Predicting Vulnerable Classes in an Android Application

Riccardo Scandariato
KU Leuven
Departement of Computer Science
Celestijnenlaan 200A
3001 Heverlee, Belgium
riccardo.scandariato@cs.kuleuven.be

James Walden
Northern Kentucky University
Department of Computer Science
Nunn Drive
Highland Heights, KY 41076
waldenj@nku.edu

ABSTRACT

Smart phones have been outselling PCs for several years. The Android operating system dominates the smart phone market, and the Android Market (now Google Play Store) recently passed the mark of 15 billions application downloads. Therefore, there is a large base of users that is attractive for hackers to target. In this paper, we will examine the questions of whether mobile applications developed for the Android platform are vulnerable or not, and how to predict which classes of an Android application are vulnerable. This paper approaches an answer to these questions by analyzing one very popular application of the Android Market and developing a vulnerability prediction model with high accuracy (over 80%) and precision (over 75%).

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Empirical software engineering

Keywords

Mobile application security, static analysis, Android, metrics

1. INTRODUCTION

Consumers have purchased more smart phones than PCs since the last quarter of 2010 [1], bringing a change to the way most users access online resources. Instead of using a web browser, people increasingly use mobile applications on their smart phone to perform online tasks such as banking and shopping. Most of these mobile applications run on the Android platform. The Android operating system dominates the smartphone market with a 59% share of the market and 90 million phones sold in the first quarter of 2012 [2].

MetriSec'12, September 21, 2012, Lund, Sweden.

Smart phone users typically download their mobile applications from application markets. Application markets aggregate hundreds of thousands of mobile applications in a single location for users to purchase or download free of charge. The primary source for Android applications is the Google Play application market, with half a million applications and 15 billion downloads as of May 2012 [3]. Third party Android markets with different standards for including applications can also be used as a source of mobile applications. Finally, smart phone users can directly download applications created by their employers or other organizations that do not make their applications available on one of the application markets.

Users expect application markets to ensure the security of applications [4]. However, the quantity of malware has been increasing exponentially, from 139 malware applications found in the first quarter of 2011 to over three thousand pieces of malware found in the first quarter of 2012 [5]. Malware is not the only source of security problems in mobile applications. As with desktop and web applications, mobile applications may contain vulnerabilities that allow them to be exploited by attackers to obtain confidential data, modify the integrity of software or data, or deny availability of data and systems. Due to the extensive use of web services in mobile applications, some of these vulnerabilities are of the same types as impact web applications, including Cross-Site Scripting (XSS) and SQL Injection (SQLi) vulnerabilities, but other types of vulnerabilities are specific to mobile applications. Programs designed to detect malware typically do not detect unintentional security flaws in applications.

Smart phones offer attackers new abilities to obtain funds from users, such as premium rate SMS messages and foreign phone calls, as well as the use of new payment systems. Smart phones function in a different network environment from traditional PCs, one in which network access is considerably more expensive and in which the mobile network operator has a high degree of control over the device, making it more difficult for the user to manage security updates. Due to their small size and high mobility, smart phones are easily lost and stolen, increasing the vulnerability of applications to attackers with physical access to the device.

While smart phones offer additional exploitation opportunities, these systems provide security controls that applications running on traditional platforms typically do not have. Android applications run in a sandbox, each with its own user and virtual machine. The Android platform also has an additional permission system on top of the traditional Linux permission system. When installing an Android application, the user can see a list of permissions to access data and services that the application requests and can cancel the installation if the permissions requested are excessive. However, only 17% of users pay attention to permissions during application installation, since most users find permissions confusing and do not have the ability to connect permissions to security risks [7]. As a result, users freely install applications that request permissions that they do not need but which could be exploited if an attacker found a vulnerability.

While much research has been published on mobile application malware, this is the first empirical study of vulnerabilities in mobile applications. While malware is an important problem, less than 1% of Android applications are malware of some type. However, almost all mobile applications likely contain some type of vulnerability as almost all applications outside the mobile space contain vulnerabilities. Furthermore, the problem of vulnerable applications is growing due to the rapid proliferation of mobile applications and the simultaneous growth in the size and complexity of mobile applications. Like many web applications, developers release new versions of mobile applications rapidly, resulting in little time being available for security assurance practices. As a result of these trends, the ability to focus security assurance efforts on a subset of an application's classes would be beneficial to developers.

This paper will attempt to build a model to predict vulnerable classes in a specific Android application and to assess the ability of the model to correctly predict vulnerable components in future versions of the application. Developers can use such a model to focus security inspection and testing efforts. We will build the model using a variety of source code metrics, including size, complexity, and object-oriented metrics. We will also analyze the distribution and evolution of vulnerabilities found in the application.

The rest of the paper is organized in the following manner. Section 2 presents the research methodology and the support-vector machine model used for vulnerability prediction in this paper. Section 3 discusses the distribution and evolution of vulnerabilities in the mobile application studied, while section 4 describes results of the vulnerability prediction model. Section 5 discusses threats to validity, while section 6 summarizes related work in vulnerability prediction. Section 7 finishes the paper with a discussion of conclusions and future work.

2. RESEARCH METHODOLOGY

We considered approximately 200 open source applications that were available on the Android application market as of May 1, 2011. Our consideration was limited to open source applications, as we needed application source code for our analysis. We characterized each application with respect to a number of criteria, including popularity of the application (as number of downloads), size of the project (as lines of code), and activity of the project (as number of versions in the source code repository.) We also assessed whether the applications could be analyzed for security vulnerabilities with the static analysis tool of our choice.

We selected the second largest project for inclusion in this study. The largest application was not analyzed because of a major refactoring in the code base that makes the application's vulnerability trends particularly erratic, which in turn makes the vulnerability modeling a challenging task.

Table 1: Versions of FBReader for Android

Version	Month
0.7.6	Oct 2010
0.99.0	Jan 2011
1.0.0	Apr 2011
1.1.0	Jun 2011
1.2.0	Oct 2011

The selected application was FBReader, a free e-book reader written in Java[1]. The application contained about 39K lines of code and has evolved over more than 3700 revisions in the code repository. Classes found in the repository that belonged to external libraries were ignored. Only classes in the `org.geometerplus.fbreader` namespace were analyzed. On the date of writing, the application has received very good user feedback (4.7 out of 5 from 32K feedbacks) and it has been downloaded more than a million times.

From the FBReader source code repository we downloaded five versions of the source code. As shown in Table 1, we selected one publicly released version from each quarter in the period from October 2010 to October 2011.

We identified vulnerabilities in each version using Fortify's Source Code Analyzer (SCA), an automated static code analysis tool[2], which scans application source code and generates a report describing all of the vulnerabilities found in the source code, with locations and descriptions.

We also computed a rich set of object-oriented code metrics for each version. We used the JHawk 5 tool[3], which collects about 40 different metrics for each class, including lines of code, coupling, cyclomatic complexity, depth of the inheritance tree, and so on. JHawk has been used in several fault-prediction papers.

Finally, we used Support Vector Machines (SVM) to build a classification model that predicts whether a Java class is likely vulnerable (outcome) by using the code metrics as features (a.k.a. predictors).

2.1 Static analysis

We used a static analysis tool to count vulnerabilities instead of using reported vulnerabilities for several reasons. The primary reason is that only about a hundred vulnerabilities for Android applications have been published in vulnerability databases. Combined with the fact that there are over half a million Android applications in the Google Play application market, it is clear that few applications have a vulnerability history. Furthermore, most published vulnerabilities for Android applications are incomplete, with the National Vulnerability Database vulnerability listing describing vulnerabilities as having "unknown impact and attack vectors." Of the applications represented in vulnerability databases, most have only a single vulnerability reported and none have sufficient historical data to make building a prediction model based on reported vulnerabilities possible.

The second reason for using vulnerabilities reported by an automated tool is that static analysis is an objective and repeatable technique for counting vulnerabilities. Static anal-

[1] `http://www.fbreader.org/FBReaderJ`
[2] `https://www.fortify.com/products/hpfssc/source-code-analyzer.html`
[3] `http://www.virtualmachinery.com/jhawkprod.htm`

Table 2: JHawk5 class-level metrics (partial list)

Metric	Description
NOMT	Number of methods in class
LCOM	Lack of Cohesion of Methods
AVCC	Average Cyclomatic Complexity of all the methods in the class
NOS	Total Number of Java Statements in class
UWCS	Unweighted class size
INST	Number of instance variables (or attributes) defined in this class
PACK	Number of packages imported by this class
RFC	Response for class
CBO	Coupling Between Objects
CCML	Total number of comment lines in the class
NLOC	Total number of Lines of Code in the class

ysis tools apply the same algorithms and rulesets each time the tool is run. Using static analysis enabled us to collect a sufficient number of vulnerabilities for analysis, along with detailed information about the location and type of each vulnerability. However, some SCA vulnerabilities are false positives, which are discussed with other threats to validity below. As with lists of vulnerabilities from vulnerability databases, the list of vulnerabilities reported by static analysis tools probably do not contain all vulnerabilities in an application, so false negatives are an issue with any vulnerability study.

Vulnerabilities reported by automated static analysis tools have been shown to correlate with exploitable vulnerabilities found in both traditional desktop and server software [8] and in web applications [9], so using them as a proxy for reported vulnerabilities in an application domain where insufficient historical vulnerability data is available is reasonable.

We used version 5.10.2 of the Fortify Source Code Analyzer as our static analysis tool. This version of the tool has options for scanning mobile applications on the Android platform. The tool reports not only the vulnerability location, but also its category and severity. Eleven vulnerability categories were reported for FBReaderJ, while two severity categories out of five possible were reported. We used custom ruby scripts to automate the execution of the tool and parsing of its XML output into a CSV format that we could import into our statistical analysis scripts.

To conclude, we computed the value of the binary dependent variable **IsVulnerable** as follows. For each Java class, the variable is equal to 0 is no vulnerability warnings are reported for that class. Otherwise, the variable is equal to 1. In our model, we do not make any distinction among warnings of different severity levels or different categories.

2.2 Software metrics

The full list of metrics and the description of how the metrics are computed are both available with the JHawk tool distribution, which is licensed for free for academic use. In Table 2, we summarize some of the most important metrics that are computed by the tool. In the model, we use the whole set of metrics generated by the tool [4], including the ones mentioned here.

[4] `http://www.virtualmachinery.com/Jhawkmetricslist.htm`

Table 3: Cross-validation for the trained model

Fold	Accuracy (%)
1	67.4
2	65.1
3	79.1
4	81.4
5	88.4

A handful of code metrics computed by JHawk have been used as features in our model. In particular, with reference to the JHawk naming, NSUP (number of super classes) is equivalent to DIT (depth of the inheritance tree) and HBUG (cumulative Halstead bugs) is equivalent to HVOL (Halstead volume). Further, MPC is discarded because it is undocumented (even the meaning of the acronym is not given).

2.3 Modeling

We used Support Vector Machines (SVM) in order to build a binary classifier. In particular, we used the SVM implementation packaged in the `e1071` library of the R statistical tool.

As a training set, we used the data (vulnerabilities and code metrics) from the first of our five versions of FBReader, version 0.7.6 released in October 2010. We used a radial basis function (RBF) as kernel for the SVM; therefore, we built a nonlinear classifier. Through a grid search of the parameter space, we selected the following parameters for the kernel function: *cost* is set to 100 and *gamma* to .001. The training set contains a total of 215 Java classes, of which 59 are labeled as vulnerable.

To assess the quality of the training result, we performed a cross-validation with five folds. Table 3 reports the accuracies of each fold. The overall accuracy of the model is 76.3%.

The model built during the training phase was tested on the subsequent four versions. We assessed model performance (in terms of prediction power) by means of three indicators:

- *Accuracy* is the rate of correct results, i.e., the sum of true positives (TP, classes containing weaknesses that are correctly classified as vulnerable) and true negatives (TN, non-vulnerable classes that are correctly classified as negatives) over the total number of classifications.
- *Precision* is the probability that a class classified as vulnerable is indeed vulnerable. It is computed as the ratio $TP/(TP+FP)$, where FP is the number of false positives, i.e., Java classes erroneously classified as vulnerable.
- *Recall* is the probability that a vulnerable class is actually classified as such. It is computed as the ratio $TP/(TP+FN)$, where FN is the number of false negatives i.e., Java classes erroneously classified as not containing vulnerabilities.

3. VULNERABILITY TRENDS

In October 2010, FBReader 0.7.6 had 215 classes, 59 (27.4%) of which contained vulnerabilities reported by Fortify SCA. By October 2011, FBReader 1.2.0 had grown to 265 classes,

Table 4: Most Vulnerable FBReader Classes

Rank	Class	# vulns
1	OPDSXMLReader	126
2	LitResXMLReader	30
3	PluckerTextStream	20
4	OPDSLinkXMLReader$LinkReader	19
5	PalmDocLikeStream	16
6	PluckerBookReader	14
7	OEBBookReader	12
7	OPDSLinkXMLReader	12
8	NCXReader	10
9	OEBMetaInfoReader	9
9	OpenSearchXMLReader	9

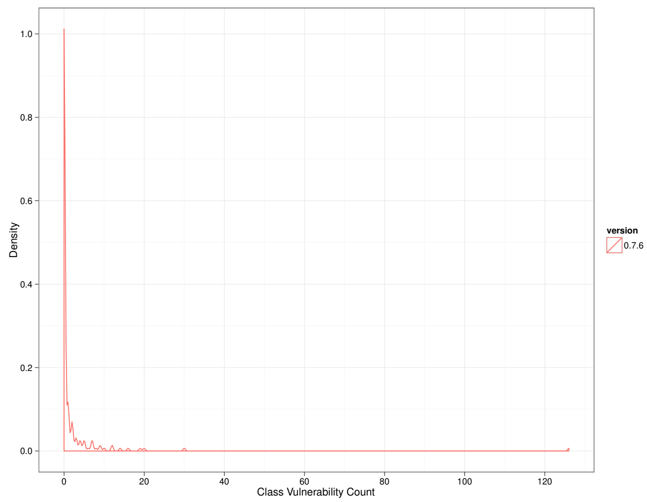

Figure 1: Distribution of Vulnerabilities

20.8% of which were vulnerable, indicating an improvement in security as the application evolved and grew in size. Of the 27.4% of classes that contained vulnerabilities in version 0.7.6, 30.5% have only a single vulnerability and only 9 classes out of 215 have 10 or more vulnerabilities. See Table 4 for a list of the most vulnerable classes in the October 2010 version.

The complete distribution of vulnerabilities across classes for version 0.7.6 is displayed in Figure 1 as a density function. The density function resembles a power law, with a high peak at zero vulnerabilities that declines rapidly. The density function reaches the x-axis after 20 vulnerabilities, with only two bumps in the long tail that ends with class `OPDSXMLReader` containing 126 vulnerabilities. The distribution of vulnerabilities for the other four versions looks almost the same at this scale, with only slight different bumps in the long tail indicating the handful of classes with a very high but slightly different number of vulnerabilities.

While the distribution of vulnerabilities in later versions follows a similar power law to the density function shown in Figure 1, the number of vulnerabilities and the number of classes change over time. The number of vulnerabilities ranges from a high of 410 in version 0.7.6 to a low of 365 in version 1.1.0, with a slight increase to 381 vulnerabilities in the final version, 1.2.0. The top line in Figure 2 shows the change in vulnerabilities between versions.

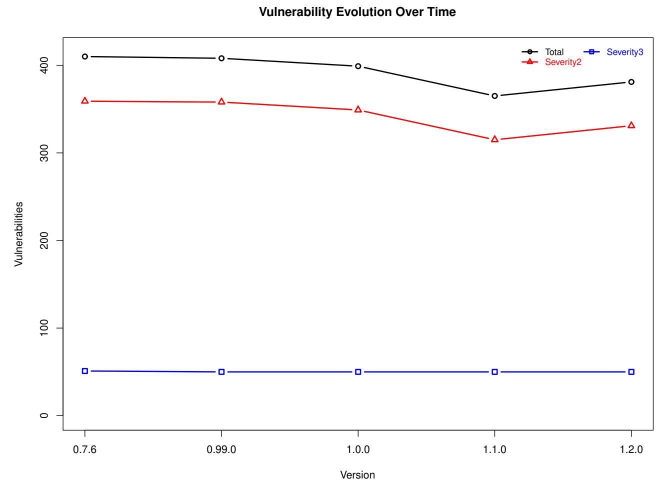

Figure 2: Evolution of Vulnerabilities

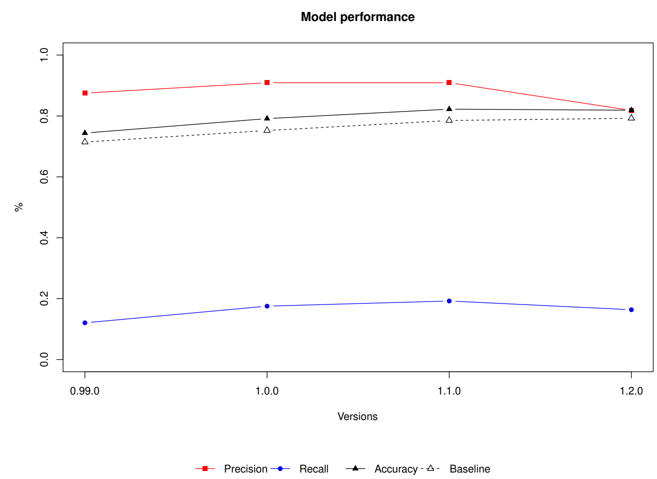

Figure 3: Prediction performance

Fortify SCA reports the severity of vulnerabilities in a range from 1.0 to 5.0. For FBReader, all vulnerabilities were classified as severity level 2.0 or 3.0. Figure 2 displays the evolution of vulnerabilities through each version of FBReader that was analyzed, showing that the number of severity 3.0 vulnerabilities held constant between versions. As a result, changes in the total number of vulnerabilities mirrored the changes in severity 2.0 vulnerabilities.

4. PREDICTING VULNERABILITIES

Our results show that is possible to build a (binary) prediction model with high accuracy and high precision.

Figure 3 shows the precision, recall and accuracy of the prediction model over the four versions used for testing.

The solid black line describes the accuracy of the model, while the dotted black line shows the accuracy achieved by a classifier that classifies all Java classes as not vulnerable. As there are many more non-vulnerable classes than vulnerable classes, the classifier whose accuracy is shown by the dotted line is to be considered as the baseline to compare the performance of the model. The model we developed is between 6.0% and 8.4% more accurate than the baseline. Note, however, that the distance between the two lines shrinks over time. Overall, our model achieves very high accuracy, with over 80% of correct results.

Figure 3 also shows the precision of the prediction model over the four versions used for testing. Precision ranges from 75.9% to 81.5%. This means that the model is very trustworthy when it flags a Java class as being vulnerable. However, as shown in the figure, the recall is lower (although still in line with the related work). Recall ranges from 37.9% to 42.3%. Therefore, the model is not to be trusted when it predicts that a Java class is not vulnerable.

In summary, the model can be used as a useful tool to prioritize the review of Java classes that are vulnerable with high probability. However, the complete list of classes that are actually vulnerable might be larger than predicted.

Retraining. We also re-trained the model on version 3 and used it for prediction in versions 4 and 5. We compared the performance of the two models on these two versions, finding that using a "fresher" model did not improve the performance indicator significantly. Actually, the precision of the fresher model is 1% worse, accuracy is the same, and recall is only between 2% and 4% better. In summary, the prediction model has a validity of *at least* one year for this application.

Severity. The results shown in Figure 3 also apply to the case where only vulnerabilities of severity equal to 2 are considered. We built a prediction model for this alternative dependent variable, and the performance indicators are the same over the four versions. This is expected, as severity 2 the biggest contributor to the vulnerability count, as shown earlier in Figure 2

5. THREATS TO VALIDITY

As with all empirical studies, our results may not be generalizable to other applications, especially to applications running on platforms other than Android or written in languages other than Java. To generalize the models described in this paper to other applications in different languages, application domains, or sizes would require additional studies.

As mentioned above, vulnerabilities reported by automated static analysis tools have been shown to correlate with exploitable vulnerabilities from vulnerability databases [8, 9], so they are a reasonable proxy for reported vulnerabilities if not exactly the same. While lists of both reported vulnerabilities and static analysis vulnerabilities include false negatives, automated static analysis reports are much more likely to include positives, where vulnerabilities are reported that are not actually present in the code.

The results might also be influenced by the way the JHawk code metrics tool computes source code metrics. The descriptions of how metrics are computed in the JHawk documentation are insufficiently precise to determine the exact algorithm used, and metrics tools are known to produce different results also for standard software metrics like coupling and cohesion. While results may vary between tools, the collection of code metrics is repeatable by running JHawk 5 on the source code.

6. RELATED WORK

Gegick [8] developed models to identify vulnerable components in commercial applications using alert density from the Fortify Source Code Analyzer in combination with code churn and lines of code. Walden and Doyle [12] found correlations between code size and complexity metrics and static analysis vulnerabilities in open source web applications.

Shin et. al. [13] developed models based on complexity, churn, and developer metrics to predict vulnerable files in both Firefox and Linux. Logistic regression models using all three types of metrics together predicted over 80% of the known vulnerable files with fewer than 25% false positives. Other modeling techniques produced similar results.

Neuhaus et al. [10] predicted vulnerable components in the Mozilla open source project by analyzing the import and function call relationship between components. Their prediction model correctly predicted about half of all vulnerable components, with about two thirds of all predictions being correct.

7. CONCLUSIONS AND FUTURE WORK

This paper we demonstrated that vulnerabilities can be predicted using an SVM model based on a set of code metrics for a specific Android application. The classification model exhibits good performance in terms of both accuracy and precision.

In ongoing work, we are replicating this study on a larger pool of applications and a bigger number of versions. The aim is to confirm whether similar results can be obtained from other Android applications. Furthermore, we are interested in generalizing the prediction model by jointly analyzing multiple applications. Finally, this work used SVM to build the prediction model as it is a common classification technique that is used in related work in fault prediction, which makes it interesting for comparison purposes. However, alternative modeling methods, including Bayes and classification trees are also the subject of further study.

8. ACKNOWLEDGMENTS

This research is partially supported by the EU FP7 project NESSoS, the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund KU Leuven, and an NKU project grant and sabbatical.

9. REFERENCES

[1] IDC, "Android Rises, Symbian and Windows Phone 7 Launch as Worldwide Smartphone Shipments Increase 87.2% Year Over Year, According to IDC," Feb 7, 2011, `http://www.idc.com/getdoc.jsp?containerId=prUS22689111`, accessed May 27 2012.

[2] Eric Zeman, "Android, iOS Crush BlackBerry Market Share," Information Week, May 24, 2012, `http://www.informationweek.com/news/mobility/smart_phones/240001008`, accessed May 27 2012.

[3] Ingrid Lunden, "Google Play About To Pass 15 Billion App Downloads? Pssht! It Did That Weeks Ago," Tech Crunch, May 7, 2012, `http://techcrunch.com/2012/05/07/google-play-about-to-pass-15-billion-downloads-pssht-it-did-that-weeks-ago/`, accessed May 27 2012.

[4] Patrick McDaniel, William Enck, “Not So Great Expectations: Why Application Markets Haven't Failed Security,” IEEE Security and Privacy, vol. 8, no. 5, pp. 76-78, Sept.-Oct. 2010.

[5] F-Secure, “Mobile Threat Report Q1 2012,” May 11, 2012, `http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf`, accessed May 27 2012.

[6] Hiroshi Lockheimer, “Android and Security,” Google Mobile Blog, `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, accessed May 27 2012.

[7] Adrienne Porter Felt et. al., “Android Permissions: User Attention, Comprehension, and Behavior,” Technical Report No. UCB/EECS-2012-26, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.pdf`, Feb 11, 2012.

[8] Michael Charles Gegick. 2009. Predicting Attack-Prone Components with Source Code Static Analyzers. Ph.D. Dissertation. North Carolina State University. Advisor(s) Laurie Williams.

[9] James Walden, Maureen Doyle, “SAVI: Static-Analysis Vulnerability Indicator,” IEEE Security and Privacy, Jan. 2012. IEEE computer Society Digital Library. IEEE Computer Society.

[10] Neuhaus, S., Zimmermann, T., and Zeller, A., “Predicting Vulnerable Software Components,” Proc. the 14th ACM Conference on Computer and Communications Security (CCS '07), Alexandria, Virginia, USA, pp. 529 - 540, October 2007.

[11] NVD, `http://nvd.nist.gov/`, accessed May 27 2012.

[12] Maureen Doyle, James Walden, “An Empirical Study of the Evolution of PHP Web Application Security,” International Workshop on Security Measurements and Metrics (Metrisec), Sept. 2011.

[13] Y. Shin, A. Meneely, L. Williams, J. Osbourne, Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities, IEEE Transactions in Software Engineering, to appear, 2011.