to fault- and failure-prone components, we replace error with fault to simplify the nomenclature parallelism to reliability even though the second definition of error is essentially identical to the second definition of fault. We use the first definition of error in accordance with the fault tolerance discipline. A mistake is the human action that leads to the actual vulnerability in the software although the vulnerability may never be exploited.

Error may need refined in the security context because the outcome of an attack may not be immediately known. Also, it is not clear how to measure the difference between the value of a valid computation and the exploit or result of an attack. For example, the difference between a valid input string for a software system and the exploit that causes a buffer overflow and spawns a root shell may not be trivially differenced. A risk analysis that indicates business impact from the attack may be more useful than the error.

In practice, vulnerabilities have been distinguished into three categories: design flaws, implementation bugs, and operational vulnerabilities [9, 26, 41]. The definitions of each are as follows.

Implementation bug - a vulnerability at the code level, such as buffer overflows, not checking return codes, and unsecured SQL statements [26].

Design flaw - a vulnerability that is related to the design of the system and can occur even if the program is well-coded. Examples of design flaws include a lack of or incorrect auditing/logging, ordering and timing faults, and improper authentication [26, 41].

Operational vulnerability - a vulnerability in the configuration, environment, or general use of the software [9]. For example, an unsafe `open()` on a file where read/write locks on the same file are possible by other users in the system.

These three vulnerability categories and descriptions are not standardized and the set of vulnerabilities that belong to each is subject to much contention. Security experts see vulnerabilities at different levels of abstraction. For example, a buffer overflow could be considered an implementation bug because a developer failed to set a limit on a call to the

`strcpy()` C library function in their source code. The same vulnerability could be considered a design flaw because variables, program state, and function arguments are adjacent on the stack in the architecture of the C programming language and thus apt to be overwritten.

We employ both definitions of fault from the IEEE dictionary. The first definition states that a fault is a defect in a hardware device or component. The inclusion of "component" enables us to apply vulnerability to architectural designs and operational vulnerabilities. The second definition gives a more precise meaning than the first definition since "defect" is not defined by IEEE.

According to McGraw [26], up to 60% of security vulnerabilities are design flaws. The quantity and severity of ASA alerts would, therefore, need to detect the three classifications of vulnerabilities to be an effective predictor for the overall product. ASA tools analyze code and we therefore suspect that the analyses will reveal more implementation bugs than design flaws and operational vulnerabilities. The predictive part of this research will attempt to indicate the existence of the more complex and abstract design flaws and operational vulnerabilities based on the presence of ASA alerts.

2.2 Automated Static Analyses

An ASA tool analyzes the content of a software system without executing the code [17]. ASA tools can be run on developers' desktops, integration environments, at build-time, and at major milestones of the software process [4] to make informed decisions early in the development phase.

Increasingly, ASA tools are being used to identify security vulnerabilities [5, 6]. ASA tools are faster than a manual security analysis and encapsulate security knowledge that is known by the expert tool developer but may not be known by the tool operator/software developer [6], which has contributed to the rise in adoption [4]. ASA tools can detect calls to potentially insecure library functions, bounds-checking errors and scalar type confusion, type confusion among references or pointers, memory allocation errors, and pointer-aliasing. ASA tools can also perform control- and data-flow analyses and scan for conditions based on customized rules.

2.3 Metrics and predictors for reliability

There are many metrics and models that have been used for predicting the reliability of software after release. A component is fault-prone if there is a high risk that faults will be discovered during [verification] [18]. Fault-proneness can be estimated based on measurable software attributes if relations are found between these attributes and fault-proneness [8]. Failure-prone components are components that are likely to fail after release. The metrics derived from fault-proneness often yield models that correlate the metrics to failure-proneness [7]. Research in this field has been directed in two main directions: 1) the definition of metrics to capture software complexity and testing thoroughness; and 2) the identification and experimentation of models that relate internal software metrics to fault-proneness [8].

There have been several recent research efforts on internal static software metrics (i.e. metrics obtained without executing source code) and external software metrics such as quality. Nagappan et al. [30] performed an analysis on Windows Server 2003 where they compared the defect density of components identified by ASA to defect density of components determined by testing teams, integration teams, build results, external teams, third parties, etc. found before the release of the software. They used a Spearman rank correlation to show a statistically-significant positive correlation between the two defect densities. Based on these results they used data splitting techniques to build multiple regression models for estimating the actual defect density with ASA tools. Their models yielded a similar relationship between estimated defect density and actual defect density. Thus, the alerts generated by the ASA on Microsoft-centric source code are a viable means for predicting actual defect density.

There have been many analyses that show object-oriented (OO) design metrics are good predictors of quality [10, 11, 39]. One such suite of metrics is the Chidamber and Kemerer (CK) metrics suite which is composed of six OO metrics: Weighted Methods Per Class (WMC), Depth of Inheritance Tree of a Class (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), Lack of Cohesion in Methods (LOCM). Zhou et al. [43] have examined the effectiveness of a subset of the CK metrics suite to determine if the metrics can accurately predict faults when fault severity is considered. The CBO, WMC, RFC, and LCOM metrics are statistically significant for predicting fault severity. However DIT is not and NOC is only significant with low severity faults. When the metrics are applied to predicting fault-proneness, the metrics have limited usefulness with high severity faults.

Discriminant analyses have been used in many instances to distinguish the quantity of faults in fault-prone and non-fault prone modules [12, 13, 20-23].

Discriminant analysis is a statistical method that categorizes components into groups based on metric values. Recently, modules have been classified as being fault-prone and non fault-prone using discriminant analyses against alerts generated by ASA [30, 42]. Nagappan et al. [30] demonstrated that they could distinguish 82.91% of their components. Due to the proprietary nature of their work, Type I and Type II misclassifications are not reported in their work. The classification results indicate that a strong enough difference can be made between faulty and non faulty components to confidently prioritize the allocation of testing resources and inspections. Zheng et al. [42] correctly classified 87.5% of the modules in their study when number of ASA faults and number of test failures are considered. They increased their classification to 91.7% when the models used the number of ASA faults and normalized test failures density or the number of test failures and normalized test failures density. These percentages indicate that ASA can be used to classify fault-prone and non-fault prone modules. Our research will further these findings to determine if ASA and other security metrics can classify vulnerability-prone and attack-prone modules based on discriminant analyses.

3. Quantity and severity of automated static analysis alerts as a metric

Few validated code-level security metrics can be found in the software engineering literature [40]. Currently, a prime candidate internal metric is the quantity and severity of security-related ASA alerts. ASA tools have become astute at identifying implementation bugs, but cannot be relied upon to detect design flaws [6, 30]. However, studies have indicated that ASA alerts can be used to predict the presence of classes of problems beyond the specific implementation bugs for which there is a bug pattern coded into the tool [30, 42]. These studies indicated that ASA tools can identify which components in a software system have the most problems and that ASA tools can be used as an early estimator for overall defect density.

The ultimate goal of our predictive model is to be able to discriminate between a vulnerability- or attack-prone component and a component without security concerns using the most information possible for each component. Risk can then be assessed on the vulnerability- or attack-prone component to prioritize which components are re-designed, inspected by a security expert, or undergo extensive penetration (or other security) testing. Differences in the ease of exploiting a vulnerability and the value of the targeted asset may reveal that some types of vulnerabilities (1) are more likely to escape discovery during V&V efforts; and/or (2) are more likely to be attacked than others.

4. Research methodology

We outline four steps in our research that will ultimately yield a model to distinguish vulnerability-prone and attack-prone components. The primary metrics used for this analysis are the quantity and severity of alerts generated from ASA. After the four steps are completed, then the procedure will repeat to stabilize and validate the model.

4.1 Step One: Industrial data collection

Our research will be fueled by post hoc examination of industrial data. For each code base, an ASA will be performed along with meticulously examining defect records from inspections, testing, and customers. From the myriad of defects reported, those related to security concerns need to be filtered from other defects (e.g. reliability and performance) for the inclusion in our model. Prior to beginning this step, a measurement framework will need to be developed to make this classification objective and repeatable.

4.2 Step Two: Statistical relationships

The data from Step One will be fed into statistical models to examine the correlation between the static analysis alerts and the actual quantity of vulnerabilities and/or attacks identified later in the development process or by customers. The correlation may indicate that alerts are good predictors of vulnerability.

4.3 Step Three: Discriminant analysis and ranking

Discriminant analysis[2] is a statistical technique used to classify components into groups based on metric values. The purpose is to determine the class of an observation based on a static analysis alerts. A model is built based on a set of observations for which the classes are known. This set of observations is sometimes referred to as the training set. Based on the training set, the technique constructs a set of linear functions of the predictors, known as discriminant functions, such that

[2] http://www.resample.com/xlminer/help/DA/da_intro.htm

$L = b_1x_1 + b_2x_2 + ... + b_nx_n + c$, where the b's are discriminant coefficients, the x's are the input variables or predictors and c is a constant.

Discriminant analysis has been used as a tool for the detection of fault-prone programs [19, 22, 28]. In our case, the discriminant function is the prediction model developed in Step Two. The groups are (1) vulnerability-prone and not vulnerability-prone; and (2) attack-prone and not attack-prone. Once the components are classified, we can also rank the risk of the components based upon the value of the model from Step Two.

4.4 Step Four: Examination of Results

A comparison of the model classification from Step Four and the actual vulnerability and/or attack metrics will reveal the following results:

- True positives: the component is classified as vulnerability- and/or error-prone, and the actual results are consistent with the prediction.
- False positives: the component is classified as vulnerability- and/or error-prone, but the actual results indicate no appreciable security problems.
- False negatives: the component is classified as not vulnerability- and/or error-prone, but the actual results indicate appreciable security problems.

The long term goal is to develop a stable model for classification that meets or surpasses an accuracy goal, such as "95% of all components are classified properly." As the model is stabilizing, a thorough examination of the design and source code of the components classified incorrectly (the false positives and false negatives) will yield insight into the metrics that should or should not be included in the model. The results of this examination should be fed into the next iteration of Step One of this iterative research process until our model stability goal is reached.

5. Summary

Extensive research has shown that software metrics can be used to identify fault- and failure-prone components and to predict the overall quality of a system early in the software development lifecycle. Our goal is to parallel this work in the security realm. We will attempt to develop a model for the identification of vulnerability-prone and attack-prone components. The metric used to build the model will be based on alerts (e.g. alert density) generated by ASA of source code. The verification of ASA alerts as early and good indicators of vulnerability and attack will inform risk management and prioritize re-design and V&V efforts in the later phases of the SDLC.

6. References

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object Orient Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* vol. 21, pp. 751-761, 1996.

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

[3] L. C. Briand, J. Wust, and H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study," ISERN-98-29, 1998.

[4] P. Chandra, B. Chess, and J. Steven, "Putting the Tools to Work: How to Succeed with Source Code Analysis," in *IEEE Security & Privacy*. vol. 4, 2006, pp. 80-83.

[5] B. Chess, "Improving Computer Security using Extended Static Checking," in *IEEE Symposium on Security and Privacy*, Berkeley, CA, 2002, pp. 160-173.

[6] B. Chess and G. McGraw, "Static Analysis for Security," in *IEEE Security and Privacy*. vol. 2, 2004, pp. 76-79.

[7] G. Denaro, S. Morasca, and G. Popek, "Deriving models of software fault-proneness," in *International Conference on Software Engineering and Knowledge Engineering*, pp. 361-368.

[8] G. Denaro, A. Polini, and W. Emmerich, "Early Performance Testing of Distributed Software Applications," in *Workshop on Software and Performance*, Redwood Shores, California, 2004, pp. 94 - 103

[9] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston, MA: Addison-Wesley, 2007.

[10] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.,* vol. 27, pp. 630 - 650 July 2001.

[11] T. Gyimóthy, R. Ference, and L. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.,* vol. 31, pp. 897-910, Oct. 2005.

[12] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. Hudepohl, "Evolutionary Neural Networks: A Robust Approach to Software Reliability Problems," in *Eighth Int'l Symp. Software Reliability Eng.*, 1997, pp. 13-26.

[13] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. Hudepohl, "Using the Genetic Algorithm to Build Optimal Neural Networks for Fault-Prone Model Detection," in *Seventh Int'l Symp. Software Reliability Eng.*, 1996, pp. 152-162.

[14] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.

[15] J. Hudepohl, S. J. Aud, T. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop," *IEEE Software,* vol. 13, pp. 56-59, September 1996.

[16] J. Hudepohl, W. Jones, and B. Lague, "EMERALD: A Case Study in Enhancing Software Reliability," in *Eighth*

International Symposium on Software Reliability Engineering (Case Studies), Albuquerque, New Mexico, 1997, pp. 85-91.
[17] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.
[18] T. M. Khoshgoftaar, E. B. Allen, and J. Deng, "Using Regression Trees to Classify Fault-Prone Software Modules," *IEEE Transactions on Reliability,* vol. 51, pp. 455-562, December, 2002 2002.
[19] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of Software Modules with High Debug Code Churn in a Very Large Telecommunications System," in *International Sympoisum on Software Reliability Engineering*, White Plains, NY, 1996, pp. 364-371.
[20] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Applications of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System," *Trans. Neural Networks,* vol. 8, pp. 902-909, 1997.
[21] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and W. Jones, "Classification Tree Models of Software Quality over Multiple Releases," in *10th Int'l Symp. Software Reliability Engineering*, 1999, pp. 116-125.
[22] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, N. Goel, J. P. Hudepohl, and J. Mayrand, "Detection of fault-prone program modules in a very large telecommunications system," in *IEEE International Symposium on Software Reliability Engineering*, Toulouse, France, 1995, pp. 24-33.
[23] T. M. Khoshgoftaar, E. B. Allen, A. Naik, W. Jones, and J. P. Hudepohl, "Using Classification Trees for Software Quality Models: Lessons Learned," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 9, pp. 2127-231, 1999.
[24] T. M. Khoshgoftaar and J. C. Munson, "Predicting Software Development Errors using Software Complexity Metrics," *EEE Journal on Selected Areas in Communications,* vol. 8, pp. 253-261, 1990.
[25] I. Krsul, "Software Vulnerability Analysis," in *Computer Science*. vol. PhD West Lafayette: Purdue University, 1998.
[26] G. McGraw, *Software Security: Building Security In*. Boston: Addison-Wesley, 2006.
[27] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.,* vol. 33, pp. 2-13, 2007.
[28] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering,* vol. 18, pp. 423-433, 1992.
[29] N. Nagappan, "A Software Testing and Reliability Early Warning (STREW) Metric Suite," in *Computer Science* Raleigh, NC: North Carolina State University, 2005.
[30] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-release Defect Density," in *International Conference on Software Engineering*, St. Louis, MO, 2005, pp. 580-586.
[31] N. Nagappan, T. Ball, and B. Murphy, "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures," in *International Symposium on Software Reliability Engineering*, Raleigh, NC, 2006, pp. 62-74.
[32] N. Nagappan and L. Williams, "A Software Reliability Estimation Framework for Extreme Programming," in *International Symposium on Software Reliability Engineering (ISSRE) Student Paper*, Denver, CO, 2003.
[33] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson, " Providing Test Quality Feedback Using Static Source Code and Automatic Test Suite Metrics," in *Chicago, IL*, International Symposium on Software Reliability Engineering (ISSRE) 2005, 2005, pp. 85-94.
[34] N. Nagappan, L. Williams, and M. Vouk, "Initial Results of Using In-Process Testing Metrics to Estimate Software Reliability," North Carolina State Department of Computer Science CSC TR-2004-5, January 25, 2004.
[35] N. Nagappan, L. Williams, M. Vouk, J. Hudepohl, and W. Snipes, "A Preliminary Investigation of Automated Software Inspection," in *IEEE International Symposium on Software Reliability Engineering*, St. Malo, France, 2004, pp. 429-439.
[36] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Using In-Process Testing Metrics to Estimate Software Reliability: A Feasibility Study," in *Fast Abstract at the International Symposium on Software Reliability Engineering (ISSRE) 2004*, St. Malo, France, 2004, pp. 21-22.
[37] N. Nagappan, L. Williams, and M. A. Vouk, "Towards a Metric Suite for Early Software Reliability Assessment," in *International Symposium on Software Reliability Engineering Fast Abstract*, Denver, CO, 2003.
[38] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *International Symposium on Software Testing and Analysis*, Boston, Massachusetts, 2004, pp. 86-96.
[39] R. Subramanyam and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering,* vol. 29, pp. 297-310, April 2003.
[40] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller, "Defining an Adaptive Software Security Metric from a Dynamic Software Failure Tolerance Measure," in *COMPASS '96*, Gaithersburg, MD, 1996, pp. 250-263.
[41] C. Wysopal, L. Nelson, D. Dai Zovi, and E. Dustin, *The Art of Software Security Testing: Identifying Software Security Flaws*. Boston: Addison Wesley, 2006.
[42] J. Zheng, L. Williams, W. Snipes, N. Nagappan, J. Hudepohl, and M. Vouk, "On the Value of Static Analysis Tools for Fault Detection," *IEEE Transactions on Software Engineering,* vol. 32, pp. 240-253, 2006.
[43] Y. Zhou and L. Hareton, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Transactions on Software Engineering,* vol. 32, pp. 771-789, October 2006.