Open access • Journal Article • DOI:10.1109/TSE.2014.2340398

Predicting Vulnerable Software Components via Text Mining — Source link

Riccardo Scandariato, James Walden, Aram Hovsepyan, Wouter Joosen

Institutions: Katholieke Universiteit Leuven, Northern Kentucky University

Published on: 01 Oct 2014 - IEEE Transactions on Software Engineering (IEEE)

Topics: Software security assurance, Source code and Component-based software engineering

Predicting Vulnerable Software Components via Text Mining

Riccardo Scandariato, James Walden, Aram Hovsepyan and Wouter Joosen

Abstract—This paper presents an approach based on machine learning to predict which components of a software application contain security vulnerabilities. The approach is based on text mining the source code of the components. Namely, each component is characterized as a series of terms contained in its source code, with the associated frequencies. These features are used to forecast whether each component is likely to contain vulnerabilities. In an exploratory validation with 20 Android applications, we discovered that a dependable prediction model can be built. Such model could be useful to prioritize the validation activities, e.g., to identify the components needing special scrutiny.

Index Terms—Vulnerabilities, prediction model, machine learning

1 Introduction

Verification and validation (V&V) techniques like security testing, code review and formal verification are becoming effective means to reduce the number of post-release vulnerabilities in software products [1]. This is an important achievement, as fixing a bug after the software has been released can cost much more than resolving the issue at development time [2]. However, V&V is not inexpensive. An early estimation assessed that V&V adds up to 30% to development costs [3]. Indeed, it has been noted that “it is possible to spend too much on software quality” [4]. In this respect, being able to predict where the vulnerabilities are likely to show up in a code base provides the opportunity to point V&V activities in the right direction and to manage and optimize cost.

Machine learning techniques have been used as means to build such prediction models [5], [6]. Prior research has primarily focused on finding relationships between vulnerabilities and properties of the source code–like cyclomatic complexity–or aspects of the code development process–like developer activity [5]. In these cases, a model is built based on the intuition that, for instance, a more complex software component (or one that is modified very often) is more likely to contain security issues. Less obvious intuitions like vulnerable software being related to the use of certain libraries also turned out to be correct [6]. In the above examples, the choice of the *features* that are used as predictors is determined by the expectations of a knowledgeable individual.

In our work, we investigated a technique that relies less on a particular underlying axiom. Starting from the observation that a programming language is a language after all (like English) and that syntax tokens equate to words, we set out to analyze the source code by means of text mining techniques, which are commonplace in information retrieval. Text mining applied to source code was introduced by Hata et al. [7] for the prediction of software defects and is here applied to the domain of software vulnerabilities. We use the bag-of-words representation, in which a software component (a Java source file in this paper) is seen as a series of terms with associated frequencies. The terms are the features we use as predictors. Hence, the set of features used for modeling is not fixed or predetermined but rather depends on the vocabulary used by the developers. In this sense, this technique is less constrained or biased by an underlying theory of what is a-priori expected to happen.

As its main contribution, this paper explores the value of a technique backed by text mining and machine learning and applies the technique to a relevant class of applications, thus ensuring a potentially high impact in case of success. The approach presented here is applied to the problem of predicting software vulnerabilities. In particular, we analyzed 20 “apps” for the Android OS platform and followed their evolution over time. In total, we analyzed 182 releases, spanning a period of two years.

We applied the above-mentioned text mining technique in a series of three experiments of increasing complexity. In the first experiment, we focus on the first release of each application. We show that it is possible to build a classifier of good quality that predicts whether a file is vulnerable using term frequencies. In particular, we show that the model has good performance in terms of precision and recall. Informally, precision represents the probability that the classification of a file as vulnerable is correct, while recall represents the probability that the model finds a file which is known to be vulnerable. Therefore, high precision implies that the classification

- *R. Scandariato, A. Hovsepyan and W. Joosen are with IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium.*
- *J Walden is with the Department of Computer Science, Northern Kentucky University, Highland Heights, KY, 41099.*

results do not contain noise, while high recall means that the results are complete, which is often more important in the field of security. In the second experiment, we show that such a model is a good predictor of vulnerabilities for the subsequent versions of the applications. Clearly, this is the reason for analyzing 182 releases. In the third experiment, we explore whether prediction models are specific to one application only or can rather be applied across projects. The set-up of the first two experiments is quite common in the related work. The third set-up is used here for the first time in the domain of vulnerability prediction. We have used both Naïve Bayes and Random Forest machine learning techniques in all three experiments.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of the related work. In Section 3, we describe the research methodology. In Section 4, we present the working set of applications. In Sections 5, 6 and 7, we give the results of the three experiments. Finally, in Section 8, we discuss the threats to validity and we conclude in Section 9.

2 Related Work

In this section, we focus primarily on the prediction of vulnerabilities. Clearly, the work on software defect prediction is also relevant, but is also very broad. Therefore, it is not exhaustively covered here. For an extensive survey we refer the interested reader to the work of Catal et al. [8]. A recent study by Shin and Williams [9] provides some evidence that certain types of defect prediction models can be used for vulnerability prediction. Nonetheless, in general, defect prediction models do not directly transfer to the task of vulnerability prediction.

2.1 Vulnerability prediction models

This category of works focuses on classifying software entities (components, classes, modules, etc.) as “vulnerable” or “clean”. In order to compare the results of these studies with each other and with our paper, we defined five dimensions along which each related work has been characterized: the type of prediction features used, the source of the vulnerability data, the type of prediction technique, the applications used for the validation, and the available performance indicators. Table 1 presents a summary of the findings, while the rest of this subsection discusses each approach individually.

Shin and Williams [10], [11] investigated whether complexity metrics may be used to predict the location of security issues. The authors' initial results demonstrated that there is a correlation (albeit weak) between complexity metrics and security problems in the Mozilla JavaScript Engine. Shin et al. [5] further explored the relationship of complexity, code churn and developer activity metrics with vulnerabilities. They used logistic regression and achieved an average recall of 80% with a fall-out of 25% (precision was not reported).

Chowdhury and Zulkernine [12] investigated whether complexity, coupling and cohesion metrics could be used to predict vulnerabilities. The authors empirically validated the proposed approach on fifty-two releases of Mozilla Firefox that were developed over the period of four years. Their approach is based on decision trees and achieves a mean accuracy of 72.85%, mean recall of 74.22%, mean fall-out of 28.51%, and mean F_1 score of 73.00%.

Neuhaus et al. [6] focused on investigating the correlation between vulnerabilities and `include` statements in C. They successfully leveraged machine learning techniques to predict vulnerabilities in the context of one snapshot the Mozilla project (including Firefox and Thunderbird). The authors reported an average precision of 70% and recall of 45%.

Zimmermann et al. [13] found a weak correlation between vulnerabilities and various metrics, including code churn, code complexity, dependencies, and organizational measures. In the context of Windows Vista, they built two different predictors. The first was based on conventional metrics (i.e., code churn measures, code complexity metrics, dependency measures, code coverage measures and organizational measures) and resulted in a median precision of 66.7% and median recall of 20%. The second prediction model was based on dependencies between binaries and has resulted in slightly lower precision (60%), but higher recall (40%).

Gegick et al. [14] investigated whether non-security failure reports could be used to predict whether a given component is vulnerable. In the context of a Cisco software system, the authors found a 0.4 correlation between security faults and non-security failures. Using CART (Classification And Regression Trees), the authors ranked all components in descending order of their probabilities of being vulnerable. They demonstrated that 57% of all vulnerable components are in top 9% of the ranking, but with a 48% fall-out.

Nguyen and Tran [15] proposed an approach based on dependency graphs to predict vulnerable components. The authors validated their approach on two versions of the Firefox JavaScript Engine (JSE). For JSE version 1.5 the average precision is 68.01%, recall is 59.53%, fall-out is 8.70% and accuracy is 84.61%. For JSE version 2.0 the average precision is 60.60%, recall is 60.00%, fall-out is 10.12%, and accuracy is 84.08%.

Smith and Williams [16] investigated the predictive power of the so-called SQL hotspots, i.e., places containing a large number of SQL statements. The authors determined that the more SQL hotspots a file contains per lines of code, the higher the probability that the file will contain any type of vulnerability. This vulnerability prediction model scores between 2% and 50% for precision and 10% and 40% for recall for the WordPress blog engine, and between 4% and 100% for precision and 9% and 100% for recall for the WikkaWiki application.

TABLE 1
Vulnerability prediction models in the related work.

Study	Predictors used	Vulnerability sources	Prediction technique	Applications	Performance
Shin et al. [5]	software metrics, code churn, developer activity metrics	MFSA, Red Hat Bugzilla, Read Hat package management system (RPM)	Logistic regression	Firefox, Red Hat Linux Kernel	Firefox — precision: 3%, recall: 79-86%, fall-out: 22-25%; Red Hat Linux Kernel — precision: 5%, recall 80-90%, fall-out: 22-25%
Shin et al. [9]	software metrics, code churn, developer activity metrics	MFSA	Logistic regression	Firefox	precision: 12%, recall: 83%
Shin et al. [10], [11]	code complexity metrics	MFSA / CVE / Bugzilla	Logistic regression	Firefox JS Engine	recall: 3-93%, accuracy: 43-98%, fall-out: 0-58%
Chowdhury et al. [12]	complexity, coupling and cohesion metrics	MFSA / Bugzilla	Naïve bayes, C4.5 Decision Tree, Random Forest, Logistic regression	Firefox	C4.5 decision tree — mean precision: 72%, mean recall: 74%, mean accuracy: 73%, mean fall-out: 29%
Neuhaus et al. [6]	imports, function calls	MFSA	SVM	Mozilla	precision: 70%, recall: 45%
Zimmermann et al. [13]	code churn, code complexity, dependency measures, code coverage, organizational metrics, actual dependencies	NVD	Logistic regression, SVM	Windows Vista	median precision: 60-67%, median recall: 20-40%
Gegick et al. [14]	non-security failure reports	Cisco security reports	CART	Cisco software system	recall: 57%, fall-out: 48%
Nguyen et al. [15]	component dependency graphs	MFSA / CVE / Bugzilla	Bayesian network, Naïve Bayes, Neural networks, Random forest, SVM	Firefox JS Engine	precision: 61-68%, recall: 60-61%, accuracy: 84-85%, fall-out: 9-10%
Smith et al. [16]	SQL hotspots	Trac issue reports	Logistic regression	WordPress, WikkaWiki	Wordpress — average precision: 28%, average recall: 24%; WikkaWiki — average precision: 62%, average recall: 39%

2.2 Static code analysis and vulnerabilities

In our investigation, we used the Fortify Source Code Analyzer (SCA) security-oriented static analysis tool to identify potential vulnerabilities in source code rather than using the reported issues contained in vulnerability databases, like the NVD (nvd.nist.gov). Often, the above-mentioned databases are believed to be reliable sources, which are unlikely to contain false positives. In fact, the research community is still debating about what are the reliable sources to be used as "ground truth" for prediction models. Massacci and Nguyen [17] have empirically demonstrated for Mozilla Firefox that using different vulnerability databases can lead to completely different results. Further, Martin and Christey [18], members of the CVE (Common Vulnerabilities and Exposures) Editorial Board, have outlined how vulnerability databases contain several types of bias and are easily misused.

A common criticism to static analysis tools is that they can produce many false positives [19]. However, recent studies have shown that vulnerability warnings from static analysis tools are not so unreliable after all. Walden and Doyle [20] demonstrated that the warnings generated by the Fortify SCA tool are strongly correlated to (and a good proxy of) NVD vulnerabilities. Gegick et al. [21], [22] also demonstrated a statistically significant correlation between static analysis warnings and vulnerabilities. Nagappan and Ball [23] reported similar correlations in the area of software defect prediction, i.e., defect density discovered via static analysis is related to pre-release defects determined by testing. Edwards and Chen [24] found a statistically significant correlation between the change in number and density of static analysis warnings reported by Fortify SCA with the change in the rate of publication of NVD vulnerability entries. Finally, Zheng et al. [25] determined, based on three large-scale industrial systems, that static analysis is an effective technique at identifying assignment and checking faults that have the potential to cause security vulnerabilities.

2.3 Text mining and vulnerabilities

A few approaches are related to our work as they leverage text mining techniques and treat all or parts of the source code as text. However, most work focusses on defect prediction and not on vulnerability prediction, which is the topic of our work.

Hata, Mizuno et al. [7], [26] used text features and spam filtering algorithms to predict defects in software. In their earlier work [26], the approach was used to predict defects on the ArgoUML and Eclipse BIRT applications with precision values of 72%-75% and recall values of 70%-72%. In later work, they experimented with five open source Eclipse projects, achieving maximum precision and recall values of 40% and 80% respectively.

Aversano et al. [27] analyzed the source code of the changes as text in order to build a predictor to determine whether the introduced changes are buggy. The authors determined that the use of K nearest neighbors (kNN) technique results in a significant trade-off in terms of precision and recall. The approach was validated using two open source Java applications, yielding precision and recall values of 59%-69% and 59%-23% respectively.

Kim et al. [28] have gone even further by using certain change metadata, complexity metrics, change log metrics and filenames in addition to the change source code to build a prediction model. They validated this approach on 12 open source projects and achieved average accuracy and average recall values of 78% and 65% respectively.

Gruska et al. [29] performed a large scale study by mining more than 6000 open source Linux projects to obtain sixteen million properties reflecting normal interface usage. New projects can be checked against these properties to detect anomalies. The authors validated their approach based on a sample of 20 projects, where 25% of the top-ranked anomalies uncovered actual defects.

All the above-mentioned approaches belong to the domain of defect prediction. In the area of security, Bosu and Carver [30] have presented some preliminary work showing how the comments added to the code by code reviewers can hint to the presence of software vulnerabilities.

Yamaguchi et al. [31] have mined the source code of the Linux kernel and of the FFmpeg library. For each C method, they have extracted so-called API symbols, i.e., the identifiers of function calls and the data types used in the method. Thise info is used to cluster methods by similarity, so that recurring flaws can be identified. This work can be seen as a specialization of our approach.

There are also more simple approaches to text mining than do not leverage on machine learning but, rather, look for predefined patterns in code. Several static code analysis employ this technique. For instance, the ITS4 Security Scanner (http://www.cigital.com/its4) searches the code for simple patterns, most of them being about APIs in UNIX or Windows-based systems.

A couple of approaches leverage the analysis of bug reports in order to predict vulnerable software components. Unlike our work, these approaches do not mine the source code but rather the natural language contained in the bug reports. Gegick et al. [32] designed an approach that uses text-mining techniques to train a model to identify which bug reports are security-relevant. The approach was applied on a large Cisco software system and identified 78% of the security-relevant bug fixes. Bozorgi et al. [33] have proposed a slightly different approach where vulnerability disclosure reports are mined in order to rank the vulnerabilities in terms of their exploitability.

3 Research Methodology

In order to enable replication of this study, all of the experimental material is available online, including the data we used and additional details about the procedures we followed [34].

Our overall research *goal* is to build a prediction model in the form of a binary classifier that can predict whether a software component is likely to be vulnerable. This type of prediction (i.e., with the granularity of a component) is standard in the related work, as the goal is to highlight the areas of the code base that deserve particular attention and not to identify the code line where a vulnerability is located. As in other studies [6], software components are Java files in the scope of this work. Working at the level of files is a convenient choice that simplifies the labeling of the samples used by the learners during the training phase, as the static code analyzer annotates vulnerability warnings with file location information.

A software component is defined as vulnerable[1] if one or more vulnerability warnings are reported for that component by a static code analyzer. Hence, the dependent variable is defined as follows:

$$vulnerable = \begin{cases} 1 & \text{if number of warnings} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The independent variables (features) are the terms present in the source code of the software component, together with their frequencies. Hence the model looks like: $vulnerable = f(term\ frequencies)$

For each file that is classified by the model as either vulnerable (1) or clean (0), we compare the prediction with the observed value, i.e., the real value that is based on the static analysis. Accordingly, the prediction can be a *true positive* (TP, the file is both predicted and observed as vulnerable), a *true negative* (TN, the file is clean for both the predictor and the static code analyzer), a *false positive* (FP, the file is predicted as vulnerable albeit is clean), or a *false negative* (FN, the file is predicted as clean but in reality is vulnerable).

3.1 Performance Indicators

Our performance indicators of choice, *precision* ($\mathcal{P}$) and *recall* ($\mathcal{R}$), are commonly used in information retrieval and widely accepted in the literature. They are defined as follows:

$$\mathcal{P} = TP/(TP + FP)$$

$$\mathcal{R} = TP/(TP + FN)$$

Precision represents the probability that a file classified as vulnerable is indeed so. High precision is desirable because it means that the results returned by the classifier do not contain false alarms and, hence, no time is wasted on scrutinizing files that are actually clean. Recall (sometimes called probability of detection, true positive rate, or sensitivity) represents the probability that a vulnerable file is successfully classified as such. A high value of recall is desirable because it means that the results returned by the classifier are very complete and the risk of not scrutinizing a vulnerable file is minimal. The two indicators can be combined in a single value, called the *F score* ($\mathcal{F}_\beta$):

$$\mathcal{F}_\beta = (1 + \beta^2)\mathcal{P}\mathcal{R}/(\beta^2\mathcal{P} + \mathcal{R})$$

$\mathcal{F}_1$ (i.e., $\mathcal{F}_{\beta=1}$) is the harmonic mean weighting recall and precision evenly. In the context of security, recall is often considered to be more important because, in general, it is preferable that positives are not disregarded. Therefore, $\mathcal{F}_2$ is a more useful indicator as it weights recall higher than precision.

[1] In the literature, researchers also use the term “vulnerability-prone component”.

In order to facilitate the comparison of our results with the results of other studies, we also report the *fall-out* ($\mathcal{O}$), which represents the probability that a false positive is generated by the model. Therefore, lower values for this indicators are preferable. This quantity is also known as the false positive rate and is defined as:

$$\mathcal{O} = FP/(FP + TN)$$

Our *benchmark* to judge a high quality model is 80 percent for both precision and recall.

As shown in Section 2 (and Table 1), these values are never simultaneously achieved in the related work. The highest precision for vulnerability prediction is 72% and is achieved by Chowdhury et al. [12] (with a corresponding recall of 74%). The highest value for recall is in the range of 80-90% and is achieved by Shin et al. [5]. However, this value is obtained at the expense of precision, which is equal to 5% at best. Therefore, the chosen benchmark, although being somewhat arbitrary, is a challenging one.

3.2 Selection of Applications

We decided to focus on vulnerabilities of mobile applications for the Android platform due to the popularity of mobile devices in general and of the Android platform in particular. The high popularity of a platform means that the potential impact of any vulnerability impacts a huge number of users. We only considered open source applications, as we needed application source code for our analysis. We selected our applications from the *F-Droid* repository (f-droid.org) of free and open source Android applications. Since this study commenced in early 2012, we selected applications for which a sufficient number of versions were released between the first day of 2010 and the last day of 2011.

Our *selection criteria* for applications included the programming language, application size, and the number of versions released. As different languages use different keywords and naming conventions, we focused our text mining analysis on applications written in Java, which is the standard programming language used for Android application development. In order to develop reliable models, we needed applications larger than a certain minimum size, so we restricted our selections to applications with more than one thousand lines of Java source code. As we needed multiple versions of each application for our analysis, we selected applications that had at least five versions for which we could download source code.

Out of over 200 applications present in the F-Droid repository as of the end of 2011, we found ten applications that met all of our selection criteria.

In order to increase the scope of our investigation, we also selected ten applications that come pre-installed with the *Android OS*. These are basic utilities that, for instance, support the users in managing their agenda, contacts, and pictures. We have considered the versions that have been released with the OS in the same time window mentioned before, which corresponds to six versions spanning from Éclair to Ice Cream Sandwich.

3.3 Dependent Variable

To identify vulnerable files, we used HP FortifySCA[2], an automated static code analysis tool that has specific support for Android applications written in Java. In general, professional-grade static analysis tools are rather expensive. When these tools are not available to engineers, the utility of the prediction model presented in this paper increases. We used version 5.10.2 of SCA in our analysis, with Fortify Secure Coding Rules version 2012-1. This tool scans the source code of an application, including the Android XML manifest file, and generates a report describing all the vulnerabilities that were discovered. Each warning is tied to a location, i.e., the line number of a file where the vulnerability is present. We use this information to label files as vulnerable. As mentioned, a file is considered vulnerable if SCA reported at least one vulnerability warning associated with that file. We scanned each version of each application in our working set with SCA and labeled all files as either vulnerable or clean, accordingly.

The tool also reports the type of vulnerability, such as Cross-Site Scripting or SQL Injection, as well as the severity of the discovered vulnerability, rated on a scale form 1 (low) to 5 (high). We plan on using these additional data in future work.

From a computational perspective, scanning an application is a time consuming task. As a reference, it took about 51 minutes to scan the initial version of a large application (K9Mail) on a PC with a 2.8GHz processor and 8GB of RAM.

Why SCA? We chose to use SCA because too few vulnerabilities related to Android applications have been reported in vulnerability databases at this point. For instance, NVD (nvd.nist.gov) contains only 7 vulnerability reports which are related to Android apps.

This can be explained in terms of the young age of the Android platform (or Google not submitting NVD entries). Much as in the early days of PC security, attackers in the mobile space are focused on attacking the operating system and developing malware. As these avenues of attack become more difficult over time, we expect attackers to focus on applications as they have in the PC space.

Furthermore, as illustrated in Section 2, empirical evidence suggests that vulnerability warnings obtained from static analysis tools are predictors of and correlate with actual vulnerabilities in software. Therefore, it is reasonable to use static analysis warning for the construction of the prediction model. However, manually validating the vulnerability reports produced by SCA could be useful. We have done this for two applications

[2] http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812

used in this study (AnkiDroid and Mustard) and removed the false positives we found. These higher quality data have been used to support our results. An extensive manual validation of all applications and versions is an endeavor that would require the collaboration of the research community. We incentivize this by making our data publicly available [34].

3.4 Independent Variables

The starting point in our study is the source code (including comments) of a software application that consists of a number of Java files. Each Java file is tokenized into a vector of terms (a.k.a. monograms in text processing terminology) and the frequency of each term in the file is counted. The frequencies are not normalized to the length of the file. This procedure has been attempted in our early experimentation and caused a deterioration of performance. The routine used for tokenization uses a set of delimiters that includes white spaces, Java punctuation characters (such as comma and colon) and both mathematical and logical operators. The routine is implemented in R and is available at [34].

Listing 1. Source code in file HelloWorldApp.java

```
/* The HelloWorldApp class prints "Hello World!" */
class HelloWorldApp {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

For instance, Listing 1 would be tokenized and transformed into the feature vector of Listing 2, where each monogram is followed by a count.

Listing 2. Feature vector for file HelloWorldApp.java

```
args: 1, class: 2, Hello: 2, HelloWorldApp: 2,
main: 1, out: 1, println: 1, prints: 1, public: 1,
static: 1, String: 1, System: 1, The: 1, void: 1,
World: 2
```

From a computational perspective, creating the feature vectors for one version of a large application (K9Mail) took an average of 40 seconds (on a PC with a 2.5GHz processor and 8GB of RAM).

3.5 Machine Learning Techniques

In this paper, we rely on machine learning techniques for building prediction models. In a classic setup, a set of feature vectors (from the Java files) and the corresponding labels (either vulnerable or clean) are used as the training set, i.e., the data set that is used to build the prediction model. The model is subsequently applied to a different set of feature vectors called the testing set. The predicted classifications of the files in the testing set are then compared with their real labels in order to compute the performance indicators, as explained earlier.

There are various learning techniques and their performance depends greatly on the characteristics of the data to be classified. It is normal that some techniques work better than others. In fact, different techniques make different assumptions about the data. Therefore, a technique can be unfit if the data set at hand does not comply with those assumptions. In machine learning, this phenomenon is known as the "no free lunch theorem" [35]. The conditions for which a method is expected to work are not theoretically understood in the machine learning community. However, some heuristics exist. For instance, for small data sets (e.g., under 1000 instances in our case) methods that have strong bias and small variance tend to work better[3].

In this study, we have initially explored five, well-known learning techniques: Decision Trees, k-Nearest Neighbor, Naïve Bayes, Random Forest and Support Vector Machine (SVM). These techniques are rather common in the context of vulnerability prediction, as illustrated in Table 1. At the beginning of our experiments, we discovered that the best results are obtained with Naïve Bayes and Random Forest. Therefore, we have focused our efforts on these two techniques and this paper elaborates on them only. Incidentally, we have chosen two techniques (and not one) because we do not want our conclusions to be biased by the choice of a particular learner. Note that other techniques might also work well, provided that the necessary tuning of the parameters is performed. However, we decided to focus on simpler learners that require fewer configurations, as advised by Domingos [36]. Yet, we are not promoting some algorithms over others. Plainly, this paper offers two defaults that are likely to work out of the box.

The experiments are conducted with the Weka tool and the machine learning algorithms that are implemented in its default library. We used the default parameters of the algorithms, with the exception of the Random Forest algorithm, which is configured to generate 100 random trees (instead of 10). From a computational perspective, the Random Forest algorithm is the most expensive. For reference, we report that it takes 38 seconds to perform the cross-validation on a large application (K9Mail). These numbers were obtained on a PC with a 2.4GHz processor and 8GB of RAM.

We found that the *discretization* of the features significantly improves the performance of the prediction models. Discretization refers to the process of transforming a numeric range into nominal classes. For instance, instead of using the number of occurrences of the term 'if' in each file, the discretization algorithm could map the feature to two bins: whether or not the term occurs more than 5 times. The number of bins and the cutoffs depend on both the discretization algorithm and the distribution of the values. In our work, we used the the method proposed by Kononenko [37]. Only a small fraction of the terms (1.5% on average) survive the discretization process. Indeed, many features are mapped to one bin only ('All values') and therefore

[3] Bias is related to the expressivity of the model. E.g., a linear model has stronger bias that a polynomial model. Variance refers to the tendency to generate a completely different model when the data set is perturbed slightly.

get effectively eliminated as they lost any discriminative power.

3.6 Research Questions and Experiments

In this study, we investigated three research questions.

RQ1: Can a prediction model be built? First of all, we were interested in finding out whether it is feasible at all to build a vulnerability prediction model of good quality starting from the features described earlier in Section 3.4.

To answer this question, in the first experiment we have focused on one version only (the first one in our data set) of each application. For each application we ran a stratified 10-fold cross-validation experiment, with the support provided by the Weka tool. The tool randomly divides the files in 10 subsets (folds), with the constraint that the proportion between vulnerable and clean files is the same in all folds. Iteratively, each fold is retained as the testing set and the other folds are used as the training set to build a model, with both Naïve Bayes and Random Forest. As a way of averaging the 10 models that have been built, the tool computes the performance indicators over the collection of the predictions made over the 10 testing folds.

RQ2: Can predictions in the future be made? Moving on, we were interested in knowing whether a prediction model built for one version can predict the vulnerable files in the subsequent versions of the same application, with good performance according to our benchmark. If so, we also wanted to understand at what point in the future the model loses some of its prediction power (i.e., performance loss of 10% in terms of $\mathcal{F}_2$ score) and training a fresh model might become desirable.

To answer this question, in the second experiment we analyzed all versions. For each application, we used all files in the first version to train a prediction model (with both learners) and we applied the model to predict the vulnerability of the files of all subsequent versions. Therefore, each subsequent version represents a testing set, on which the performance indicators are computed. We do not look at the relationships between the versions (e.g., how the structure of the code base has evolved). These additional features related to the code evolution might be interesting for future work but are out of scope in this study.

RQ3: Can cross-project predictions be made? Finally, we wanted to investigate whether it is possible to apply with good performance the prediction model of one application to other applications. This question relates to the existence of a generalized model that can predict vulnerable components in multiple applications.

To answer this question, we considered the first version of each application. We trained a prediction model using all the files in (the first version of) one application. The model has been then tested on (the first version of) the other 19 applications and for each test we have computed the performance indicators. This procedure has been repeated for all applications, leading to 20 models, each tested 19 times.

TABLE 2
Working set of applications.

	Application	Category	Downloads	Versions
F-Droid	AnkiDroid	education	100k - 500k	8
	BoardGameGeek	books	10k - 50k	8
	Connectbot	communication	1M - 5M	12
	CoolReader	books	1M - 5M	13
	Crosswords	brain & puzzle	5k - 10k	17
	FBReader	books	1M - 5M	14
	K9Mail	communication	1M - 5M	19
	KeePassAndroid	tools	100k - 500k	13
	MileageTracker	finance	100k - 500k	6
	Mustard	social	10k - 50k	12
OS	Browser, Calendar, Camera, Contacts, DeskClock, Dialer, Email, Gallery2, Mms, QuickSearchBox			6

4 Descriptive Statistics

Table 2 summarizes the 20 applications in our working set. The apps are organized in two groups and listed in alphabetical order. The first group contains the apps we downloaded from F-Droid. The second group refers to the apps that are included in the Android OS. The table also reports the type of the applications and their popularity in terms of number of downloads. Clearly, this information is not relevant for the OS apps. The working set contains a variety of application types ranging from utilities to games. All applications are quite popular, with five applications passing the threshold of one million downloads. The last column in the table reports the number of versions used in this study. In some cases, like K9Mail, the development has been very active and releases have been pushed out almost every month in the period of time we considered. Other applications, however, have slower release cycles and we had to settle for a smaller number of versions. In the case of the OS apps, the number of versions is dictated by the release cycle that Google decided for the operating system itself. The detailed list of versions used in this study is available online [34].

In this section, we describe the applications from the perspective of their number of lines and the positives ratio. Further information about other characteristics of the applications, including the number of files and the total number of vulnerable files, is available online [34].

Figure 1 characterizes the size of the applications (as lines of text) across the versions. In their initial version, which we call v_0, the applications range from 1,700 lines (MileageTracker) to 90,500 (Gallery2). All applications in the F-Droid group (left-hand side of Figure 1) increased in size throughout the various releases. Over time, Connectbot has grown only marginally (less than 10%), KeePassAndroid, Mustard and FBReader have grown moderately (between 20% and 50%), while the size of the remaining applications have increased substantially (between 50% and 200%). The steeper growth curve is represented by CoolReader, K9Mail and AnkiDroid. In particular, the latter grows almost exponentially. The

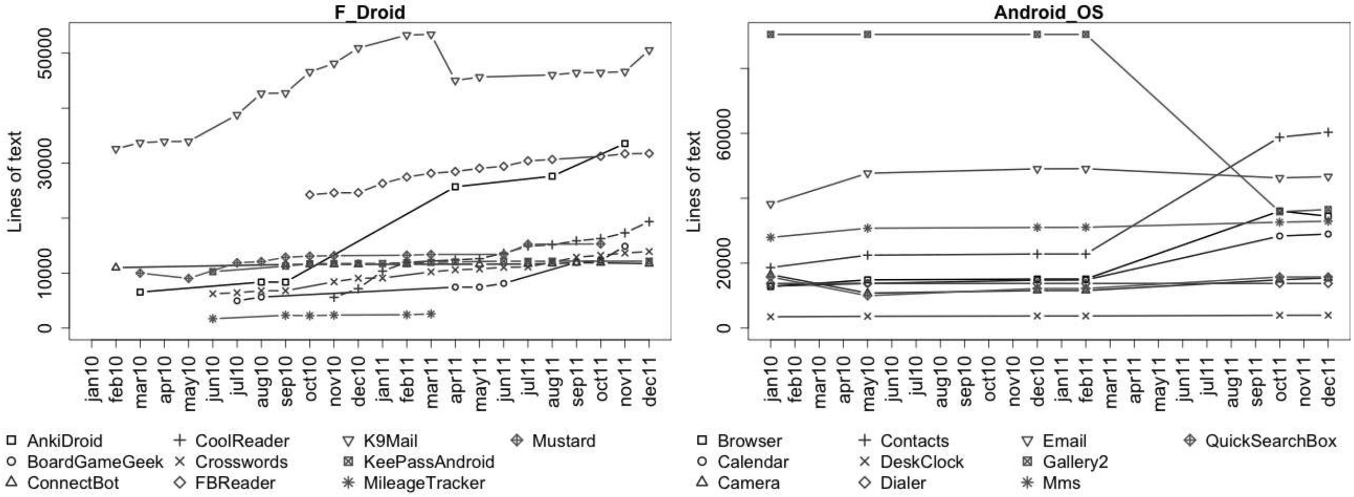


Fig. 1. Application size throughout the versions.

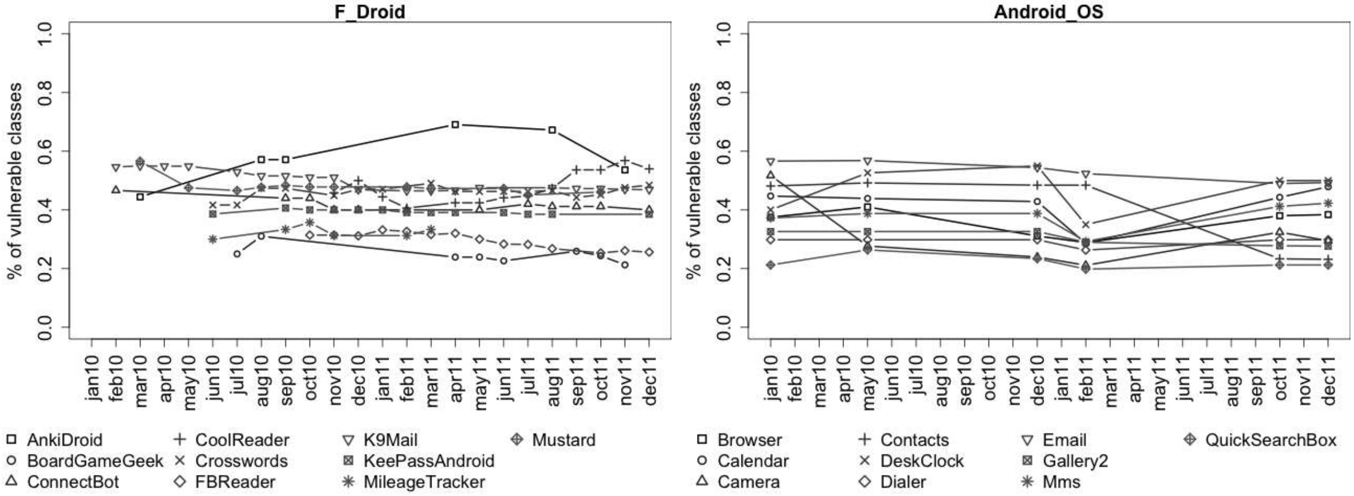


Fig. 2. Ratio of vulnerable files throughout the versions.

apps in the OS group (right-hand side of Figure 1) are rather stable in size, with the exceptions of Browser and Calendar, which grow, and Gallery2, which gets reduced significantly.

SCA reported a total of 116,286 vulnerabilities for all 182 versions. The types of vulnerabilities reported by SCA (and the counts) are provided online [34]. As some vulnerabilities remain unchanged from version to version of a particular application, not all of these reports are unique. In fact, there are only 22,776 unique vulnerabilities in the aggregate code base. As shown in Figure 2, most applications are barely improving over time, as their ratio of vulnerable files is approximately constant. Once more, AnkiDroid stands out for its erratic trend. CoolReader's ratio of vulnerable files worsens over time.

K9Mail, the largest application, had the largest number of vulnerabilities, ranging from 1,011 in version v_{12} to 2,922 in version v_8. No other application contained over 1,000 vulnerabilities. The application with the smallest number of vulnerabilities was MileageTracker, ranging from 7 in its initial version to 17 in later versions.

5 Experiment 1: Cross-validation

In the first experiment, we validated the use of text mining of source code to predict vulnerable software components. We built models with both Naïve Bayes and Random Forest machine learning techniques based on the first version (v_0) of each application. In order to avoid over-fitting effects, we resorted to stratified 10-fold cross-validation, which is a well-known technique and has been explained in Section 3.6.

5.1 Results

Table 3 summarizes the results for precision and recall (our key performance indicators) obtained in Experiment 1 for both Naïve Bayes and Random Forest. The

TABLE 3
Experiment 1: Average performance across 10 folds. In the majority of the apps at least one model passes the benchmark.

Application	Naïve Bayes			Random Forest		
	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$
AnkiDroid	0.92	0.92	0.07	0.85	0.92	0.13
BoardGameGeek	1.00	0.86	0.00	1.00	0.86	0.00
ConnectBot	1.00	0.81	0.00	1.00	0.86	0.00
CoolReader	1.00	0.91	0.00	1.00	0.91	0.00
Crosswords	0.87	0.87	0.10	0.87	0.87	0.10
FBReader	0.67	0.66	0.15	0.87	0.58	0.04
K9Mail	0.95	0.75	0.05	0.91	0.81	0.09
KeePassAndroid	0.85	0.77	0.09	0.88	0.80	0.07
MileageTracker	0.75	1.00	0.14	0.75	1.00	0.14
Mustard	0.97	0.86	0.03	0.93	0.86	0.09
Browser	0.92	0.92	0.05	0.92	0.92	0.05
Calendar	1.00	0.82	0.00	1.00	0.82	0.00
Camera	0.92	0.77	0.07	0.92	0.77	0.07
Contacts	0.91	0.77	0.07	0.91	0.77	0.07
DeskClock	0.75	0.75	0.17	0.71	0.63	0.17
Dialer	0.75	0.71	0.10	0.91	0.59	0.03
Email	0.94	0.72	0.06	0.93	0.81	0.08
Gallery2	0.69	0.62	0.13	0.88	0.55	0.04
Mms	0.86	0.70	0.07	0.94	0.59	0.02
QuickSearchBox	0.62	0.84	0.14	0.83	0.48	0.03

fall-out (i.e., false positive rate) is reported as well in the table. The table contains the average values obtained by each model across the 10 folds. A pair of cells colored in gray means that the corresponding model passed the performance benchmark stated in Section 3, i.e., the precision and the recall are above or equal to 80 percent.

We report that in 11 out of 20 applications at least one of the two models achieved excellent results, according to our benchmark. In eight cases, both models demonstrated excellent results. If we consider the models that achieved at least 75% of recall (which is often considered a 'good enough' value in the literature), we observe that Naïve Bayes reached that goal in fifteen applications and Random forest in fourteen.

> In summary, concerning RQ1, this exploratory experiment shows that the presented prediction technique can be used to build high quality prediction models for Android applications.

5.2 The Role of File Size

A skeptical reader might think that we are confusing causes and symptoms [38]. In fact, the size of the files could be the actual characteristic picked up by the learners, as a larger file will contain more terms with higher frequency and is also more likely to contain coding bugs that lead to security vulnerabilities. For instance, let us consider CoolReader, which is our best performer in Experiment 1 (Naïve Bayes). Indeed, in v_0, the average size of the vulnerable files is 2.8 times larger than clean files and the location shift is statistically significant (two-sided Wilcoxon test, p-value=0.0088).

The term-based features do retain information related to the size (and complexity) of the source code and are not independent from the above-mentioned characteristics. Nevertheless, the model evaluated in this study performs better than one based, for instance, on size metrics. To illustrate this point, we built a Naïve Bayes model based on file size of the type $vulnerable = f(lines\ of\ text)$, and found that the performance of this model is much lower than the performance values for the bag-of-words model shown in Table 3. Precision is 25 percentage points lower and recall goes down by 37 percentage points. Therefore, our technique is much more effective at predicting vulnerable components than file size alone.

TABLE 4
Experiment 1 was repeated on two applications with a manually validated vulnerability data set. The numbers are in line with the previous results.

App		Naïve Bayes		Random Forest	
		$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{R}$
AnkiDroid	Fortify SCA	0.92	0.92	0.85	0.92
	Validated	**1.00**	**1.00**	**1.00**	**1.00**
Mustard	Fortify SCA	0.97	0.86	0.93	0.86
	Validated	**1.00**	**0.86**	**1.00**	**0.79**

5.3 On the False Positives

Although in the related work section we highlighted how vulnerability databases are far from being perfect as sources of reliable vulnerability data, a skeptical reader might be still not entirely convinced by our use of static analysis to identify vulnerabilities. Therefore, we manually analyzed the Fortify SCA reports for two applications: AnkiDroid and Mustard. We analyzed the initial versions (v_0) of the above-mentioned applications. Both applications performed very well in the cross-validation experiment, with $\mathcal{F}_2$ values above 90%. Size-wise, AnkiDroid and Mustard are similar (see Figure 1). AnkiDroid, however, has a higher positive rate, as shown in Figure 2.

For these applications, each vulnerability warning generated by Fortify SCA has been manually checked by a security expert (i.e., the code has been inspected for the presence of a real vulnerability) and false positives have been removed. The manual checking took two working days. For AnkiDroid, 10 out of 12 vulnerability warnings were false positives. For Mustard, 14 out of 43 warnings have been flagged as false positives. Table 4 compares the prediction results we have obtained with the manually inspected data to the results presented previously in Section 5.

Removing the false positives and, hence, having more reliable data does not impact the essence of our results, at least in the two cases we have analyzed. Actually, the performance is generally better (in particular for Naïve Bayes) when the false positives are removed. From these results, we could hypothesize that the performance in

Table 3 is to be considered as a lower bound to what could be possibly achieved.

5.4 An Experiment with Documented Vulnerabilities

To further convince the above-mentioned skeptical reader, we applied the text mining technique to another application that does have a record of documented vulnerabilities. Therefore, this particular (cross-validation) experiment is not affected by false positives.

We selected an application that has similar characteristics (e.g., size and positive rate) compared to the Android apps we have analyzed so far. This way, the results shown in this sub-section are directly comparable to those shown earlier. We had to draw from a different application domain because, as we mentioned, there are no repositories of vulnerabilities for Android apps. We selected the Drupal content management system (drupal.org), which is written in PHP. According to the latest statistics [39], Drupal is one of the top 3 content management systems most used by the top 10 million web sites listed by Alexa (alexa.com). We considered version 6.0 of the software, which was released in February 2008. This version is more than five years old and, hence, the set of discovered vulnerabilities can be considered to be rather complete, i.e, there is no much influence due to false negatives.

We have analyzed all PHP files and extracted the term frequencies as described in Section 3.4. Clearly, the tokenization routine has been adapted to deal with the PHP syntax rules. We collected the vulnerabilities from the Drupal Security Advisories database (drupal.org/security). Each advisory is identified by a unique identifier and refers to the versions affected by the security bug. We retained a total of 17 advisories that impacted version 6 of Drupal and were part of the server side PHP code of the application. To identify the files that contained the vulnerabilities, we examined diffs of the source code between two versions of the application, one in which the vulnerability was present and one in which the vulnerability was fixed.

Drupal version 6.0 contains 157 PHP files in total and this size is comparable that of some larger apps analyzed previously, like FBReader, K9Mail or Mms. In total, there are about 3,100 terms, which is similar to the majority of the F-Droid apps (Android OS apps are slightly richer in this respect). The positive rate is 36%, which is similar to many apps analyzed previously, like KeePassAndroid, MilageTracker, and Gallery2.

We ran a 10-fold cross-validation experiment using both the Naïve Bayes technique and the Random Forest technique (configured with 100 random trees as previously). Especially in the case of Random Forest, we obtained good performance indicators. For Naïve Bayes, recall is 73 percent on average and precision is 55 percent. For Random Forest, recall is 82 percent. Hence, the most important indicator is above the threshold. Precision is 59 percent. Although precision is below par, we remark that the file inspection reduction ratio is 75 percent on average. This means that to achieve an 82 percent recall it is necessary to inspect 75 percent less files when using the prediction model compared to a random selection.

In summary, the text mining technique works convincingly in the case analyzed in this experiment, which is based on vulnerabilities discovered by both the security researchers worldwide and the security team of Drupal.

6 Experiment 2: Prediction in Future Releases

In the previous experiment, we demonstrated that our approach can deliver a dependable prediction model. In the second experiment, we investigated our approach further by attempting to predict vulnerabilities in future releases. For each application, we built a prediction model based on the initial version (using all source files available in v_0) and predicted all subsequent versions of that application (v_1 and following). This setup is similar to the approach taken, for instance, by Shin et al. [5] and Chowdhury et al. [12]. We were interested not only in determining if good forward predictions are possible, but also how far in the future a prediction model would work. We set the threshold for model performance deterioration at 10% of the $\mathcal{F}_2$ score. Thus, whenever the performance drops below the threshold we would ideally need to retrain the prediction model in order to have a dependable predictor again.

Figure 3 presents the performance obtained with the Random Forest model for both precision (left) and recall (right). In the figure, the 80% threshold is marked by a dashed line. To avoid clutter, the detailed results for the Naïve Bayes classifier are not depicted here. However, a summary of the results obtained with Naïve Bayes is shown in Table 5, as discussed later. Overall, the Random Forest technique achieves better performance results compared to Naïve Bayes. In particular, many more applications stay consistently above the threshold over time as far as recall is concerned. We remind the reader that recall is actually more important than precision in the field of security. Nevertheless, precision also appears to be better in the case of Random Forest, as the applications tend to stand higher above the threshold than in the Naïve Bayes case. Therefore, predictions in the future from the Random Forest model can be trusted a bit more (lower probability that a file reported as vulnerable is actually clean) and are more inclusive (lower probability that vulnerable files go unnoticed).

Concerning the Random Forest technique, we have not bounded the depth of the trees during the training phase. This is the default configuration in Weka. The average depth of the 1000 trees generated by the learning algorithm (100 trees per each application) is 7.5, with standard deviation of 5. The leaves are pure, i.e., all samples at one leaf belong to the same class. In the case of a Random Forest, pure leaves are not a sign of

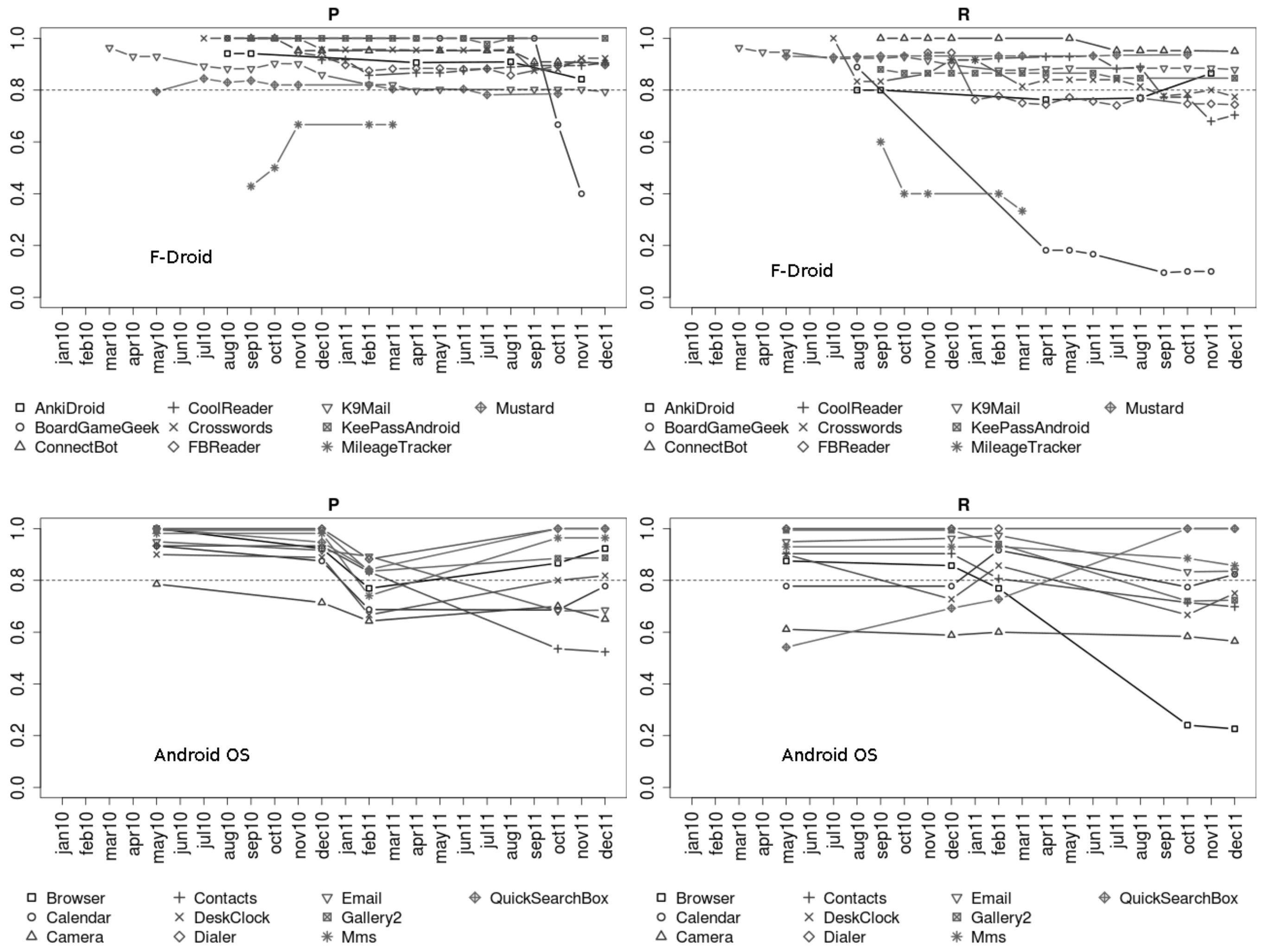


Fig. 3. Experiment 2: Precision and recall in future predictions with Random Forest. Performance is very satisfactory.

over-fitting, as this phenomenon is mitigated by having multiple trees and by bootstrapping the samples for each (random) tree.

For both Naïve Bayes and Random Forest, Table 5 summarizes the average performance indicators scored by each model (i.e., application) in the predictions over the future. A pair of cells colored in (dark) gray means that the corresponding model passed the performance benchmark stated in Section 3. The performance is very satisfactory for Random Forest. Not only the majority of the cases pass the benchmark, but also several other cases are very close to the threshold, like FBReader, Calendar, Contacts, DeskClock and QuickSearchBox (see light gray cells). Three applications, i.e., MileageTracker, BoardGameGeek and Browser, show very low recall. The low values for MileageTracker are most likely due to the extremely small size of the training set (only ten files). For BoardGameGeek, we noticed that the code on which the prediction model is built has been completely re-factored in v_2. This is why the precision and recall values are very high at v_1 but drop afterward. The case of Browser is less clear, although we suspect that a sudden rise in the vulnerabilities of version v_4 might have had an influence on the performance drop.

Intuition would say that the models perform well due to over-fitting, as a number of files that are used for testing were also present during training with the initial version. Reality, however, is different. First, the applications grow significantly in size as shown previously in Figure 1 (at least for the F-Droid group). Sometimes, the number of files doubles, which means that many files were unknown at training time. Second, files that were present (and clean) at training time become vulnerable in future versions, and vice versa. Therefore, the results obtained in this experiment are not obvious.

In general, the performance of both classifiers is rather stable over time, with the exception of the two applications mentioned earlier. However, performance declines for some applications over time. This can be observed in Table 6. For both techniques, the table reports the number of months that it takes for a model to lose 10% of its predictive performance, as measured by the $\mathcal{F}_2$

TABLE 5
Experiment 2: In the majority of the cases, the average performance in future prediction is above the benchmark with Random Forest. Instead, the performance of Naïve Bayes is inferior.

Application	Naïve Bayes			Random Forest		
	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$
AnkiDroid	0.85	0.82	0.23	0.91	0.80	0.13
BoardGameGeek	0.51	0.32	0.08	0.87	0.24	0.01
ConnectBot	0.99	0.92	0.01	0.95	0.98	0.04
CoolReader	1.00	0.77	0.00	0.89	0.85	0.10
Crosswords	0.90	0.76	0.07	0.95	0.84	0.04
FBReader	0.66	0.74	0.16	0.89	0.78	0.04
K9Mail	0.87	0.73	0.10	0.85	0.91	0.15
KeePassAndroid	0.85	0.69	0.08	1.00	0.86	0.00
MileageTracker	0.59	0.43	0.18	0.59	0.43	0.18
Mustard	0.83	0.86	0.15	0.81	0.93	0.19
Browser	0.88	0.57	0.04	0.90	0.59	0.03
Calendar	0.77	0.75	0.17	0.79	0.81	0.16
Camera	0.72	0.49	0.07	0.70	0.59	0.09
Contacts	0.76	0.70	0.11	0.75	0.81	0.13
DeskClock	0.79	0.68	0.16	0.81	0.78	0.16
Dialer	0.72	0.70	0.11	0.98	1.00	0.01
Email	0.83	0.71	0.16	0.83	0.91	0.21
Gallery2	0.70	0.64	0.12	0.92	0.87	0.03
Mms	0.80	0.73	0.11	0.93	0.91	0.04
QuickSearchBox	0.62	0.80	0.14	0.96	0.79	0.01

TABLE 6
Experiment 2: Need for re-training. In most cases, retrain is never needed over the two years period.

Application	Retrain after (months)	
	Naïve Bayes	Random Forest
AnkiDroid	21 *	21 *
BoardGameGeek	9	9
ConnectBot	22 *	22 *
CoolReader	10	10
Crosswords	2	2
FBReader	14 *	3
K9Mail	22 *	22 *
KeePassAndroid	18 *	18 *
MileageTracker	4	4
Mustard	21 *	21 *
Average	**14 months**	**13 months**
Browser	13	13
Calendar	23 *	23 *
Camera	21	23 *
Contacts	21	13
DeskClock	23	23
Dialer	23 *	23 *
Email	21	21
Gallery2	23 *	21
Mms	23 *	23 *
QuickSearchBox	23 *	23 *
Average	**21 months**	**21 months**

TABLE 7
Experiment 3: Applicability of each model across projects. Some models have a more general applicability.

Application	Model is applicable to (no. of other apps)	
	Naïve Bayes	Random Forest
AnkiDroid	1	4
BoardGameGeek	0	0
ConnectBot	0	0
CoolReader	0	1
Crosswords	0	0
FBReader	5	0
K9Mail	3	2
KeePassAndroid	0	0
MileageTracker	0	0
Mustard	4	1
Browser	0	0
Calendar	1	1
Camera	0	1
Contacts	1	0
DeskClock	0	0
Dialer	0	0
Email	0	0
Gallery2	2	0
Mms	0	0
QuickSearchBox	0	0

indicator. We remind that $\mathcal{F}_2$ weights recall more than precision. The choice of a 10% threshold is arbitrary but reasonable. Note that a star sign means that the $\mathcal{F}_2$ score never drops below the threshold until the last available version in the time window of two years we consider in this study.

In both techniques, in half of the cases the performance stays within tolerance in all future versions. In this respect, Naïve Bayes is slightly more stable, as retraining is not needed in 11 cases, i.e., one case more than Random Forest. On average, the retrain is not necessary for at least 13 months for the F-Droid applications and at least 21 months for the OS applications. This appears to be an encouraging result. Further investigation is necessary to understand whether this horizon is compatible with the expectations and needs of application development companies.

> In summary, concerning RQ2, this exploratory experiment shows that the presented prediction technique can forecast with excellent performance the vulnerable files of the future versions of an Android application. Furthermore, such a model is stable for at least 13 months on average.

7 Experiment 3: Cross-project

In the last experiment, we explored whether a model is applicable only to one application or rather can adequately predict vulnerable components across applications. This experiment follows the spirit of the work done by Zimmermann et al. [40] in the domain of defect prediction. However, to the best of our knowledge, this setup has not been used for vulnerability prediction yet.

In this experiment, we first built 20 models using version v_0 of each application. We then tested each model by predicting vulnerable files in the v_0 versions of the other 19 applications. For each test, we computed the performance as both precision and recall. We executed this experiment with both Naïve Bayes and Random Forest. In this experiment, the pure performance does not matter. Rather, we value the fact that some models apply beyond the scope of a single application.

For each application, Table 7 reports the numbers of other applications to which the corresponding models can be applied. A model is considered as applicable if precision and recall are simultaneously above the 80%, which is the same benchmark used previously. As shown in the table, there are thirteen models (from nine applications) that are generic enough in order to be successfully applicable to at least one other application. Furthermore, in the case of AnkiDroid, FBReader and Mustard, one model is applicable to several projects (four or more), which is a very encouraging result.

On the other hand, there are eleven applications whose models are totally incapable of predicting other projects. Among these, BoardGameGeek and MileageTracker are not surprising as they already exposed a poor performance in the "within project" prediction setup.

With reference to the Random Forest case, K9Mail and Mustard have interesting properties in terms of general applicability. Indeed, they have a recall above 80% in 8 (K9Mail) and 12 (Mustard) other projects, which yields also an high average recall. We remind that a high recall is very desirable in case of security, as a high recall means that fewer vulnerabilities are overlooked. K9Mail and Mustard are also the best all-around performers with $\mathcal{F}_2$ scores of 78% (in the Random Forest case).

We remark that K9Mail and Mustard both have a quite high percentage of vulnerable files. Instead, they are different in terms of both number of lines of text and number of text features. Further investigation is necessary to understand whether the percentage of positives is influential in creating a good cross-project prediction model, as we suspect that the nature of the text features might be of importance too.

Concerning RQ3, this exploratory experiment illustrates that some models built on a single application can predict which software components are vulnerable in other applications. However, we do not have a technique to identify which applications have data that can be used to produce vulnerability prediction models with the above-mentioned property.

In a more advanced setup, we plan to investigate a prediction model that is built on the merged code from two or more applications. Combining the files from multiple, possibly diverse applications might enlarge the set of features used by the learner and lead to improved results.

8 Threats to Validity

In this section, we discuss construct, internal, and external validity. We could not find any threat to conclusion validity, which is therefore not discussed here.

Construct validity. We used the Fortify SCA tool to identify vulnerabilities via static source code analysis rather than using the vulnerabilities reported in a database such as the NVD. This choice was obligatory, as there are no public databases with sufficient numbers of vulnerabilities to analyze for Android applications. As shown in the related work section, the state of the art has illustrated that static analysis vulnerability warnings are a good proxy of actual vulnerabilities. However, static analysis tools have a tendency to produce many false positives [19]. Therefore, we manually inspected and validated the vulnerability warnings for the initial versions of two of the applications that we studied. Although this is a limited subset of the data we used in this study, this effort was useful to demonstrate the applicability of our approach in the case of more validated vulnerability data. Nevertheless, we acknowledge that a more extensive validation should be done, possibly with the contribution of the research community. We have published the data used in this study in order to incentivize this endeavor.

Internal validity. In this study, we considered only the Java source files of each Android application. We excluded the XML manifest file that is packaged with each Android application. The manifest contains important information, such as the permissions that need to be granted to the application. Another threat refers to vulnerability warnings that affect several versions and, in particular, both the training set and the testing sets in the case of prediction in the future. These vulnerabilities might have inflated the results.

External validity. Our study is exploratory and a larger scale validation is necessary. Our results might be specific to the 20 applications we have selected. In an attempt to counter this threat, we evaluated our approach using a range of applications that vary in terms of size, revision history, and popularity. However, further studies using a different set of experimental materials are necessary in order to generalize the results to the entire class of Android applications. For instance, this study did not focus on apps that are smaller than 1 KLOC. The same reservation about generalizing this study applies to other types of applications, such as web applications, or to applications written in programming languages other than Java.

9 Conclusions and Future Work

In this paper, we presented an approach to predict whether a software component is vulnerable based on text mining of its source code. The approach has never been used before in the context of vulnerability prediction. The approach presented here analyzes the source code directly instead of indirectly via software or developer metrics. The results of our exploratory study demonstrate that the approach has good performance for both precision and recall when it is used for within-project prediction. In general, we obtained a prediction power that is equal or even superior to what is achieved by state of the art vulnerability prediction models.

In the future, we plan to further investigate the presented approach by exploring three directions. First, we are interested in generalizing the results to different classes of applications and languages. Second, although we used Java files as software components, the technique may be applied successfully on the level of classes and methods too. Third, we believe that our approach is complementary to the existing techniques that use software metrics for prediction.

Acknowledgments

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

References

[1] Microsoft, "The Microsoft Security Development Lifecycle (SDL): SDL builds more secure software," http://www.microsoft.com/security/sdl/about/benefits.aspx, [Accessed 10 June 2013].

[2] B. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[3] D. Wallace and R. Fuji, "Software verification and validation: Its role in computer assurance and its relationship with software product management standards," NIST Special Publication 500-165, Tech. Rep., 1989.

[4] S. Slaughter, D. Harter, and M. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.

[5] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[6] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[7] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, 2010.

[8] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[9] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–29, 2013.

[10] ——, "Is complexity really the enemy of software security?" in *ACM Workshop on Quality of Protection (QoP)*, 2008.

[11] ——, "An empirical model to predict security vulnerabilities using code complexity metrics," in *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2008.

[12] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[13] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[14] M. Gegick, P. Rotella, and L. Williams, "Toward non-security failures as a predictor of security faults and failures," in *Symposium on Engineering Secure Software and Systems (ESSoS)*, 2009.

[15] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *International Workshop on Security Measurements and Metrics (MetriSec)*, 2010.

[16] B. Smith and L. Williams, "Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[17] F. Massacci and V. H. Nguyen, "Which is the right source for vulnerability studies?: an empirical analysis on mozilla firefox," in *International Workshop on Security Measurements and Metrics (MetriSec)*, 2010.

[18] B. Martin and S. Christey, "Buying into the bias: why vulnerability statistics suck," BlackHat USA, 2013.

[19] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011.

[20] J. Walden and M. Doyle, "SAVI: Static-analysis vulnerability indicator," *IEEE Security & Privacy*, vol. 10, no. 3, pp. 32–39, 2012.

[21] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *ACM Workshop on Quality of Protection (QoP)*, 2008.

[22] M. Gegick, P. Rotella, and L. Williams, "Predicting attack-prone components," in *International Conference on Software Testing Verification and Validation (ICST)*, 2009.

[23] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *International Conference on Software Engineering (ICSE)*, 2005.

[24] N. Edwards and L. Chen, "An historical examination of open source releases and their vulnerabilities," in *ACM conference on Computer and Communications security (CCS)*, 2012.

[25] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[26] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno, "Spam filter based approach for finding fault-prone software modules," in *International Workshop on Mining Software Repositories (MSR)*, 2007.

[27] L. Aversano, L. Cerulo, and C. Del Grosso, "Learning from bug-introducing changes to prevent fault prone code," in *International Workshop on Principles of software Evolution (IWPSE)*, 2007.

[28] S. Kim, J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[29] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[30] A. Bosu and J. Carver, "Peer code review to prevent security vulnerabilities: An empirical evaluation (extended abstract)," in *International Conference on Software Security and Reliability (SERE)*, 2013.

[31] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2011.

[32] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2010.

[33] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2010.

[34] R. Scandariato, J. Walden, and A. Hovsepyan, "Experimental material," https://sites.google.com/site/textminingandroid/.

[35] D. Wolpert, "The lack of a priori distinctions between learning algorithms," *Neural Computation*, vol. 8, no. 7, pp. 1341–1390, 1996.

[36] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM CACM*, vol. 55, no. 10, pp. 78–87, 2012.

[37] I. Kononenko, "On biases in estimating multi-valued attributes," in *International Joint Conference on Articial Intelligence (IJCAI)*, 1995.

[38] A. Zeller, T. Zimmermann, and C. Bird, "Failure is a four-letter word: A parody in empirical research," in *International Conference on Predictive Models in Software Engineering (PROMISE)*, 2011.

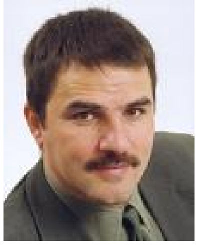
[39] W3Techs, "Usage of content management systems for websites," http://w3techs.com/technologies/overview/content_management/all, [Accessed 25 October 2013].

[40] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Symposium on the Foundations of Software Engineering (FSE)*, 2009.

Riccardo Scandariato has a Ph.D. in Computer Science from Politecnico di Torino (Italy). He leads a team of researchers in the area of secure software engineering at the KU Leuven (Belgium). His main research interests are in the area of software architectures and machine learning for software security. He has published over 50 papers in the field of security and software engineering. He is also an Associate Editor of the International Journal of Secure Software Engineering (IJSSE).

James Walden is an Associate Professor at NKU (USA). He has a Ph.D. from Carnegie Mellon, and he worked as a senior software engineer at Intel. His research focuses on applying empirical software engineering techniques to software security problems. He worked with SANS to develop their GIAC Secure Software Programmer (GSSP) certification and worked with MITRE and SANS to develop their Top 25 Most Dangerous Software Errors listings.

Aram Hovsepyan received a Ph.D. in Engineering from KU Leuven (Belgium). He is a postdoc researcher at the Department of Computer Science of KU Leuven. His main research interests lie in the area of model-driven software development and empirical software engineering with a particular focus on security vulnerability prediction techniques.



Wouter Joosen is full professor at the Department of Computer Science of KU Leuven (Belgium), where he teaches courses on software architecture, distributed systems and the engineering of secure service platforms. His research interests are in aspect-oriented software development, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.