Prioritizing Software Security Fortification through Code-Level Metrics

Michael Gegick, Laurie Williams, Jason Osborne, Mladen Vouk
North Carolina State University
Raleigh, NC 27695
{mcgegick, lawilli3, vouk}@ncsu.edu, jaosborn@stat.ncsu.edu

ABSTRACT
Limited resources preclude software engineers from finding and fixing all vulnerabilities in a software system. We create predictive models to identify which components are likely to have the most security risk. Software engineers can use these models to make measurement-based risk management decisions and to prioritize software security fortification efforts, such as redesign and additional inspection and testing. We mined and analyzed data from a large commercial telecommunications software system containing over one million lines of code that had been deployed to the field for two years. Using recursive partitioning, we built attack-prone prediction models with the following code-level metrics: static analysis tool alert density, code churn, and count of source lines of code. One model identified 100% of the attack-prone components (40% of the total number of components) with an 8% false positive rate. As such, the model could be used to prioritize fortification efforts in the system.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *statistical methods*. D.2.8 [**Software Engineering**]: Metrics – *Product metrics*.

General Terms
Security, Measurement.

Keywords
Vulnerability-prone, attack-prone.

1. INTRODUCTION
Limited resources preclude software engineers from finding and fixing all vulnerabilities in their software system. The inspection of all problem areas may cause a development team to expend valuable and limited fortification resources on low risk areas of the code. *Our research objective is to use internal, code-level metrics to create and evaluate security prediction models that can be used to guide the prioritization of fortification efforts.*

QoP'08, October 27, 2008, Alexandria, Virginia, USA.

The candidate internal metrics to our predictive models are counts and density of alerts from the static analysis tool FlexeLint[1], code churn, and count of SLOC. We explicitly investigate internal metrics that can be obtained during development and may have predictive power where system- and feature-level testing identified vulnerabilities. The failures were identified during late-cycle system testing and post delivery. We evaluated security-based predictive models based upon each of the internal metrics individually and then all together in the same model.

We conducted a case study on a large commercial[2] telecommunications software system comprised of over 1.2 million source lines of code (SLOC). We used the internal metrics to create and evaluate predictive models that identify the components that were most likely to be attacked.

The remainder of the paper is organized as follows: Section 2 provides background information, Section 3 provides an overview of prior research on predicting problem areas, Section 4 details our research methodology, Section 5 presents the limitations, Section 6 presents results from predictive modeling with internal metrics. Finally we provide a discussion in Section 7 and summarize in Section 8.

2. BACKGROUND
This section presents background material on the definitions and techniques related to our work.

2.1 Definitions
Internal metrics - **"**Those metrics that measure internal attributes of the software related to design and code. These "early" measures are used as indicators to predict what can be expected once the system is in test and operation" [10].

Fault - "An incorrect step, process, or data definition in a computer program. Note: A fault, if encountered, may cause a failure" [11].

Fault-prone component - "A component that will likely contain faults" [5].

Failure – "The inability of a software system or component to perform its required functions within a specified performance requirements [11]."

[1] http://www.gimpel.com
[2] The corporation is anonymous due to the sensitivity of the data.

Failure-prone component – A component that will likely fail due to the execution of faults [22].

Vulnerability - 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 [13].

Vulnerability-prone component - A component that is likely to contain one or more vulnerabilities that may or may not be exploitable [7].

Attack - The inability of a system or component to perform functions without violating an implicit or explicit security policy. We borrow from the ISO/IEC 24765 [11] definition of failure to define attack, but remove the word "required" because attacks can result from functionality that was not stated in the specification.

Attack-prone component - A component that will likely be exploited [7].

Extensive research (including [2, 15, 20]) has shown that software metrics can be used to identify fault- and failure-prone components[3] and to predict the overall reliability of a system. These models indicate where in the software reliability problems most likely exist so that software engineers have an objective strategy that focuses their verification and validation efforts.

Fault-prone prediction models can estimate which components are fault-prone, but if the faulty code is never executed it will not be prone to failure. The inspection of *all* fault-prone components may cause the development team to expend valuable and limited verification resources on low risk areas of the code that may be of high quality and/or may rarely or never be used by a customer. Failure-prone prediction models, based on a customer's operational profile, and historical failures from the field, can further guide fault-finding efforts toward low quality components that are likely to cause problems for the end user.

A vulnerability-prone component is analogous to a fault-prone component in that vulnerabilities may remain latent (similar to faults) until encountered by an attacker (or tester/customer) in an executing system. The vulnerabilities in a vulnerability-prone component can include a wide range of severity and likelihood of exploitation.

A similar relationship between vulnerability-prone and attack-prone components exists as with fault- and failure-prone components. An attack-prone component is a vulnerability-prone component if an attacker will likely exploit one or more vulnerabilities in that component. Vulnerability-finding techniques can cause security experts to expend valuable and limited security resources on low risk areas of the code that may be adequately fortified, may be uninteresting to an attacker, or contain difficult-to-exploit vulnerabilities. Attack-prone prediction models based on test failure data and attacks in the field can make vulnerability-finding efforts more efficient and effective by identifying those components with the highest security risk. While failures in the reliability realm are dependent on the operational profile, attacks can occur anywhere in a software system regardless of the operational profile.

[3] A component is constituent part, element, or piece of a complex whole [11].

2.2 Automated Static Analysis (ASA)

We used static analysis tool output as one of our internal metrics. A static analysis tool analyzes the content of a software system to detect faults without executing the code [3]. We use the term automated static analysis (ASA) to refer to the use of static analysis tools. The output of an ASA tool is an alert. An alert is a notification to a software engineer, of a potential fault in the source code that has been identified via static analysis [9]. Static analysis tools provide an early, automated, objective, and repeatable analysis for detecting faults.

2.3 Recursive Partitioning

Our predictive models are comprised of a statistical technique and the following input variables: ASA alerts, code churn, and SLOC. Recursive partitioning, also known as classification and regression trees (CART), is a statistical technique that recursively partitions data according to X and Y values. The result of the partitioning is a tree of groups where the X values of each group best predicts a Y value. The threshold or split between leaves is chosen by maximizing the difference in the responses between the two leaves. Recursive partitioning produces leaves of a decision tree that contain observations that are distinctly different than observations in other leaves.

3. RELATED WORK

We now present prior research that has shown how internal metrics can predict fault-, failure-, and vulnerability- prone components.

3.1 ASA Alerts as Static Metrics for Fault- and Failure-prone Prediction

Discriminant analyses have been used in many instances to distinguish fault-prone and not fault-prone components [12, 16, 23]. Nagappan et al. [16] demonstrated that they could distinguish 82.91% of their components. Zheng et al. [23] correctly classified 87.5% of the modules in their study when the number of ASA faults and number of test failures are considered. Our research will further these findings to determine if ASA can be used to classify attack-prone components based on recursive partitioning and logistic regression.

3.2 Prediction with Code Churn

In our setting, churn is the count of SLOC that has been added or changed in a component since the previous revision of the software. Nagappan et al. [17] experimented with churn to determine if there was a positive association between churn and defect density. In an analysis with Windows Server 2003, they discovered that there is a correlation and that churn can discriminate between fault-prone and non fault-prone binaries with an accuracy of 89%.

3.3 Vulnerability-prone Component Predictions

Neuhaus et al. [18] have also investigated predictive models that identify vulnerability-prone components. They created a software tool, Vulture, that mines a bug database for data that predict which components will likely be vulnerable. The predictors for their models are libraries and APIs that are associated with vulnerabilities. They performed an analysis with Vulture on Bugzilla, the bug database for the Mozilla browser, using imports and function calls as predictors. They were able to identify 50% of all of the vulnerable components in Mozilla.

4. RESEARCH METHODOLOGY

We examined the following hypothesis in this study:

H_0: The internal metrics, count and density of ASA alerts, code churn, and count of SLOC cannot discriminate between attack-prone and non attack-prone components.

We do not accept the null hypothesis if we can show, with statistically significant results, that metrics can be used to discriminate between attack-prone and non attack-prone components. To examine the hypothesis, we mined and analyzed data from a large commercial telecommunications software system that had been deployed to the field for two years. The system contained 38 well-defined components whereby each component consisted of multiple source files. A full set of information necessary for our analysis was only available for 25 (66%) of the components of the system, and thus the study focuses on those components. Data at the file-level were unavailable. The 25 components we analyzed summed to approximately 1.2 million SLOC. All faults in the failure reports have since been fixed.

The failure reports included pre- and post-release failures. A pre-release failure for our study is a failure discovered by an internal tester during feature and/or system robustness testing. A post-release failure indicates that a failure occurred in the field and was reported by a customer. Both the pre- and post-release reports explicitly identified the component where the solution was applied.

An ASA analysis was performed on the system by FlexeLint. Although FlexeLint is a reliability-based ASA tool, we sought to determine if the fault types identified by the tool could be warnings of security vulnerabilities on a per-component basis. The alerts produced by the tool were audited by a 3rd party contracted by the vendor. The audited alerts in this study contain only alerts that were determined to be actual problems in the software. The unaudited list of alerts in this case study includes both the alerts that are actual software problems and those that do not.

4.1 Failure Report Classifications

The first author and additional research student, doctoral students in software security, independently reviewed each of the 1255 pre- and post-release failure reports to classify which failure reports were security problems and which were non security-related reliability problems. Based on the failure reports, we developed criteria that identified which failures were non-security problems and which were indicative of security problems. Some failure reports were explicitly labeled as security problems (approximately 0.5%) by either internal testers or security engineers. We analyzed all other failure reports that were not explicitly labeled a security problem to determine if they were related to a vulnerability. We found that many reports contained the following keywords that are often seen in security literature: crash, denial-of-service, access level, sizing issues, resource consumption, and data loss. These keywords increased our suspicion of whether or not the failure could be a security problem, but did not necessarily indicate a security problem. The criteria for a failure report to be classified as a security problem are now listed:

- **Remote attacks.** The failure reports explicitly indicated when the failure was due to a remote user or machine. Pre- and post-release failure reports that contained any of the security keywords (above) and could be remotely initiated had the highest confidence of an exploitable vulnerability.
- **Insider attacks.** If the failure report did not indicate that the failure was due to an external user or machine, then we looked for attacks that did not require remote access to the system.
- **Audit capability.** Weak or absent logging for important components was considered a security vulnerability to prevent non-repudiation. An example of an attack on logging problems can be found in [21].
- **Security documentation.** We also considered if the fundamental principles of software security were followed. For instance, in two failure reports, the testers indicated that the problem would occur if the users were not "well-behaved," which breaks the principle of *Reluctance to Trust* [1].

The vendor's security engineer audited our report and eliminated false positives (6.8%) from our report. False positives were reliability faults that we claimed to be security vulnerabilities. The number of vulnerabilities in our analysis 52 (4.1%) of the total failure reports were classified as security failures. Any of the failure reports that we misclassified as non-security problems are false negatives in the study. We used the failure reports that were verified as security problems in our statistical analyses. A security-based failure (i.e. an attack) represents the presence of a security vulnerability.

For this paper, failure and alert densities are calculated by dividing the number of failures and alerts by the number of KLOC (thousands lines of source code) of that component.

4.2 Classification of Components

We manually classified a component as attack-prone if it had at least one pre-release or post-release security failure. A component with no reported security failures was classified as a non attack-prone component. We use the threshold of one failure because there is little variability in the failure count per component and only one attack is needed to cause substantial business loss. The failures reported by customers were not necessarily due to attacks. However, according to the failure report and the security engineer, the vulnerability could have been exploited maliciously and thus we consider the failure an "attack" for the purposes of this paper. No malicious attacks

were reported for the software system. In our setting, ten of the 25 components were classified as attack-prone.

4.3 Predictive Modeling

The models we build identify the internal metrics that best predict which components are attack-prone and non attack-prone. Our four predictive models used recursive partitioning with the following three internal predictors: ASA alerts, churn, and SLOC. We approached model selection in a stepwise fashion where we analyze how each metric performs individually and then combine the metrics into one model. If the predictions generated by our models are consistent with the post hoc manual classifications based upon testing results and customer reports, then the models may be a viable approach to prioritizing security-based verification efforts for our software system.

In our setting, Type I and Type II errors are as follows:

Type I error (false positive) - a non attack-prone component that was predicted to be attack-prone.

Type II error (false negative) - an attack-prone component that was predicted to be non attack-prone.

To validate the efficacy of each model, we evaluate the models with two techniques: k-fold cross-validation and receiver operating characteristic (ROC) curves. We cross-validated the R^2 to determine how much variation was accounted for by the model rather than by random error. In our setting we used five-fold cross-validation where there were five groups of components, each consisting of an equal number of randomly chosen components. Five has been shown to be a good value for performing cross-validation [8]. One group was used as the test set and the training set contained the other four groups. The R^2 of the training set was compared to the test set during five trials where each group was allowed to be in the test set once.

The second test was with the ROC curve. With ROC, a curve is drawn on a graph where the true positive rate of attack-prone component identification is on the y-axis and the false positive rate on the x-axis. The true positive rate is the probability that the attack-prone prediction is correct. The false positive rate is the probability that the attack-prone prediction is incorrect when a component is not attack-prone. The area under the curve measures how well the predictors estimate the probability a component is attack-prone.

5. LIMITATIONS

Our security data are sparse. Our analysis included only 3.7% of the organization's faults making statistical analyses difficult and reducing the confidence in our models. Additionally, we had a small sample size of 25 components to partition, making analyses difficult; an analysis at the file-level was not possible. The models that fit our system may not fit all software systems due to differences such as architecture, programming language, and developers. Also, after testing is complete, we can only know detected faults; we do not know which faults still remain [6]. Thus, our analysis is based on incomplete vulnerability discovery. Additionally, system and feature-level testing may not be adequate for detecting all vulnerabilities while using other techniques (e.g. architectural risk analyses) may be suitable for finding different types of security vulnerabilities. Furthermore, our ASA analysis is based upon only one ASA tool, FlexeLint, and may not be representative of the predictive power of other ASA tools.

There was some subjective interpretation in the analysis of pre- and post-release failure reports though the cross examination between two doctoral students and one industrial security expert strengthens our results. Also, we could not determine if testing effort was equal for all components; it may have been driven by factors such as churn or code size. Lastly, once the model is applied, the model is not guaranteed to be effective for the next revision of the software.

6. PREDICTIVE MODEL ANALYSIS

In this section, we present the analysis of four predictive models that use internal metrics to predict attack-proneness. We formulated models based upon each of the internal metrics individually and then with all of the internal metrics together.

6.1 Predictive Model 1: ASA Alerts

We used recursive partitioning to define a threshold based on the number (or density) of alerts that will distinguish attack-prone and non attack-prone components. The threshold divides components into two groups of components (partitions) that we will designate as the lower partition (the smaller count or density of alerts) and the upper partition (the higher count or density of alerts). Model 1 uses recursive partitioning with the following ASA alerts as predictors in our models: count and density of audited and un-audited null pointer, memory leak, buffer overrun alerts, the combination of the previous three security-related alerts, and all of the reported FlexeLint alerts. We mined and organized all of these predictors to be used as independent variables in the recursive partitioning analysis. In our setting, the upper partition resulting from the split will be interpreted as having true positives when containing attack-prone components, and if attack-prone components exist in the lower partition, then they are Type II errors.

The first split (see Figure 1) with recursive partitioning was based on total alert density. Of all the ASA alert metrics, the total alert density served as the best metric for separation based on the functions of the metric that maximize the difference between attack-prone components and non attack-prone components. The upper partition correctly identified 40% (100% of the attack-prone components) of the components as attack-prone, but had a 28% Type I error rate where components were misclassified as attack-prone. The lower partition contained 32% true negatives. The value of total alert density at the split is 0.19 alerts/KLOC. That is, a component with a total alert density below 0.19 alerts/KLOC is in the lower partition and a component with an alert density greater than or equal to 0.19 alerts/KLOC is in the upper partition. The p-value of the split is 0.012 and the R^2, is 31.5%.

The value of R^2 is low which indicates that the proportion of the response that can be attributed to the alerts is small compared to that of standard error. Therefore, we made a second split to increase the ability of the model to account for more variability. The second split again used total alert density of all of the possible metrics and produced the split in the upper partition of the first split. The value of R^2 is low which indicates that the

proportion of the response that can be attributed to the alerts is small compared to that of standard error. Therefore, we made a second split to increase the ability of the model to account for more variability.

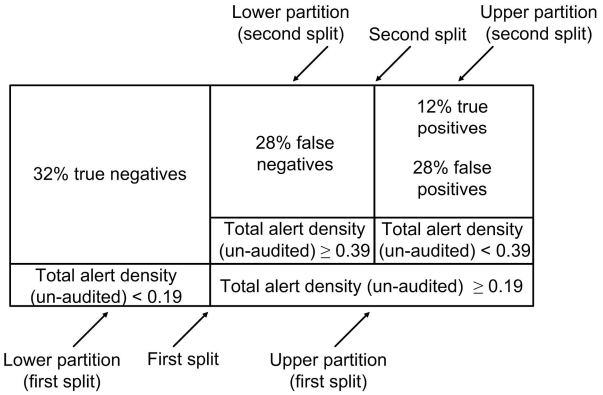

Figure 1. Model 1 after two splits with recursive partitioning to separate attack-prone from non attack-prone components.

The second split again used total alert density of all of the possible metrics and produced the split in the upper partition of the first split. The value for the total alert density at the split is 0.39 alerts/KLOC. The p-value for the second split is .007 and the R^2 is 32.2%. The combined R^2 of the model is therefore 63.7%. Twelve percent of the attack-prone components are in the upper partition of the second split and 28% in the lower partition. The 28% misclassification (shown in lower partition of the second split in Figure 1) was originally correctly classified as attack-prone by the first split, but the second split misclassified them as non attack-prone. The second split indicates that the higher alert density, then the less chance that a component is attack-prone, contradicting the first split. No further splitting with a p-value at or less than .05 was possible. We do not include the second split in our model because it is not intuitive for software engineers to use a model that indicates more alerts means that a component is less likely to be attack-prone.

The average R^2 in the cross-validation was taken from the five trials and was calculated as 60.7% suggesting that the R^2 is correct within the overall model. The area under the ROC curve for Model 1 is 93.0%. The large area under the curve indicates that the predictors for each model are accurate at specifying attack-prone components.

A summary of the results from Model 1 and the following models are presented in Table 1. The Type I and II error rates, R^2, validated R^2, and ROC curve values are reported in Table 1 for the remainder of the paper unless explicitly stated for emphasis.

There are two possible reasons why the second split showed that a higher alert density means there is less of a chance that a component is attack-prone. First, the metric, total alert density includes all alerts, both security-related and not security-related, which inflated the alert density value. Secondly, system testers may be finding vulnerabilities that ASA cannot find and ASA may be finding faults in components that the testers do not find.

Observation 1: *ASA alerts by themselves cannot accurately predict pre-release attack-prone components.*

6.2 Predictive Models 2 and 3: Churn and SLOC

We attempted to build predictive models with churn and SLOC by themselves, but we were similarly unsuccessful. In Model 2, we used churn as the single predictor and only one split (p=.01) was possible. The value of the churn at the split is 3,861 SLOC. No further splits at or below the .05 level could be made.

Observation 2: *Code churn by itself cannot accurately predict attack-prone components.*

We also tried modeling with churn normalized against SLOC according to Nagappan et al. [17]. We found that only one split could be made with 36% Type I errors and no Type II errors. The R^2 was only 21.9%. The R^2 values are too low to accept as potential models.

We also tried to predict attack-prone components with SLOC as the only metric in Model 3, but no splits were possible at or below the .05 level.

Observation 3: *SLOC by itself cannot accurately predict pre-release attack-prone components.*

6.3 Predictive Model 4: Alerts, Churn, and SLOC

We created a model, Model 4, with the following predictors: alerts, churn and SLOC. The first split was determined by churn and had 100% of the attack-prone components in the upper partition with a Type I error rate of 28%. The value of churn at the split is 3,861 SLOC with an R^2 of 31.5%. The p-value is .012. We performed another split as shown in Figure 2 and the metric chosen was total alert density. The total alert density further separated the attack-prone and non attack-prone components in the upper partition from the first split. The results showed that the higher the alert density, the more likely the component was attack-prone. The value of the alert density at the split was 0.19 alerts/KLOC and the p-value for the second

Table 1: Summary of recursive partitioning models results after all statistically significant splits have been made.

Model	Metric	Type I (False Positive)	Type II (False Negative)	R^2	Cross-validated R^2	ROC
1	alerts	28%	0%	31.5%	19.4%	76.7%
2	churn	28%	0%	31.5%	30.1%	76.7%
3	SLOC	--	--	--	--	--
4	**alerts, churn, SLOC**	**8%**	**0%**	**67.9%**	**61.1%**	**93.3%**

split was less than .0001. No further splitting was possible and thus SLOC has no predictive power in the model. We tested the Spearman correlation between total alert density and churn and found there was no correlation. Thus, the two predictors are independent of each other and multi-collinearity is not a concern.

Since Model 4 indicated that two different metrics, churn and alerts, could be combined we tried using discriminant analysis to distinguish attack-prone from non attack-prone components. The model did not exhibit a goodness-of-fit to the data to discriminate between attack-prone and non attack-prone components and thus we could not use discriminant analysis.

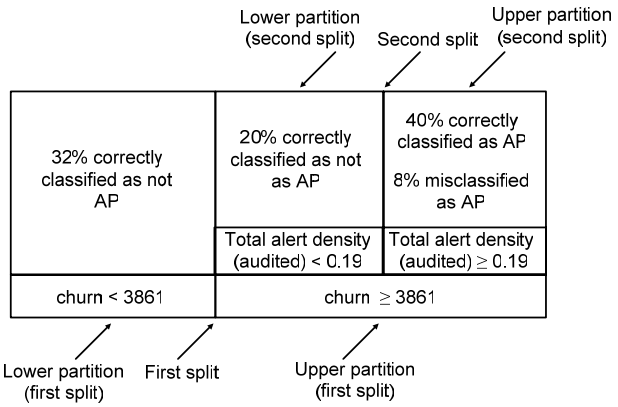

Figure 2. Model 4 after two splits with recursive partitioning to separate attack-prone from non attack-prone components.

We were successful with using total alert density and churn with logistic regression, a linear regression technique used in generalized linear models, to identify attack-prone and non attack-prone components. The tests from the whole-model fit, which compares the model to a model with just the intercept (i.e. without the two predictors) is shown in Table 2. The probability that the model fits better than with just the intercept parameter is given by the Prob>ChiSq, 0.0009, which indicates the model fits better than with just the intercept parameter.

Table 2: Logistic regression whole model test.

Model	-LogLikelihood	ChiSquare	Prob>ChiSq
Difference	8.24	16.5	0.0009
Full	8.06		
Reduced	16.30		

We found that the churn and total alert density are good predictors of attack-prone components. The third parameter in Model 5, the product of churn and total alert density, is however weakly significant (p=.07) as shown in Table 3. The logistic regression supports the recursive partitioning results where components with high churn and high total alert density are also more likely to be attack-prone components. We provide unit odd ratios as shown in Table 3 to show how the estimated probability of a component being attack-prone increases per unit increase in predictor. The results show that there is a positive increase in likelihood of a component being attack-prone if the predictors increase in count or density. We also tested the predictors under logistic regression with an ROC curve and found that 93.6% of the area is under the curve representing that most of the variability is accounted for.

Table 3: Significance of predictors with logistic regression. Note: For logistic regression, total alert density was measured in 100KLOC.

Predictor	Prob>ChiSq	Unit Odds Ratio
Churn	.0573	1.0
total alert density	.0527	1.77
(churn-17650.7)* (totalAlertDensity-41.7)	.0720	1.0

Observation 4: *ASA alerts and code churn can predict pre-release attack-prone components.*

6.4 Interpretation of Results

The results from Model 1 are inconclusive for determining if the ASA alerts can distinguish between attack-prone and non attack-prone. The total alert density in the second split represents that more alerts means less security failures, and there was no correlation between the alerts and failures. However, Model 4 represents that components with high churn and high total alert density are more likely to be attack-prone. The total output of FlexeLint in total alert density was a better predictor than our security-based alerts. Models 1, and 4 have similar R^2 values and ROC curves, but Model 1 had more Type II errors (28%) than Models 4. In our setting, the combination of different metrics in a single model have more predictive power than a model with only one metric. Lastly, for each combination of internal metrics, we also attempted to build models using discriminant analysis and logistic regression. In all cases (excluding Model 4 for logistic regression), the models lacked a goodness-of-fit.

7. DISCUSSION

While no single fault detection technique can detect all problems, static analysis tools have been shown to predict reliability problems that are not explicitly detected by ASA based on alert counts and density [16, 23]. ASA tools analyze code and are thus more apt at coding problems then the more complex, high-level design or operational problems identified by testing. In our research, Model 4 indicates that ASA alerts can, in part, identify components that contained vulnerabilities identified by late-cycle robustness testing.

The association of simple faults to complex faults is known as the coupling effect [4]. The coupling effect has also been observed in the context of mutation testing [14, 19]. Evidence of the coupling effect shows that ASA can be used while code is written to identify coding faults as well as predict that more complex design and operational vulnerabilities exist.

8. SUMMARY AND FUTURE WORK

We have created models to predict which components are prone to attacks on a large scale industrial software system. The internal metrics used in our models are churn, count of SLOC, and density of ASA alerts from the static analysis tool, FlexeLint. We have shown that churn and alerts can be used to

identify attack-prone components using recursive partitioning. We chose the model with churn and ASA alerts to have the most applicability to our industrial system because of the 8% false positive rate 0% false negative rate. Alerts and churn as metrics can indicate future problems in the software so that security experts can prioritize their security efforts.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under CAREER Grant No. 0346903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] S. Barnum and M. Gegick, "Design Principles," https://buildsecurityin.us-cert.gov/portal/article/knowledge/Principles, 2005.

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

[3] P. Chandra, B. Chess, and J. Steven, "Putting the Tools to Work: How to Succeed with Source Code Analysis," IEEE Security & Privacy, vol. 4, no. 3, pp. 80-83, May/June, 2006.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, vol. 11, no. 4, pp. 34-41, April, 1978.

[5] G. Denaro, "Estimating software fault-proneness for tuning testing activities," *International Conference on Software Engineering*, St. Malo, France, pp. 269-280, 2000.

[6] E. Dijkstra, *Structured Programming*, Brussels, Belgium, 1970.

[7] M. Gegick and L. Williams, "Toward the Use of Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components," *First International Workshop on Systems Vulnerabilities (SYVUL '07)* Santa Clara, CA, July 1-6 2007.

[8] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning*, New York, Springer, 2001.

[9] S. Heckman and L. Williams, "Automated adaptive ranking and filtering of static analysis alerts," *Fast abstract at the International Symposium on Software Reliability Engineering*, Raleigh, NC, November 2006.

[10] ISO, "ISO/IEC DIS 14598-1 Information Technology - Software Product Evaluation - Part 1: General Overview," October 28 1996.

[11] ISO/IEC 24765, "Software and Systems Engineering Vocabulary," 2006.

[12] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and W. Jones, "Classification Tree Models of Software Quality over Multiple Releases," *10th International Symposium on Software Reliability Engineering*, pp. 116-125, 1999.

[13] I. Krsul, "Software Vulnerability Analysis," PhD Thesis in Computer Science at Purdue University, West Lafayette 1998.

[14] R. J. Lipton and F. G. Sayward, "The Status of Research on Program Mutation," *In Digest for the Workshop on Software Testing and Test Documentation*, pp. 355-373, December 1978.

[15] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering,* vol. 18, no. 5, pp. 423-433, 1992.

[16] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-release Defect Density," *International Conference on Software Engineering*, St. Louis, MO, pp. 580-586, 2005.

[17] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict Defect Density," *International Conference on Software Engineering*, St. Louis, MO, pp. 284-292, 15-21 May 2005.

[18] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting Vulnerable Software Components," *Computer and Communications Security*, Alexandria, VA, pp. 529-540, 29 October-2 November 2007.

[19] A. J. Offutt, "The Coupling Effect: Fact or Fiction?," *International Symposium on Software Testing and Analysis*, Key West, Florida, pp. 131-140, 1989.

[20] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *International Symposium on Software Testing and Analysis*, Boston, Massachusetts, pp. 86-96, 2004.

[21] V. Prevelakis and D. Spinellis, "The Athens Affair," IEEE Spectrum, vol. 44, no. 7, pp. 26-33, July, 2007.

[22] A. Schroter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," *International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, pp. 18-27, September 21-22 2006.

[23] 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, no. 4, pp. 240-253, April 2006.