Software Vulnerability Prediction using Text Analysis Techniques

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen
IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
{first.last}@cs.kuleuven.be

James Walden
Department of Computer Science, Northern Kentucky University
waldenj@nku.edu

ABSTRACT

Early identification of software vulnerabilities is essential in software engineering and can help reduce not only costs, but also prevent loss of reputation and damaging litigations for a software firm. Techniques and tools for software vulnerability prediction are thus invaluable. Most of the existing techniques rely on using component characteristic(s) (like code complexity, code churn) for the vulnerability prediction. In this position paper, we present a novel approach for vulnerability prediction that leverages on the analysis of raw source code as text, instead of using "cooked" features. Our initial results seem to be very promising as the prediction model achieves an average accuracy of 0.87, precision of 0.85 and recall of 0.88 on 18 versions of a large mobile application.

1. INTRODUCTION

Software security is a crucial concern within the software development process as software vulnerabilities can not only incur additional costs, but also cause severe damages to an organization. It is essential to have the right tools and techniques in order to assess and predict the vulnerability of software components produced by the development team(s).

In this position paper, we propose a novel approach for vulnerability prediction of a software component. Our approach is based on creating and using a prediction model by means of machine learning techniques. Although this idea is not new, most of the existing works are focused on security vulnerability prediction based on various derived features of the source code (e.g., total lines of code, total complexity, code churn, etc.). As opposed to these, we propose an approach that relies on the textual analysis of the source code and treats every monogram in that source as a feature. In the context of this paper we have used a version of an email client application for Android for building the prediction model. We have predicted 18 subsequent versions of the same application with a very good precision, but low recall.

The remainder of our paper is organized is as follows. In section 2, we provide an overview of the related work. In section 3, we present our approach for classifying a component based on the source code. In section 4, we summarize our preliminary findings. Finally, section 5 concludes the paper and provides an outline of the future research path.

MetriSec'12 September 21, Lund, Sweden

2. RELATED WORK

While there is a large body of work on defect prediction, the body of work on vulnerability prediction is smaller. Vulnerabilities differ from defects in that there are many fewer security flaws in code compared to defects, and defect prediction techniques do not directly transfer to the task of vulnerability prediction.

Neuhaus et al. [4] have focused on investigating the correlation between vulnerabilities and import statements. They have successfully leveraged machine learning techniques to predict vulnerabilities based on imports in the context of the Mozilla project. Neuhaus et al. have reported average precision of 0.70 and recall of 0.45.

Zimmermann et al. [8] have investigated the correlation between vulnerabilities and various metrics measuring code churn, code complexity, dependencies, code coverage, organizational measures and actual dependencies. With a statistical significance they have found a weak correlation for each of the investigated metric. The authors have also used logistic regression methods to predict vulnerabilities based on these metrics. The study was performed in the context of a proprietary commercial product, i.e., Windows Vista. The results of the study indicate that most metrics can actually predict vulnerabilities with an average to good precision (median precision was 0.60), but with a relatively low recall (median recall was 0.40).

A study by Shin et al. [5] explored the relationship between complexity, code churn and developer activity metrics with vulnerabilities. The authors have utilized two classification techniques, i.e., linear discriminant analysis and Bayesian network. They have determined that these metrics are indeed predictive of vulnerabilities.

All these approaches rely on extracting certain features from the source code (e.g., complexity, number of imports, code churn, etc.) and using them for building a prediction model. As opposed to these techniques in our approach we propose to use the source code itself for building a prediction model. The advantage of this method is that it does not make any assumptions regarding the impact of a certain feature on the software vulnerabilities. The machine learning techniques have yet to create these features based on the complete code base. The disadvantage of this approach is that the learning

may fail to create any meaningful features. In the following section, we present the proposed approach in detail.

3. OUR APPROACH

As Java is a language, we looked at Java files as text. The starting point for our approach is the source code of a software system that consists of a number of Java files. Each file is transformed into a feature vector where every word (also called a "monogram" in text processing) within that file is treated as a feature.

Before splitting the file source code into a set of words representing the features we run a preprocessing step. Certain blocks in the source code files are likely to pollute the prediction model. Such blocks are, for instance, the comments. Indeed, we believe that it is rather unlikely that comments could have an impact on the vulnerability of a file. Hence, in a preprocessing step we filter out all the comments from the source. For the same reasons, we also filter out all strings and numerical values.

In order to transform the preprocessed source code into a feature vector, we need to tokenize the textual representation of the source into a set of monograms. As a set of delimiting we have chosen to use not only white spaces, but also the Java "punctuation" characters (such as, ". , ;) (} {] [") as well as mathematical and logical operators (such as, "+ - / * ^ | || & && !"). In a feature vector each monogram (i.e., feature) must also have an assigned value. We use the count of a given monogram in a given file source code as its value.

Consider the figure 1 that depicts the HelloWorld.java file.

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Figure 1: Hello World Java File

In order to transform this file into a corresponding feature vector we filter out all the comments from this file as well as the *"Hello World!"* string. What remains from the source of this file is tokenized into a feature vector that treats each monogram as a feature. Hence, the feature vector of the HelloWorld.java file becomes:

class:1, HelloWorldApp:1, public:1, static:1, void:1, main:1, String:1, args:1, System:1, out:1, println:1

where each of the monograms is followed by a count (in this case 1). Note that in this example we do not follow any particular (e.g., SVM) notation.

During the learning phase each file represented as a feature vector also has a vulnerability label assigned to it. We use this training set to build a prediction model. Throughout this paper we consider a binary classification scheme where a file is either classified as *vulnerable* or *clean*. Once the prediction model is created from the training set, we can use this prediction model to predict the vulnerability of arbitrary files each represented as a feature vector.

We have leveraged the concept of the support vector machine (SVM) for both the training phase where a prediction model is built from a set of training examples, and the prediction phase where a feature vector is classified based on the previously built prediction model. In our initial exploration, we have used a radial basis function with a set of parameters (cost and gamma) that are selected by running a grid search algorithm. The precise details of the training algorithm are out of the scope of this paper.

4. PRELIMINARY RESULTS

We have performed an initial exploration of the presented approach using a concrete application. In this section, we briefly present the preliminary results of our investigation.

4.1 Application

Market analysis has shown that consumers are purchasing more smart phones than PCs since the last quarter of 2010 [1]. Hence, a potential vulnerability in any mobile application may affect a huge number of users. Most of these mobile applications are running on the Android platform [7]. This is why we have chosen to investigate the vulnerabilities of mobile applications developed for the Android platform. Repositories containing a large version history of open source mobile applications for the Android platform are readily available and represent an ideal testbed for our approach. For the purposes of our initial exploration, we have selected to use 19 versions of the K9 mail client application spread over the period of 22 months. The timespan between each version is approximately one month. We have used the first version in order to build the prediction model and we have predicted the vulnerabilities of the files of each subsequent version using this prediction model.

In order to assign the vulnerability labels we have leveraged the state-of-the-practice Fortify tool [3] that analyzes the source code for various known types of software security vulnerabilities. Fortify not only spots a vulnerability, but also assigns a severity for each vulnerability found. In our exploratory work, we have treated a file as *vulnerable* if Fortify has assigned any type of vulnerability to it and as *clean* otherwise. By using Fortify we rely on vulnerabilities that are extracted during a static analysis of the source (based on common vulnerabilities and exposures) rather than reported vulnerabilities. There are systematic studies that have shown that there are strong correlations between such static analysis metrics and the quantity of subsequently reported vulnerabilities [6]. Nevertheless, this issue is rather controversial as commercial tools are said to produce high false positives [2].

4.2 Results

We have used the version k9-2.504 to build the prediction model. We assessed the model performance (in terms of prediction power) by means of three indicators:

- *Accuracy* is the percentage of correct results.
- *Precision* is the probability that a file classified as vulnerable is indeed vulnerable.
- *Recall* is the probability that a vulnerable file is classified as such.

Figures 2 and 3 illustrate the initial results that we have obtained. The main observation is that the prediction model scores very high (above 80%) for all three indicators. Figure

2 also shows the positive rate of the application, i.e., the percentage of vulnerable files, which is between 40% and 60%. Therefore, a "naive" classifier that classifies all files as vulnerable (or alternatively as clean) would achieve a precision in the range of 40% to 60% as well. This range is a baseline for the accuracy indicator and our approach performs substantially better compared to the baseline.

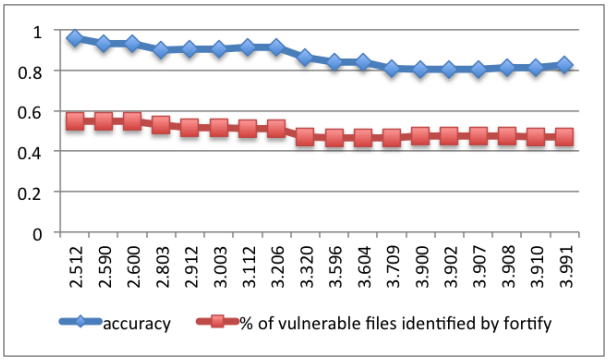

Figure 2: Accuracy vs % of vulnerable files identified by fortify

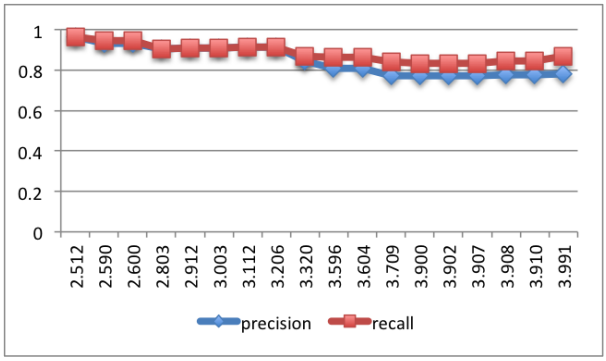

Figure 3: Precision vs recall

Finally, note that the number of files grows substantially from the first training set used to build the prediction model (97 Java files in k9-2.504) to the last version (177 Java files in k9-3.991). Hence, in the testing phase, the model is also classifying many new files that were not present in the training set.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel approach that can predicts the vulnerability of a file based on its source code. As opposed to a number of current state-of-the-art approaches that build a vulnerability prediction model based on a certain characteristic (e.g., software metrics) of the source code, our approach treats each "word" in the source as a feature. We have explored this approach on an open source mobile application, i.e., K9 email client for the Android platform. Our initial results indicate that the proposed approach has very values for accuracy (average of 0.87), precision (average of 0.85) and recall (average of 0.88). These results are very promising and encourage further research in this area.

In the future, we plan to further investigate the presented approach by looking at various alternatives in building the feature vector. We also plan to investigate the possibilities to build a vulnerability prediction model that uses the six-class classification supported by Fortify (i.e., non-vulnerable, vulnerable with severity 1 to 5). Finally, we believe that our approach is complementary to using the existing techniques that use, e.g., internal metrics for building a prediction model. Hence, an even more interesting research track would be to expand our approach to use a feature vector that consists both of the complete source code treated as text and a list of code metrics.

6. REFERENCES

[1] Android rises, symbian and windows phone 7 launch as worldwide smartphone shipments increase 87.2% year over year, according to idc (2011), `http://www.idc.com/`

[2] Austin, A., Williams, L.: One technique is not enough: A comparison of vulnerability discovery techniques. In: ESEM. pp. 97–106 (2011)

[3] Fortify: Fortify. https://www.fortify.com/ (2011)

[4] Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (October 2007)

[5] Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Software Eng. 37(6), 772–787 (2011)

[6] Walden, J., Doyle, M.: Savi: Static analysis vulnerability indicator. IEEE Security and Privacy (to appear) (2012)

[7] Zeman, E.: Android, ios crush blackberry market share (2011), `http://www.informationweek.com`

[8] Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (April 2010)