Text Filtering and Ranking for Security Bug Report Prediction

Fayola Peters, Thein T. Tun, *Member, IEEE,* Yijun Yu, *Member, IEEE,* and Bashar Nuseibeh, *Member, IEEE*

Abstract—Security bug reports can describe security critical vulnerabilities in software. Bug tracking systems may contain thousands of bug reports, where relatively few of them are security related. Therefore finding unlabelled security bugs among them can be challenging. To help security engineers identify these reports quickly and accurately, text-based prediction models have been proposed. These can often mislabel security bug reports due to a number of reasons such as class imbalance, where the ratio of non-security to security bug reports is very high. More critically, we have observed that the presence of security related keywords in both security and non-security bug reports can lead to the mislabelling of security bug reports. This paper proposes FARSEC, a framework for filtering and ranking bug reports for reducing the presence of security related keywords. Before building prediction models, our framework identifies and removes non-security bug reports with security related keywords. We demonstrate that FARSEC improves the performance of text-based prediction models for security bug reports in 90% of the cases. Specifically, we evaluate it with a total of 45,940 bug reports from Chromium and four Apache projects. We found that with our framework, we reduced the number mislabelled security bug reports by 38.1%.

Index Terms—security cross words, security related keywords, security bug reports, text filtering, ranking, prediction models, transfer learning.

1 Introduction

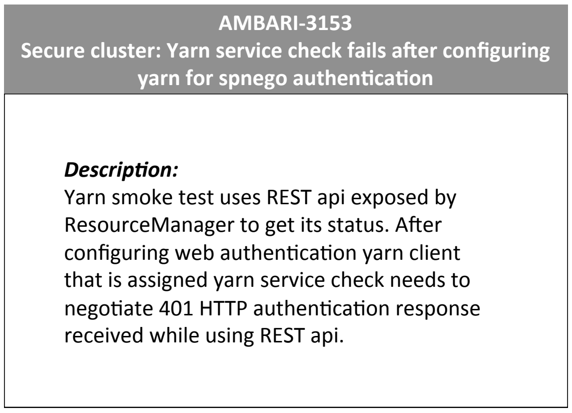
Bug tracking systems help developers maintain software products by allowing reporters to submit any bugs encountered while using a software product. Some bug reports can describe security vulnerabilities which could be exploited by attackers if they are exposed before they are fixed. A security bug is a security vulnerability that allows a user to have inappropriate access to the system and thus cause harm or damage to the software or to persons using software [1]. Vendors usually request that bug reporters do not disclose any suspected security vulnerabilities in public bug tracking systems. Instead, they suggest that suspected security vulnerabilities be reported directly and privately to security engineers who assesses them and, when necessary, provide patches to customers before an attacker discovers and exploits the vulnerability. Once a patch has been disseminated, vulnerabilities are often documented and disclosed via the bug tracking system [2].

In reality, because of the lack of security domain knowledge on the part of some bug reporters [3], or others who simply ignore the request from vendors and security engineers, suspected security bug reports are often publicly disclosed before they are assessed and fixed [4]. To help security engineers identify security bug reports quickly and accurately, text-based prediction models have been proposed [1], [3], [4], and implemented in industry [3]. These prediction models are a combination of labelled bug reports and machine learning algorithms. However, there is one underlying issue not fully explored in these models, which we call *security cross words*. Security cross words denote *the use of the same security related keywords in both security* ***and*** *non-security bug reports*. We conjectured that text-based prediction models can mislabel security bug reports when security cross words are present. The problem is magnified if there is class imbalance where non-security bug reports outnumber security bug reports by a large margin. For instance, among the 45,940 bug reports studied in this paper, only 0.8% are known to be security bug reports.

We propose *FARSEC, a framework composed of a combination of* ***F****iltering* ***A****nd* ***R****anking methods to reduce the mislabelling of* ***SEC****urity bug reports by text-based prediction models.* When building prediction models, FARSEC automatically identifies and removes non-security bug reports containing security cross words. It begins by finding security related keywords from the security bug reports of a project (Section 3.1). Each security related keyword is *scored* according to its frequency in both security and non-security reports. Using the keyword scores of bug reports, we remove non-security reports with scores as high as those of security bug reports (Section 3.2). The remaining bug reports are used to build prediction models (Section 3.3). Finally, FARSEC uses the results of the prediction models to present security engineers with ranked lists of bug reports where most of the actual security bug reports are closer to the top of the ranked lists than at the bottom.

Consider the example in Figure 1, where a bug report, `Ambari-3153` was mislabelled by researchers as non-security as of May 5th 2017 when the data was downloaded (Section 4.1). However, according to its security label in the JIRA bug tracking system, `Ambari-3153` describes a secu-

• *F. Peters and B. Nuseibeh are with Lero - The Irish Software Research Centre, University of Limerick, Limerick, Ireland.*
E-mail: {fayola.peters, bashar.nuseibeh}@lero.ie
• *T. Tun, Y. Yu and B. Nuseibeh are with the Department of Computing and Communications, The Open University, Milton Keynes, United Kingdom.*
E-mail: {thein.tun, yijun.yu, bashar.nuseibeh}@open.ac.uk


AMBARI-3153
Secure cluster: Yarn service check fails after configuring yarn for spnego authentication

Description:
Yarn smoke test uses REST api exposed by ResourceManager to get its status. After configuring web authentication yarn client that is assigned yarn service check needs to negotiate 401 HTTP authentication response received while using REST api.

Fig. 1. `Ambari-3153` is a bug report mislabelled as non-security.

rity vulnerability[1]. If `Ambari-3153` and others like it, i.e. SBRs labelled as NSBRs are used to build prediction models, any potential SBRs are more likely to be mislabelled by the models. Therefore, FARSEC would not use `Ambari-3153` to build prediction models for SBRs.

We evaluate FARSEC with a total of 41,940 bug reports from Chromium and four Apache projects (Section 4.1) each containing 1000 reports. We also use five machine learning algorithms (Section 4.3) and six performance measures (Section 4.4). We then consider the following research questions:

RQ1: Can security cross words lead to mislabelled security bug reports by prediction models?
We look at the number of security cross words present before and after using FARSEC. We then consider these numbers in conjunction with the prediction of SBRs using prediction models. The results indicate that security cross words can contribute to mislabelled SBRs (Section 5.1).

RQ2: How do we build *effective* prediction models for security bug reports when data scarcity is an issue?
To build effective prediction models we need *sufficient* data from both security and non-security bug reports. When data scarcity is an issue, one solution is to use data from other projects to build prediction models. Results show better performance in favour of using FARSEC and data from other projects (Section 5.2).

RQ3: How do we generate *useful* lists of ranked bug reports?
FARSEC produces ranked lists of bug reports. We view the *usefulness* of the ranked lists as having more actual SBRs at the top of the lists than at the bottom. Our results show that for all the projects, FARSEC lists are more useful than non-FARSEC lists (Section 5.3).

The paper presents FARSEC, a framework designed to reduce the mislabelling of security bug reports by text-based prediction models. Thus we make the following contributions:

1) An approach to automatically identify security related keywords and security cross words from security bug reports.
2) An automatic filtering and ranking method to build better text-based prediction models for security bug reports by removing NSBRs with security cross words from the prediction model. Better prediction models reduce the mislabelling of security bug reports.
3) A tractable method to use both bug reports from within a single project and bug reports from other projects to build text-based prediction models for security bug reports.
4) A ranking capability used to generate a useful ranked list of bug reports where most of the actual security bug reports are closer to the top of the list.

1. https://issues.apache.org/jira/browse/AMBARI-3153

2 Background and Related Work

Existing research has proposed prediction models to detect security vulnerabilities in both the pre- and post-release phases of software development. In the pre-release phase the source code is used to build prediction models [5], [6], [7], [8], [9], [10] while bug reports are used in the post-release phase [1], [3], [4]. Although the data used in these phases are different, the process for building the prediction models face the same issues of class imbalance, cross words, and insufficient data.

2.1 Security Vulnerability Models with Source Code

Source code can be used to derive metric-based and text-based prediction models. Metric-based models are fashioned after defect prediction models. Similarly, they use code metrics such as code churn, complexity, coupling and cohesion metrics, and code coverage [5], [6], [7], [9], [10]. Zimmermann et al. [5] and Shin et al. [6], [7] investigated the feasibility of these metrics (*classical metrics*) for security vulnerability prediction. The former performed their study on Windows Vista and found that the metrics had a low correlation with vulnerabilities and could predict them with good precision but with very low recall. In addition they found that the use of dependencies for prediction worked better than the code metrics in terms of better recall.

Studies by Shin et al. were done on open source projects, namely the Mozilla Firefox web browser and the Red Hat Enterprise Linux kernel. Out of 28 metrics, 24 were discriminative for security vulnerabilities in both projects [6]. Later they found that their vulnerability prediction model from Mozilla had a recall of 83% and precision of 12% at a classification threshold 0.5 [7].

Text-based models use terms and their frequencies in source code to build prediction models for security vulnerabilities. In their study of 20 android applications, Scandariato et al. [8] used text-based models to predict vulnerabilities in software components. Walden et al. [9] compared metric-based models with text-based models for source code and found that the text-based models performed best with higher recall in three web applications.

Class imbalance affects the performance of machine learning algorithms in the presence of under-represented data and severe class distribution skews [11]. In a study by Morrison et al. [10], have noted that while Microsoft product teams have adopted defect prediction models, they have not adopted vulnerability prediction models. This was due to their poor performance in terms of precision and recall with highly imbalanced data.

In this paper we focus on security vulnerability predictors built with bug reports (described in the following section). We face the same issue of class imbalance where out of the 45,940 bug reports studied, only 0.8% are security bug reports.

2.2 Security Vulnerability Models with Bug Reports

Building security vulnerability models with the natural language text of bug reports is a text-mining task. First, relevant keywords are identified with semi-automatic methods and then prediction models are built using these as a feature set along with their frequencies in each document. One of the earlier works on this topic is by Gegick et al. [3] who highlighted the problem of SBRs being mislabelled as NSBRs because of (say) the lack of security domain knowledge of bug reporters. Their research resulted in an approach that leveraged the natural language descriptions of bug reports to train a statistical model in an effort to identify SBRs that were manually mislabelled as NSBRs. Security engineers could use their model to automate the classification of bugs. Their evaluation was based on a proprietary software system from Cisco. In this paper our evaluations are done with open source data (Table 1), allowing our experiments to be reproducible.

Later research by Wijayasekara et al. [4] focused on the problem of hidden impact vulnerabilities where security bug reports are only identified after being made public. In an analysis they revealed that among the discovered vulnerabilities in the Linux kernel and MySQL, 32% and 62% respectively were hidden impact vulnerabilities. They also showed that these values increased in the following years. The authors proposed a vulnerability discovery methodology where they distinguished between long and short bug reports prior to applying text-mining methods. A base-rate fallacy evaluation was used to acknowledge the issue of class imbalance. This measure allows security engineers to choose prediction models based on the false positive alarm rates they would be willing to accept. For example, if a prediction model has a precision of 0.01, it means that out of 100 predicted SBRs, one will be an actual SBR.

2.2.1 Methods for finding Security Related Keywords

For many researchers, identifying security related keywords usually starts with a seed list which is expanded semi-automatically using external sources. For instance Gegick et al. [3] mentioned in their configuration file preparation phase that they obtained terms from bug reports to populate start, stop and synonym lists. The authors described manually adding terms such as *vulnerability* and *attack* from security bug reports to start lists. Also included were terms, which were not explicitly security related, such as *crash* and *excessive*. In a similar vein, Pletea et al. [12] used an iterative process to construct a set of security relevant keywords, which they called a *keyword-based* approach. They started with a seed list derived from the existing literature and their own security expertise. Their seed list contained keywords such as *security*, *ssl*, *encryption* and *authentication*, which is expanded using co-occurring tags from Stack Overflow.

In the analysis of software maintenance techniques, Hindle et al. [13] created three keyword lists using external sources independent of the data used in their study. Using an ontology for software quality measurement and the ISO9126 taxonomy [2], they associated keyword lists with six labels from the standard for non-functional requirement (namely, maintainability, functionality, portability, efficiency, usability, and reliability). One of the labels is functionality, to which security is associated. Hindle et al. [13] then expanded the keyword lists using WordNet and a random analysis of mailing list messages from an open source project, where any word considered to be associated with a non-functional requirement is added to the keyword list. In this paper we automatically identify security related keywords using the text in SBRs and tf-idf (Section 3.1), and without using external sources. Nevertheless, many of our keywords are similar to those found in other studies [3], [12], [13].

2.2.2 Transfer Learning and Prediction Models

One fundamental assumption of traditional machine learning algorithms is that the training (past) data and present (test) data have the same features and distribution. This assumption does not hold, for example, when the training data and present data come from different projects. *Transfer learning* aims to bridge this gap by extracting useful knowledge from past data and applying it to different present data. This is especially useful when the target data is new and has not been examined by domain experts [14].

Following defect prediction models, some vulnerability prediction approaches use code metrics. One line of research on defect prediction is cross project defect prediction (CPDP), which uses data from external sources to build models [15]. CPDP is useful because for many companies that are relatively small or have new products, local data may not be readily available. With the use of better selection tools for training data and transfer learning techniques, researchers found it possible to predict defects for "data starved" software projects by using data from external sources [16], [17], [18], [19], [20], [21], [22], [23], [24].

Results from existing studies of SBRs are currently inconclusive [3], [8]. The ability to use data from other sources to predict security bugs has been studied with mixed results. For example, Gegick et al. [3] recommended that the trained model should not be applied to software systems in which the SBRs describe different types of security bugs than those that were used to train the model [3]. On the other hand, Scandariato et al. [8] found that some models built on a single application can predict which software components are vulnerable in other applications. However, they admit that they do not have a technique to identify which applications have data that can be used to produce the general vulnerability prediction models.

Chawla et al. [1], used tf-idf (a statistic for weighting keywords described in Section 3.1) and the semantic similarity of the keywords found with tf-idf in order to generate their keyword lists. Their Multinomial Naive Bayes predictor was able to label bug reports as security, regression, polish

2. ISO/IEC 25010:2011 has recently added "security" as one of the main characteristics of software qualities.

and cleanup. Similarly, they used Chromium bug reports in their evaluation and hinted at (security) cross words when they described manually removing keywords from the four different categories of Chromium, which appeared in the top 50 keywords of each category. Their intuition was that these keywords did not seem to provide a discriminative contribution. In this paper we automatically identify security cross words, but we do not remove all of them from our feature set because at least 74% of the security related keywords identified in our study are cross words. Instead we use them to remove NSBRs from the data sets in order to improve the prediction of SBRs.

To conclude, this paper proposes a novel method for constructing text-based prediction models for security bug reports, focusing on the issue of security cross words. Our results show that appropriate dealing of security cross words provides a foundation for dealing with other issues such as insufficient data, feature selection and class imbalance.

3 FARSEC Design and Operation

With FARSEC, our goal is to improve text-based prediction models for SBRs. When building prediction models, FARSEC automatically identifies and removes NSBRs containing security cross words. Figure 2 shows the three main stages of FARSEC: 1) identifying *security related keywords* (Section 3.1); 2) filtering via the scoring of security related keywords and bug reports (Section 3.2); and 3) ranking based successive sorting and prediction (Section 3.3). The result is a sorted list of predicted bug reports, where most SBRs are expected to appear above NSBRs.

3.1 Identifying Security Related Keywords

To identify security related keywords and security cross words from bug reports, we first tokenize the SBRs and then calculate the tf-idf values of each term (explained in Step 4 below). We consider the hundred terms with the highest tf-idf values to be security related keywords. Of these 100 terms, we consider those found in NSBRs to be security cross words. The security related keywords are then used to build term-document matrices using the following steps (*keywords* and *terms* are used interchangeably):

1) Tokenize text: This is a frequently used method in text mining, which involves splitting text into sentences and words [25]. In addition, it is common to extract *token features* that are usually categorical functions of tokens such as types of capitalization, punctuation and special characters [25]. As part of tokenizing text in this paper, we make all terms lowercase.
2) Remove *stop words*: Stop words are common words that are irrelevant to the classification task [25] and are therefore removed. Some examples of English stop words include: `a, again, on, the, their, will`. In this paper we use the English stop words list included with the Natural Language Toolkit [3].
3) Remove *unwanted* terms: In this paper we go beyond stop words, and also remove unwanted terms. For bug reports used in this paper, we describe unwanted terms as those that contain punctuation and other non-alphanumeric characters. We remove unwanted terms based on the assumption that they may appear infrequently and only in a small percentage of bug reports when transfer learning is considered. Therefore, as features, the unwanted terms would lead to sparse data matrices. As a result of removing these unwanted terms, it is possible that interesting ones will be lost, such as `attacker's`, due to apostrophe. Examples of unwanted terms which appear in the projects studied in the paper are online [4].
4) Calculate term frequency-inverse document frequency (tf-idf): Tf-idf [26] is a statistic used to weight the importance of terms to a document in a corpus. We use the following equations from Manning et al. [27], [28] to calculate tf-idf.

$$tf(t,d) = 0.5 + \frac{0.5 \times f(t,d)}{max\{f(w,d) : w \in d\}} \tag{1}$$

$$idf(t,D) = log\frac{N}{|\{d \in D : t \in d\}|} \tag{2}$$

$$tf\text{-}idf(t,d,D) = tf(t,d) \times idf(t,D) \tag{3}$$

In Equations 1-3, N is the total number of documents, t represents a term, d represents a document in the set of documents D. In Equation 1, the term frequency $tf(t,d)$ represents how often a word appears in a document, normalized by the maximum frequency of each keyword in a document. Inverse document frequency $idf(t,D)$, is the log of the number of documents in which the word appears (Equation 2). In this work we choose the top 100 terms in SBRs with the highest $tf\text{-}idf$ values as our feature set. We restrict the feature set to 100 based on work by Bozorgi et al. [2], which found that the top 100 features spanned nearly all of the *feature families* in their study. The feature families represented the 28 parts of the vulnerability reports, for example the *title* and *description*.
5) Construct term-document frequency matrix: This involves creating data matrices from the 100 security related keywords where each row represents a document in a corpus and the columns represent the terms in the feature set.

3.2 Filtering Bug Reports

FARSEC filtering is about removing NSBRs with security related keywords from the term-document matrix. Filtering is based on the scoring of the terms in the feature set. Using these scores we calculate an overall score for bug reports, which contain all, some, or none of the terms in the feature set.

This is similar to text filtering approaches used to classify emails into spam and non-spam [29], [30]. FARSEC filtering method is based on Graham's Bayesian filter [30].

3. http://preview.tinyurl.com/yanmtk34

4. https://tinyurl.com/yccn6hq7

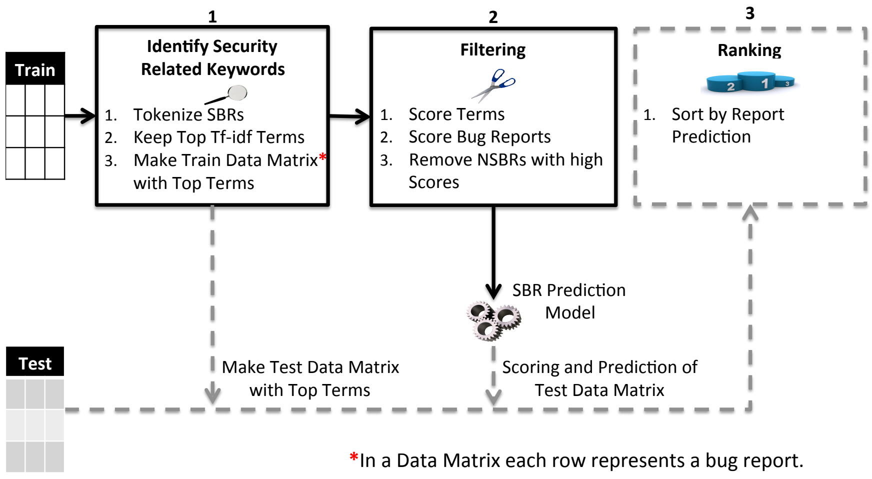

1
Identify Security Related Keywords
1. Tokenize SBRs
2. Keep Top Tf-idf Terms
3. Make Train Data Matrix* with Top Terms
2
Filtering
1. Score Terms
2. Score Bug Reports
3. Remove NSBRs with high Scores
3
Ranking
1. Sort by Report Prediction
Train
Test
SBR Prediction Model
Make Test Data Matrix with Top Terms
Scoring and Prediction of Test Data Matrix
*In a Data Matrix each row represents a bug report.

Fig. 2. Overview of the FARSEC approach.

Algorithm 1 shows how we score each keyword in our feature set. The result is a dictionary of keywords mapped to their scores (Line 8). The function **ScoreWords** accepts as input a dataset with a header row of security related keywords (Section 3.1) from a feature set and a *support* function. We partition the data into SBRs, NSBRs, and the keywords (Line 2). For each keyword, the algorithm calculates the probability of the keyword appearing in SBRs and NSBRs. We then calculate the score of the keyword using the equations in Line 3 to Line 6.

Algorithm 1 Score keywords.

```
1: ScoreWords(D, support) {D is the data with a feature set, and support adds bias in favour of SBRs.}
2: Partition(D) ↦ {$S$, $NS$, $W$} {$S$ is SBR data, $NS$ is NSBR data and $W$ is the feature set.}
3: for w in W do
4:     P($S_w$) ← Min (1, $\frac{\textbf{support}(tf(S_w))}{|S|}$)
5:     P($NS_w$) ← Min(1, $\frac{tf(NS_w)}{|NS|}$)
6:     Score($w$) ← Vector(w, Max(0.01, Min(0.99 , $\frac{P(S_w)}{P(S_w)+P(NS_w)}$)))
7: end for
8: return Hashmap(Score(w)) {Returns dictionary of $w$ mapped to Score($w$).}
```

In order to reduce false positives (mislabelled non-spam emails), Graham [30], biased the probabilities by trial and error and found that multiplying the frequency of non-spam emails by two was a good way to achieve a good bias. Similarly, work by Jalali et al. [31] found that the equation in Line 6 was a poor ranking heuristic for low frequency evidence. To alleviate this problem, their support function involved squaring the numerator of the equation (Algorithm 1, Line 6). In order to reduce mislabelled SBRs, we apply support functions to the frequency of the words found in SBRs (Algorithm 1, Line 4). This allows us to investigate three versions of filtering for FARSEC. They are denoted in our experiments as: 1) ***farsecsq***, applying the Jalali et al. [31] support function to the frequency of words found in SBRs; 2) ***farsectwo***, the Graham [30] version of multiplying the frequency by two and; 3) ***farsec***, which offers no support. They are collectively referred to as FARSEC filters.

In addition, following Graham [30], in cases where words appearing in SBRs do not appear in NSBRs, probability 0.99 is assigned as their scores. Conversely, when words appearing in NSBRs do not appear in SBRs, probability 0.01 is assigned.

Algorithm 2 shows how we calculate the overall score for a bug report. The **ScoreReport** function accepts three inputs, the bug report to be scored, a dataset, and support function used by **ScoreWords** to create a dictionary of scored keywords. For each term in the bug report, we get its score from the dictionary. If the term is not present in the dictionary, the returned score is zero. All the scores for the bug report are added to M and their complement scores are added to M' (Lines 5-11). We find the product of the scores of each set and calculate the overall score of the bug report (Line 12).

NSBRs are selected using the threshold score of ≥ 0.75 because those with higher scores are likely to be false positives. The threshold value is based on our experience with the datasets that reports with scores in the mid-range (between 0.4 and 0.6) are less likely to be SBRs. The reason for the effectiveness of this threshold value is not yet clear and is an issue for future work.

Algorithm 2 Score bug report.

```
1: ScoreReport(R, D, support) {R is a bug report and D is the data with a feature set, and support adds bias in favour of SBRs.}
2: M ← ScoreWords(D, support)
3: M* ← ∅ {Initialized list of scores for security related keywords in R.}
4: M′ ← ∅ {Initialized list of complement scores for security related keywords in R.}
5: for w in R do
6:     P(w) ← GetScore(w, M) {Returns score for each keyword (w) if present in dictionary and a score of zero if not present.}
7:     M* ← P(w)
8: end for
9: for m in M* do
10:    M′ ← 1−m
11: end for
12: return $\frac{\prod_{i=1}^{|M^*|} m_i}{\prod_{i=1}^{|M^*|} m_i + \prod_{i=1}^{|M'|} (1-m_i)}$
```

3.3 Ranking Bug Reports

When dealing with imbalanced data, the results of prediction models can yield a large number of false positives i.e. NSBRs predicted as SBRs. Therefore, after identifying predicted SBRs, we generate a useful list of ranked bug reports so that the majority of actual SBRs are closer to the top of the list.

We rank bug reports based of the idea of *ensemble learning* where the results of multiple machine learning models are combined for better predictions [32], [33]. The idea is that with ensembles even the *weaker* models are useful. There are three main ways to build ensembles 1) boosting, 2) bagging and 3) blending. Boosting uses successive learning models, which focus on the misclassified data of the previous model while bagging creates subsets of the data (with replacement) to create different models [33]. The results are then combined using a voting strategy. Blending is a stacking method [32], which combines predictions from multiple models using another (called a meta-learner).

FARSEC ranking is similar to blending, but instead of applying a meta-learner it ranks them by successively sorting the prediction results with one of the FARSEC filters or non-filtered training data. Notice that when prediction results are equal for any two bug reports, they are ordered in the chronological order in which they were entered into the bug tracking system.

For example, if we are ranking the test bug reports according to the prediction results of the ***farsecsq*** filter, We use two sorting steps as follows (again, the input bug reports are initially sorted in the chronological order):

- Step 1: (Sort by prediction in descending order): The prediction result used here is selected from non-filtered training data or the other FARSEC filters. The chosen result is the one with the least number of predicted SBRs. It is used only if the number of predicted SBRs is less than that of ***farsecsq***.
- Step 2: (Sort by prediction of ***farsecsq***): We preserve the order from Step 1 or the chronological order if Step 1 is not used.

The result is a ranked list of predicted bug reports where SBRs are closer to the top of the list than NSBRs.

3.4 Time Complexity for FARSEC

We look at the time complexity of how FARSEC is used for both offline and online computation. In this paper, when we use the training data to find the top security related keywords from the SBRs (feature set), this is an offline process which occurs once with the complexity $O(S \times T_S)$, where S represents the number of SBRs and T_S represents the total number of terms in the SBRs. In addition, building the dictionary of keywords and scores in Algorithm 1 also involves an offline computation with the linear complexity $O(W)$, where W is the number of security related keywords in the feature set. Therefore, the offline computation yields an overall complexity of $O(S \times T_S) + O(W)$ which reduces to $O(S \times T_S)$ since $T_S > W$. The online computation occurs when scoring bug reports. In the worst case, this is an $O(N \times W)$ operation, where N is the number of bug reports in the training data.

The time complexity for the ranking of bug reports in test data is seen as an offline computation with an overall complexity of *O(log(N′))*, where N′ is the number of bug reports in the test data. This log complexity is due to the *Quicksort* algorithm [34] used for Step 1 and Step 2 in Section 3.3.

4 Experiment Setup

Prediction models are created with labelled historical data and machine learning algorithms. In this section, we describe 1) the data and the data pre-processing steps we use to get the top 100 security related keywords for each project; 2) a noise detection algorithm used in defect prediction and how it compares with the FARSEC filters, and 3) the machine learning algorithms used and the different performance measures of the prediction models. The code used in our experiments along with the data and results are available online [5].

4.1 Data

To conduct experiments and answer the research questions posed in this paper, we need projects with historical bug reports, which are labelled as SBRs or NSBRs. We use a total of five projects: four from Ohira et al. [35] and a subset of bug reports from the Chromium project. Table 1 shows the domain of each project, the submit date of the first and last bug reports, the number of reports for each project, and the number and percentage of SBRs. The entries are sorted in ascending order of SBR (%).

There are six kinds of high impact bugs reports in the datasets from Ohira et al. [35]. These include, *surprise*, *dormant*, *blocking*, *security*, *performance* and *breakage*. Each of these four open source Apache projects uses JIRA [6] as its bug tracking system and each project is from a different application domain. Since Ohira et al. [35] focused on high impact bugs, they randomly selected one thousand bug reports with a BUG or IMPROVEMENT label for each project. Graduate students and faculty members labelled these bug reports. The Chromium dataset comes from the 2011 mining challenge of the Mining Software Repositories conference [7]. Here security bugs are labelled as *Bug-Security* when they are submitted to the system. In this paper we focus on the prediction of security bug reports, therefore we treat all other types of bug reports as non-security bug reports.

The bug reports from Ohira et al. [35] are supplied as comma separated value (CSV) files. Each row represents a bug report and the columns are features of the reports such as *bug id*, *title*, *description*, and *date and time* a report was submitted and fixed. In contrast, the Chromium bug reports are supplied as a directory of html files. For uniformity, we first convert Chromium's html files into a single CSV file with the column headers, `id`, `date`, `report`, and `security`. Like the datasets of Ohira et al. [35], each row represents a bug report. From each report we extracted the date created and the security label with values represented as 1 for SBRs and

5. https://bitbucket.org/fayola21-lero/farsec47
6. JIRA: https://www.atlassian.com/software/jira/
7. http://2011.msrconf.org/msr-challenge.html

TABLE 1
Characteristics of the projects and bug reports

Project	Domain	Start Date	End Date	BRs	SBRs	SBRs(%)
Chromium	Web browser called Chrome.	Aug 30 2008	Jun 11 2010	41,940	192	0.5
Wicket	Component-based web application framework for the Java programming.	Oct 20 2006	Nov 9 2014	1,000	10	1.0
Ambari	Hadoop management web UI backed by its RESTful APIs.	Sep 26 2011	Aug 8 2014	1,000	29	3.0
Camel	A rule-based routing and mediation engine.	Jul 8 2007	Sep 18 2013	1,000	32	3.0
Derby	A relational database management system.	Sep 28 2004	Sep 17 2014	1,000	88	9.0

0 for NSBRs. The ids came from the names of the html files. For the textual reports we stripped the html tags from the files, and removed comments about the initial bug report.

Originally, there are $49,986$ bug reports downloaded for Chromium. We reduce this number to $41,940$ by excluding those reports which show a *404 not found* error (e.g. 46314.html and higher) and those which require a username and password to gain access (e.g. 43307.html and 44868.html).

To further prepare the data for our experiments, we *scrubbed* the files by selecting only the necessary columns. From the data of Ohira et al. [35], we selected `issue_id`, `description`, `summary` and `Security`. We then combined description and summary columns. At this point the data of each project is partitioned into two parts according to the dates the reports were created. The first part (past) is used for training while the second part (present) is used for testing. Security related keywords are identified from the training data and used to produce text-by-frequency matrices for each project (Section 3.1).

4.2 Noise Detection

To evaluate FARSEC filters, we include a comparison with the noise detection algorithm Closest List Noise Identification (CLNI). Kim et al. [36] proposed this method for dealing with noise in defect prediction, which they found to have reasonable accuracy. CLNI works as follows: For each instance I, compute its euclidean distance from other instances. Select the N nearest neighbors of I and find portion θ of them with a different label from I. If θ is greater than a specified threshold, then I has a high probability of being noisy. The noisy instances are removed at each iteration. The above process continues until the similarity (ϵ) between the set of noisy instances from the previous iteration and the current iteration is greater than or equal to 0.99.

In order to have a fair comparison with FARSEC filters, CLNI is modified and used in our experiments as follows. To deal with the issue of imbalanced data as well as the complexity of the algorithm, we first elected to only remove noisy NSBRs, i.e. NSBRs with security cross words. Hence we kept all the SBRs in the filtered datasets. This matches what we do with FARSEC filters. Second, while Kim et al. [36] suggested that $\theta = 0.6$, the percentage of SBRs in our training data only range from 0.5 to 9%, therefore we use $\theta = \frac{SBRs}{NSBRs+SBRs}$ for each project. We use the suggested default values of all other input to CLNI, i.e. $N = 5$ and $\epsilon = 0.99$.

Second, the use of the nearest neighbor algorithm in CLNI causes its time complexity to be dependent on the number of reports in a dataset. While noisy instances are removed for smaller datasets like Ambari in a few minutes, it can take hours for the larger Chromium dataset to complete. To help reduce the complexity of the algorithm, we first select the 100 nearest NSBR neighbors of each SBR. These NSBRs have the most potential to be considered as noisy reports by CLNI.

Finally, we create three additional FARSEC filters using CLNI. We apply each FARSEC filter to the CLNI filtered data by removing any NSBRs with scores above 0.75. We denote these as ***clnifarsec***, ***clnifarsectwo*** and ***clnifarsectwo***. These comparisons will provide additional evidence about the impact of the presence security cross words on prediction results.

4.3 Machine Learning Algorithms

For our experiments we use five machine learning algorithms, namely, Random Forest, Naive Bayes, Logistic Regression, Multilayer Perceptron and K-Nearest Neighbor. We chose these because they are used widely in the software defect prediction literature [37]. In a benchmark study, Lessmann et al. [37], found that out of 22 machine learning algorithms, the top 17 had no significant difference in their predictive performance. Therefore, we restrict our experiments to four algorithms out of the top 17 and K-Nearest Neighbor, which is the top performer of the remaining five algorithms. We now give a brief overview of these algorithms and their use in this work.

1: Random Forest is shown to be relatively better than 21 other algorithms in the study by Lessmann et al. [37] and it works well on imbalanced data [38]. Breiman [39] described Random Forest as a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. In other words, it is a collection of trees, where each tree is grown from a bootstrap sample (randomly sampling the data with replacement). Additionally, the attributes used to find the best split at each node is a randomly chosen subset of the total number of attributes. Each tree in the collection is used to classify a new instance. The forest then selects a classification by choosing the majority result.

2: Naive Bayes is generally regarded as one of the most efficient and effective algorithms in machine learning [40], despite the feature independence assumption, which rarely seems to hold in real-world applications. Menzies et al. [41] and Lessmann et al. [37] reported in separate studies that it performed well for software defect prediction compared to more complex learning algorithms.

3: Logistic Regression is generally appropriate when the dependent variable is dichotomous (e.g. either fault-prone

or non-fault-prone) [42] and was used by Zimmermann et al. [15] and Weyuker et al. [43] for software defect predictions.
4: Multilayer Perceptron is a neural network classifier and has the ability to solve problems stochastically. It is a feed-forward neural network with one or more layers between input and an output layer [44]. It is trained with a back-propagation learning algorithm and can solve problems that are not linearly separable. The network nodes are sigmoid unless the class is numeric so the nodes become linear units.
5: K=1 Nearest Neighbor is a simple instance based classifier. Cover and Hart [45] described K-Nearest Neighbor as a simple non-parametric decision procedure which classifies x, an unknown instance in the category of its nearest neighbor. It is one of the simplest algorithms and is used to set a baseline in our work.

We use the Weka implementations of these machine learning algorithms with their default parameter values [46]. When using Logistic Regression, the default threshold value is 0.5, thus *false negatives* and *false positives* are treated equally. Also, in Weka, $K = 1$ Nearest Neighbor is represented by Ib-k=1, an instance based method.

4.4 Performance Measures

To evaluate the performance of the prediction models built with FARSEC filtered data, as well as the ranked bug reports, we use the measures shown in Figure 3. For the performance of the prediction models, the confusion matrix in Figure 3 is used, where TP, TN, FP and FN are true positive, true negative, false positive and false negative respectively. We report the probability of detection *pd*, probability of false alarm *pf*, precision, f-measures and g-measures. Probability of detection measures the fraction of the actual SBRs the predictor finds The probability of false alarm measures the fraction of the NSBRs that are incorrectly predicted as SBRs. Precision (*prec*) measures the fraction of actual SBRs in the predicted SBRs.

		Actual	
		SBRs	NSBRs
Predict	SBRs	TP	FN
	NSBRs	FP	TN
	pd	$\frac{TP}{TP+FN}$	
	pf	$\frac{FP}{FP+TN}$	
	prec	$\frac{TP}{TP+FP}$	
	f-measure	$\frac{2*pd*prec}{pd+prec}$	
	g-measure	$\frac{2\times pd\times(100-pf)}{pd+(100-pf)}$	
Rank	AP$_n$	$\sum_{k=1}^{n}\frac{P(k)}{n}$	
	MAP$_n$	$\sum_{i=1}^{N}\frac{AP_{n_i}}{N}$	

Fig. 3. Confusion matrix: Definitions of pd, pf, prec, g-measure and the mean average precision (MAP).

The *f-measure* and *g-measure* are calculated using the pd, prec and pf values. The f-measure is the harmonic mean between pd and prec. The optimal f-measure value is 100%. The g-measure is the harmonic mean of pd and (100-pf). 100-pf represents a value is known as *specificity* (not predicting NSBRs as SBRs). Specificity is used together with pd to form the G-mean2 measure, which is the geometric mean of the pds for both the majority and the minority classes [47]. In our case, we use these to form the g-measure. Finally, to measure any significant differences between the non-filtered and FARSEC filtered results, we use a two-tailed Mann Whitney statistical test with 95% confidence [48].

To determine the usefulness of FARSEC for ranked lists of bug reports, we use *mean average precision (MAP)* [49]. MAP is a metric used in information retrieval to measure the relevance of the top n results of a query. We use MAP to show that FARSEC ranking can return results with actual SBRs closer to the top of the list of all predicted SBRs. The formulas to calculate MAP are shown in the last row of Figure 3, where the *average precision (AP)* and then MAP are calculated. The average precision is measured cumulatively up to n which is the number of predicted SBRs, $P(k)$ is the precision at point k in the list of bug reports. For our experiments, we divided the test data into deciles and the MAP_n is calculated cumulatively for each decile.

5 Experiment Design and Results

To evaluate FARSEC, we use both the within and transfer learning techniques for building prediction models (recall Section 2.2.2). We denote these as Within Project Prediction (WPP) and Transfer Project Prediction (TPP) respectively. In our context, WPP uses labelled, historical bug reports from a project to predict SBRs in the unlabelled bug reports of the same project. TPP uses labelled bug reports from one project (but could be from many projects), to predict unlabelled bug reports of another project. We refer to the labelled data from other projects as the *source* and the unlabelled data for the project we are building the predictor for, as the *target*.

In this paper we build WPP and TPP models using datasets that are unfiltered, FARSEC filtered, and CLNI filtered. WPP can be used when a project has enough historical data to build prediction models. TPP is used for projects with insufficient data available to build prediction models. When preparing data for TPP, we use security related keywords from the source to construct the term-document frequency matrix and the predictor for the target project.

Table 2 shows the outcome of applying different filters in terms of the number of remaining bug reports and the SBR %. Since we only remove NSBRs with a score above 0.75, the number of SBRs remain the same with and without filtering. This allows for an increase in the percentage of SBRs for building prediction models. There are four interesting cases shown in the table. The first two are the ***farsecsq*** and ***clnifarsecsq*** filters for Derby. The prediction models are built with training data where the majority are SBRs, 81% and 96% respectively. The last two cases concern the limited reduction with ***farsectwo*** and ***farsec*** for Chromium. This is due to the threshold used to distinguish between NSBRs and SBRs. There are few Chromium reports with scores ≥ 0.75.

All experiments are designed around the research questions posed in the Section 1.

- RQ1: Can security cross words lead to mislabelled security bug reports by prediction models?
- RQ2: How do we build *effective* prediction models for security bug reports when data scarcity is an issue?

TABLE 2
Characteristics of the training data

Source	Filter	#SBRs	#BRs	SBRs(%)
Chromium	train	77	20970	0.4
	farsecsq		14219	0.5
	farsectwo		20968	0.4
	farsec		20969	0.4
	clni		20154	0.4
	clnifarsecsq		13705	0.6
	clnifarsectwo		20152	0.4
	clnifarsec		20153	0.4
Wicket	train	4	500	0.8
	farsecsq		136	2.9
	farsectwo		143	2.8
	farsec		302	1.3
	clni		392	1.0
	clnifarsecsq		46	8.7
	clnifarsectwo		49	8.2
	clnifarsec		196	2.0
Ambari	train	22	500	4.4
	farsecsq		149	14.8
	farsectwo		260	8.5
	farsec		462	4.8
	clni		409	5.4
	clnifarsecsq		76	28.9
	clnifarsectwo		181	12.2
	clnifarsec		376	5.9
Camel	train	14	500	2.8
	farsecsq		116	12.1
	farsectwo		203	6.9
	farsec		470	3.0
	clni		440	3.2
	clnifarsecsq		71	19.7
	clnifarsectwo		151	9.3
	clnifarsec		410	3.4
Derby	train	46	500	9.2
	farsecsq		**57**	**80.7**
	farsectwo		185	24.9
	farsec		489	9.4
	clni		446	10.3
	clnifarsecsq		**48**	**95.8**
	clnifarsectwo		168	27.4
	clnifarsec		435	10.6

- RQ3: How do we generate *useful* lists of ranked bug reports?

To answer the research questions we first generate datasets for our experiments by organizing the bug reports into training and test data. We then apply the text mining steps described in Section 3.1. Recall that some suspected SBRs have been publicly disclosed due to the lack of security domain knowledge of some bug reporters [3] and also by those with knowledge who disregard requests to submit these reports privately [4]. Therefore, for our experiments we use past data to predict the *present*. The present represents any new unlabelled or mislabelled bug reports entering the bug tracking system. Once in the system, they are available publicly, therefore to avoid any suspected SBRs being made public before they are accessed, it is important to identify them and direct them to the security engineers. Therefore, to build each prediction model we sort the data according to their bug report numbers in ascending order and then we do a 50% split on the bug reports into training (past) and test (present) data in order to identify suspected SBRs for assessment prior to public disclosure.

Table 3 shows the total number of experiments performed. With five machine learning algorithms, eight treatments and four sources for WPP and TPP, we perform a total of 200 experiments on each project.

The following section describes the approach and results which answer each research question.

TABLE 3
Number of prediction models for each dataset

Treatments	Prediction Models	Total
WPP	1 data set ×	
	5 machine learning algorithms ×	
	8 (train + FARSEC filtering + CLNI)	40
TPP	1 data set ×	
	4 sources ×	
	5 machine learning algorithms ×	
	8 (train + FARSEC filtering + CLNI)	160
Total		**200**

5.1 RQ1: Can security cross words lead to mislabelled security bug reports by prediction models?

Approach. To determine whether security cross words can cause bug reports to be mislabelled by prediction models, we compare FARSEC filters with the CLNI filter. While FARSEC filters are designed to reduce the number of security cross words in prediction models, the CLNI filter removes noisy NSBRs. We do two comparisons between the two filter types. The first is in terms of the reduction of the number of security cross words in the training data after filtering. The second is in terms of the number of SBRs mislabelled by the prediction models, which are created with the filtered data.

To determine the number of security cross words in training data, we first need to generate the feature sets for each project (Section 3.1). From the feature sets, we use tf-idf as an indicator for security cross words. Any keyword with a tf-idf value above zero is considered to be a security cross word, i.e. present in both SBRs and NSBRs. The number of mislabelled SBRs are found by building WPP models with the filtered training data and machine learning algorithms (described in Section 4.3). The models are then evaluated with the test data.

Results. ***FARSEC filters reduce the number of security cross words in training data leading to fewer mislabelled SBRs by WPP models.*** Table 4 shows the number of security cross words present in the NSBRs of the training data before and after filtering. It also shows the top 100 security related keywords, sorted in descending order of their tf-idf values, for each project (feature sets) used in this paper. Highlighted are the keywords that are not security cross words before filtering. Notably, while the number of security cross words in Chromium had minor reductions with FARSEC filters, the other projects had reductions to as low as one word for Derby (the word is *create*). Overall, the median percentage decrease of security cross words for the CLNI filter is 0.4% while for FARSEC filters it is 39%.

When viewing these reductions in conjunction with WPP results in Table 5, we see that in most cases, the FARSEC filters reduced the number of mislabelled SBRs (shown in the FN column). In most cases, the CLNI filter either maintained or increased the number of mislabelled SBRs

TABLE 4
Top 100 security related keywords with the number of security cross words (the highlighted keywords are not security cross words)

Source	Security Cross Words (SCWs)		Security Related Keywords
	Filter	# SCWs	
Chromium	train	100	file security chrome page http download user starred person notified changes may see url site
	farsecsq	95	bug open google browser like windows window https web code one memory firefox function
	farsectwo	100	tests problem seems tab also version use would using view used make users chromium crash
	farsec	100	click password think vulnerability sure browsers link attached attacker data get fix const content
	clni	100	something safari new error javascript lcamtuf malicious please could risk release try found allow
	clnifarsecsq	95	expected time example corruption test back access crashes urls int without know versions way
	clnifarsectwo	100	uses cause fail want system still files arbitrary html details ssl need loaded might
	clnifarsec	100	
Wicket	train	74	**statelesshomepage** attached calls **dataprovider** regards component limited files count **jan** couple
	farsecsq	12	database integer entries reason page get first java error unknown source **signinform** jetty
	farsectwo	13	requestlistenerinterface **credentialsexpected** params http exception interfacerequest homepage
	farsec	40	manually become **inform** stateful listener **happends** **statelesschecker** **insucceeded** creates
	clni	74	causes **sucessfull** **opensactual** final **abstractlistener** quickstart mvn login **analysis** open **unpack**
	clnifarsecsq	12	temporary enter sign **valuemap** happy **statelessproblem** signinpanel requestcycle occured fix
	clnifarsectwo	13	delete **deleting** fails upload org webapplication **hole** **space** created handle example secu-
	clnifarsec	40	rity server find important method multipart easy hope makes **incomplete** **cancelled** think
			workaround uploading **parserequest** want really anyone **disk** large **eating** bug throw **posting**
			developers call threads **servletwrapper**
Ambari	train	95	security wizard service secure fails permissions allow start validation user cluster cannot
	farsecsq	25	set make page configs datanode add use default property name request instead enable web
	farsectwo	57	ssl password used services error disable fix permission options executed setup http nagios
	farsec	88	registration hosts also change url configuration try enabling check disabled host install time
	clni	94	return provide call script issue file failures principal **touched** incorrect artifacts **assignments**
	clnifarsecsq	25	slaves side username directory path customized **effects** mode unwanted false causes broken
	clnifarsectwo	57	primary testmode mapred names working httpd state missing **navigation** ganglia **prepare**
	clnifarsec	88	locked master ambari smoke wrong hbase node test zookeeper back need either true
Camel	train	88	message http endpoint header org would uri component files also issue stop expose see static
	farsecsq	27	jetty endpoints specified port processing login route file error one messages headers support
	farsectwo	47	server lines using throws provide currently instance ftp sent means use issues ignore license
	farsec	82	dsl **interfaces** opened host fails redelivery **auth** interceptsendtoendpoint used connection check
	clni	88	**memory** **heap** nabblehttp bundles consumer pass path new case based classes however default-
	clnifarsecsq	27	errorhandler **pushing** **expectedbody** **resultendpoint** exchange following exception else apache
	clnifarsectwo	47	null **inoptionalout** handler token custom ignores servers something like protected **sftpendpoint**
	clnifarsec	82	remove jms concatenated **delegate** underlying **ssh** data copy **ftpcomponent** method second csv
			results mockendpoint false
Derby	train	95	org security server test derby **permission** java using access tests error support denied junit
	farsecsq	1	file exception database code user read manager locale run fails need version fail running
	farsectwo	72	**securitymanager** network following files call required statement source table connect class
	farsec	90	thread block **securityexception** would used like failed problem **privileged** client see jdbc set
	clni	94	method **filepermission** trying granted authentication needs directory connection new think
	clnifarsecsq	1	encryption sun jar policy start stack unknown rows revoke found information alpha thrown
	clnifarsectwo	70	without name one create end update http could make trigger though native contains looks two
	clnifarsec	89	key mode results sql use int classpath message incorrectly check

when compared with the non-filtered results (*train*). The exception is Camel where the CLNI reduced the mislabelled reports from 16 to 15.

Taking a closer look at the results in Table 5, the ***farsec*** filter has a relatively similar performance to CLNI. For three out of five projects, their numbers of mislabelled SBRs match. This could be due to the fact that the filter does not use a bias on the keywords found in the SBRs (Section 3.2). However for Wicket and Derby, the ***farsec*** filter produces fewer mislabelled SBRs than the CLNI filter. This could be due to the average percentage decrease of security cross words for the ***farsec*** filter, which is 32 times larger than that of CLNI.

There are other notable results that can be gleaned from Table 5. First, we found that there is no significant difference in the f-measure and g-measure results for each filter over the five projects. We used a two-tailed Mann Whitney test at 95% confidence. The results of the statistical test are not robust when the population size is less than 10. Therefore, we are cautious about conclusions formed based on this. Second, there does not seem to be any preference for a machine learning algorithm. Although each project has one dominant algorithm preference, others also contribute. An interesting case is `ib_k` which only appears for Derby and only in the cases where the SBRs are the majority in the training data, i.e. with filters ***farsecsq*** and ***clnifarsecsq*** (Table 2). Finally, the bold results in the table show that the FARSEC filters (***clnifarsecsq*** and ***farsectwo***) overall, had relatively better performance than the other filters according to the g-measures.

5.2 RQ2: How do we build *effective* prediction models for security bug reports when data scarcity is an issue?

Approach. When data scarcity is an issue, transfer learning is used to build prediction models (Section 2.2.2). We define an *effective* TPP model as one whose performance is comparable to or better than that of WPP models. Therefore, to build effective TPP models for SBRs, we use the training data from the previous experiment for WPP models as the source and generate new test data based on their feature

TABLE 5
WPP results with FARSEC and CLNI filtering (those with the highest g-measures are highlighted)

Target	Filter	Learner	TN	TP	FN	FP	pd	pf	prec	f-measure	g-measure
Chromium	train	logistic_regression	20815	18	97	40	15.7	0.2	31.0	20.8	27.1
	farsecsq	random_forest	20801	17	98	54	14.8	0.3	23.9	18.3	25.7
	farsectwo	logistic_regression	20815	18	97	40	15.7	0.2	31.0	20.8	27.1
	farsec	logistic_regression	20815	18	97	40	15.7	0.2	31.0	20.8	27.1
	clni	logistic_regression	20808	18	97	47	15.7	0.2	27.7	20.0	27.1
	clnifarsecsq	**multilayer_perceptron**	**20066**	**57**	**58**	**789**	**49.6**	**3.8**	**6.7**	**11.9**	**65.4**
	clnifarsectwo	logistic_regression	20808	18	97	47	15.7	0.2	27.7	20.0	27.1
	clnifarsec	logistic_regression	20808	18	97	47	15.7	0.2	27.7	20.0	27.1
Wicket	train	naive_bayes	459	1	5	35	16.7	7.1	2.8	4.8	28.3
	farsecsq	logistic_regression	305	4	2	189	66.7	38.3	2.1	4.0	64.1
	farsectwo	**logistic_regression**	**313**	**4**	**2**	**181**	**66.7**	**36.6**	**2.2**	**4.2**	**65.0**
	farsec	logistic_regression	454	2	4	40	33.3	8.1	4.8	8.3	48.9
	clni	naive_bayes	467	0	6	27	0.0	5.5	0.0	0.0	0.0
	clnifarsecsq	logistic_regression	368	2	4	126	33.3	25.5	1.6	3.0	46.1
	clnifarsectwo	logistic_regression	357	2	4	137	33.3	27.7	1.4	2.8	45.6
	clnifarsec	logistic_regression	442	3	3	52	50.0	10.5	5.5	9.8	64.2
Ambari	train	multilayer_perceptron	485	1	6	8	14.3	1.6	11.1	12.5	24.9
	farsecsq	random_forest	422	3	4	71	42.9	14.4	4.1	7.4	57.1
	farsectwo	**random_forest**	**478**	**4**	**3**	**15**	**57.1**	**3.0**	**21.1**	**30.8**	**71.9**
	farsec	multilayer_perceptron	469	1	6	24	14.3	4.9	4.0	6.3	24.8
	clni	multilayer_perceptron	480	1	6	13	14.3	2.6	7.1	9.5	24.9
	clnifarsecsq	random_forest	455	4	3	38	57.1	7.7	9.5	16.3	70.6
	clnifarsectwo	random_forest	471	2	5	22	28.6	4.5	8.3	12.9	44.0
	clnifarsec	random_forest	493	1	6	0	14.3	0.0	100.0	25.0	25.0
Camel	train	logistic_regression	464	2	16	17	11.1	3.5	10.5	10.8	19.9
	farsecsq	random_forest	426	3	15	55	16.7	11.4	5.2	7.9	28.1
	farsectwo	**logistic_regression**	**280**	**9**	**9**	**201**	**50.0**	**41.8**	**4.3**	**7.9**	**53.8**
	farsec	logistic_regression	448	3	15	33	16.7	6.9	8.3	11.1	28.3
	clni	naive_bayes	422	3	15	59	16.7	12.3	4.8	7.5	28.0
	clnifarsecsq	multilayer_perceptron	415	3	15	67	16.7	13.9	4.3	6.8	27.9
	clnifarsectwo	multilayer_perceptron	445	2	16	37	11.1	7.7	5.1	7.0	19.8
	clnifarsec	logistic_regression	458	3	15	24	16.7	5.0	11.1	13.3	28.4
Derby	train	naive_bayes	427	16	26	31	38.1	6.8	34.0	36.0	54.1
	farsecsq	ib_k	321	23	19	137	54.8	29.9	14.4	22.8	61.5
	farsectwo	**random_forest**	**401**	**20**	**22**	**57**	**47.6**	**12.4**	**26.0**	**33.6**	**61.7**
	farsec	naive_bayes	429	16	26	29	38.1	6.3	35.6	36.8	54.2
	clni	random_forest	456	10	32	2	23.8	0.4	83.3	37.0	38.4
	clnifarsecsq	ib_k	321	23	19	137	54.8	29.9	14.4	22.8	61.5
	clnifarsectwo	random_forest	416	15	27	42	35.7	9.2	26.3	30.3	51.3
	clnifarsec	naive_bayes	427	16	26	31	38.1	6.8	34.0	36.0	54.1

sets. For example, if the Wicket project is new and has little or no data to build a prediction model, we can use Ambari or another project as the source. Therefore the feature set of Ambari is used by Wicket to calculate the frequencies of each keyword for each bug report. In the end we have a test set for Wicket based on the feature set of another project. Note that in the context of our experiments, scarcity can also mean little or no SBRs present for building the models.

As shown in Table 3, we build 160 TPP models in our experiments, including the use of filtered sources. To evaluate the effectiveness of these models, we compare them with the WPP models. Specifically, we determine if the performance of the TPP models has degraded or improved from the WPP models in terms of f-measure, g-measure and the number of mislabelled SBRs. We use the Mann Whitney statistical test for comparison.

Results. ***When data scarcity is an issue, other sources can be used to build effective prediction models.*** Table 6 shows the best TPP results with FARSEC and CLNI filtering (according to f-measures). From these results, the **bold** ones indicate those with the highest g-measures.

Overall, there is no significant difference in TPP model performances when compared with WPP models. This is the case for the f-measures, g-measures and the number of mislabelled SBRs. However, a closer look at the results in Table 6 reveals five notable points:

1) TPP models for train and CLNI improved the f-measures and g-measures for the projects with the lowest number of SBRs in their training data (i.e. Wicket, Ambari and Camel). This confirms the usefulness of TPP when dealing with data scarcity, especially when SBRs are few. Furthermore, for Chromium and Derby, the CLNI filter TPP model has higher g-measures than its WPP counterparts.
2) The pfs for the TPP models are all below 25% while some of the FARSEC filtering WPP models have pfs above 25%. Namely Wicket, Camel and Derby.
3) No single source stands out. FARSEC filtered Chromium is the best source for Ambari and Derby. Ambari works best for Chromium, Derby is best for Camel and Camel is best for Wicket. However Wicket only works well for Derby, but it is not the best result.
4) Similar to the WPP result, there is no preference for a particular machine learning algorithm. However, except for Wicket, each project has a dominant algorithm. For Chromium it is `random_forest` and

TABLE 6
TPP results with FARSEC and CLNI filtering (those with the highest g-measures are highlighted)

Target	Source	Filter	Learner	TN	TP	FN	FP	pd	pf	prec	f-measure	g-measure
Chromium	Derby	train	random_forest	20835	2	113	20	1.7	0.1	9.1	2.9	3.4
	Ambari	farsecsq	random_forest	19279	34	81	1576	29.6	7.6	2.1	3.9	44.8
	Ambari	farsectwo	random_forest	20454	53	62	401	46.1	1.9	11.7	18.6	62.7
	Derby	farsec	multilayer_perceptron	20502	12	103	353	10.4	1.7	3.3	5.0	18.9
	Camel	clni	logistic_regression	20262	25	90	593	21.7	2.8	4.0	6.8	35.5
	Ambari	**clnifarsecsq**	**random_forest**	**19817**	**56**	**59**	**1038**	**48.7**	**5.0**	**5.1**	**9.3**	**64.4**
	Derby	clnifarsectwo	random_forest	20332	26	89	523	22.6	2.5	4.7	7.8	36.7
	Camel	clnifarsec	multilayer_perceptron	20590	8	107	265	7.0	1.3	2.9	4.1	13.0
Wicket	**Camel**	**train**	**naive_bayes**	**437**	**3**	**3**	**57**	**50.0**	**11.5**	**5.0**	**9.1**	**63.9**
	Chromium	farsecsq	multilayer_perceptron	475	1	5	19	16.7	3.8	5.0	7.7	28.4
	Camel	farsectwo	random_forest	490	1	5	4	16.7	0.8	20.0	18.2	28.5
	Camel	farsec	naive_bayes	431	3	3	63	50.0	12.8	4.5	8.3	63.6
	Ambari	clni	multilayer_perceptron	476	1	5	18	16.7	3.6	5.3	8.0	28.4
	Chromium	clnifarsecsq	random_forest	493	1	5	1	16.7	0.2	50.0	25.0	28.6
	Camel	clnifarsectwo	random_forest	489	1	5	5	16.7	1.0	16.7	16.7	28.5
	Camel	clnifarsec	naive_bayes	433	3	3	61	50.0	12.3	4.7	8.6	63.7
Ambari	Derby	train	multilayer_perceptron	484	2	5	9	28.6	1.8	18.2	22.2	44.3
	Chromium	**farsecsq**	**multilayer_perceptron**	**474**	**3**	**4**	**19**	**42.9**	**3.9**	**13.6**	**20.7**	**59.3**
	Chromium	farsectwo	naive_bayes	472	3	4	21	42.9	4.3	12.5	19.4	59.2
	Camel	farsec	multilayer_perceptron	492	1	6	1	14.3	0.2	50.0	22.2	25.0
	Derby	clni	multilayer_perceptron	477	2	5	16	28.6	3.2	11.1	16.0	44.1
	Chromium	clnifarsecsq	random_forest	492	1	6	1	14.3	0.2	50.0	22.2	25.0
	Chromium	clnifarsectwo	naive_bayes	474	2	5	19	28.6	3.9	9.5	14.3	44.1
	Camel	clnifarsec	multilayer_perceptron	492	1	6	1	14.3	0.2	50.0	22.2	25.0
Camel	Derby	train	naive_bayes	457	3	15	24	16.7	5.0	11.1	13.3	28.4
	Chromium	farsecsq	naive_bayes	401	7	11	80	38.9	16.6	8.0	13.3	53.0
	Derby	**farsectwo**	**naive_bayes**	**371**	**8**	**10**	**110**	**44.4**	**22.9**	**6.8**	**11.8**	**56.4**
	Ambari	farsec	logistic_regression	439	5	13	42	27.8	8.7	10.6	15.4	42.6
	Ambari	clni	logistic_regression	444	5	13	37	27.8	7.7	11.9	16.7	42.7
	Chromium	clnifarsecsq	naive_bayes	402	7	11	79	38.9	16.4	8.1	13.5	53.1
	Ambari	clnifarsectwo	random_forest	455	3	15	26	16.7	5.4	10.3	12.8	28.3
	Derby	clnifarsec	multilayer_perceptron	473	2	16	8	11.1	1.7	20.0	14.3	20.0
Derby	Ambari	train	naive_bayes	393	13	29	65	31.0	14.2	16.7	21.7	45.5
	Chromium	farsecsq	naive_bayes	370	19	23	88	45.2	19.2	17.8	25.5	58.0
	Ambari	farsectwo	random_forest	454	6	36	4	14.3	0.9	60.0	23.1	25.0
	Wicket	farsec	naive_bayes	354	19	23	104	45.2	22.7	15.4	23.0	57.1
	Ambari	clni	naive_bayes	400	12	30	58	28.6	12.7	17.1	21.4	43.1
	Chromium	**clnifarsecsq**	**naive_bayes**	**372**	**19**	**23**	**86**	**45.2**	**18.8**	**18.1**	**25.9**	**58.1**
	Ambari	clnifarsectwo	random_forest	449	8	34	9	19.0	2.0	47.1	27.1	31.9
	Wicket	clnifarsec	naive_bayes	363	18	24	95	42.9	20.7	15.9	23.2	55.6

for Ambari it is `multilayer_perceptron`. Camel and Derby work best with `naive_bayes`. Unlike the WPP results, `ib_k`, does not appear for any of the projects.

5) Apart from Wicket, the FARSEC filtered models worked best for all the projects (according to their g-measures shown in Table 6). In addition, these best results had lower numbers of mislabelled SBRs than the train and CLNI models. For Wicket, the ***farsec*** and ***clnifarsec*** filters are comparable to the best result (train) with matching number of predicted SBRs (TPs).

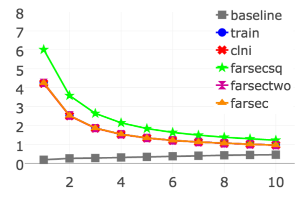
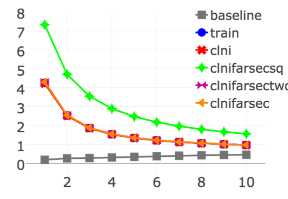
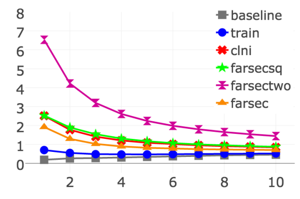
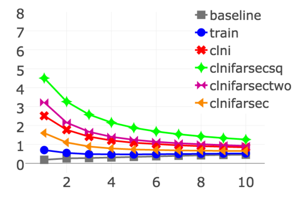
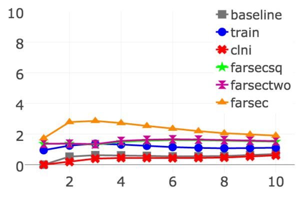
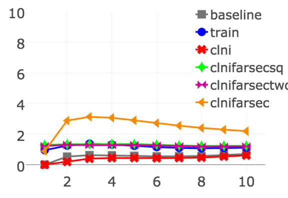
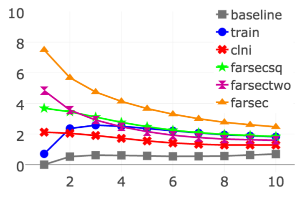
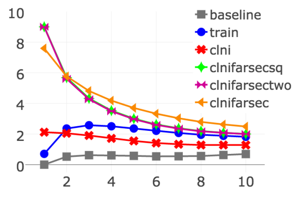
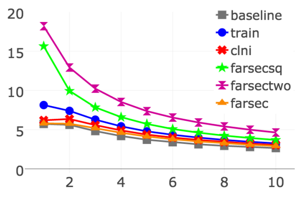
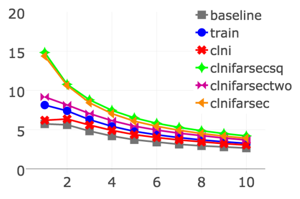
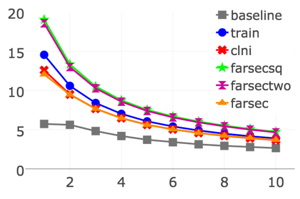
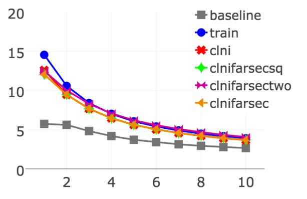
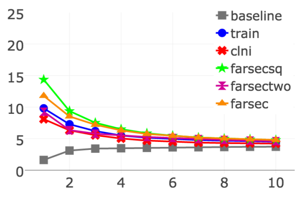
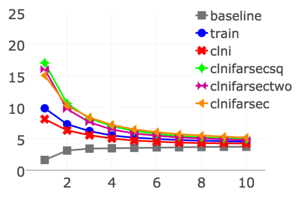
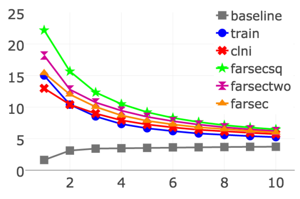
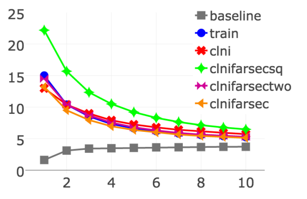
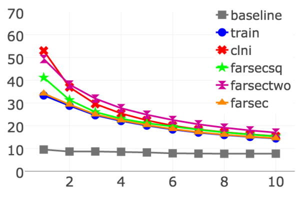
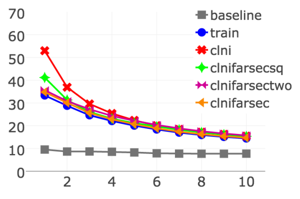
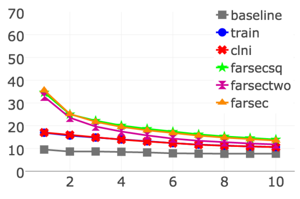
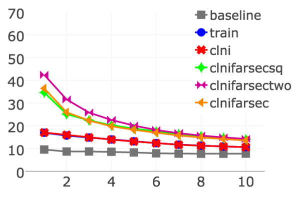
5.3 RQ3: How do we generate *useful* lists of ranked bug reports?

Approach. Results in Table 5 and Table 6, show that some of the best results with FARSEC filtering have high FPs, while results without filtering tend to have lower FPs. For a security engineer, using an approach with high FPs is akin to "finding a needle in a haystack" [5], and is not useful even if there are more SBRs in the results. In order to alleviate this problem we generate useful lists of ranked bug reports, sorted according to the steps described in Section 3.3. For non-filtered WPP and TPP models we rank the bug reports using Step 1 and Step 2 described in Section 3.3. For the models built with FARSEC filtered data, we follow both steps in Section 3.3. This sorts the prediction results with the least number of predicted SBRs with those from the FARSEC filters. This ranking method takes advantage of the lower FPs even if there is only one actual SBR present. We evaluate it using the cumulative mean average precision (MAP) over 10 deciles (see Figure 3).

Results. ***SBRs are ranked relatively highly with FARSEC.*** The cumulative MAPs over deciles of the best prediction results for each project are shown in Table 7. The left column shows the results for WPP and the right column, TPP. Each chart has a *baseline*, train (non-filtered results) and CLNI results. The baseline is the bug reports ranked chronologically (i.e., ascending order of the bug report numbers). For clarity, we have generated four charts for each project to show any differences in results for FARSEC filters with and without CLNI. The ranked lists with the higher MAP values show that more actual SBRs are closer to the top of the lists.

Overall, FARSEC ranking shows promise, outperforming *train* and *CLNI* results in nine out of ten cases Table 7. The exception is the CLNI results for Derby shown in the charts *q* and *r*, where CLNI at the first decile outperforms

TABLE 7
The cumulative MAPs over deciles of the best prediction results for each project.

WPP		TPP	
 (a) Chromium MAP	 (b) Chromium MAP	 (c) Chromium MAP	 (d) Chromium MAP
 (e) Wicket MAP	 (f) Wicket MAP	 (g) Wicket MAP	 (h) Wicket MAP
 (i) Ambari MAP	 (j) Ambari MAP	 (k) Ambari MAP	 (l) Ambari MAP
 (m) Camel MAP	 (n) Camel MAP	 (o) Camel MAP	 (p) Camel MAP
 (q) Derby MAP	 (r) Derby MAP	 (s) Derby MAP	 (t) Derby MAP

train and the FARSEC filters. However from the second decile onward, ***farsectwo*** has the highest MAP.

Other notable results include:

1) The CLNI ranking for Wicket performs worse than the baseline. This agrees with the results in Table 5, which shows zero predicted actual SBRs.
2) For Chromium and Derby, their TPP rankings are relatively worse than WPP. For Wicket, Ambari and Camel, the opposite is true. Their TPP ranking for train, CLNI and the FARSEC filters were relatively better than their WPP rankings.
3) For a few cases where the best results in Table 5 and Table 6 show high FPs, we found that they did not produce the best ranked results in Table 7. Instead they were generally inferior to other FARSEC ranked results. For instance, Table 5 showed Wicket to achieve the best result with ***farsectwo***, however, both ***farsec*** and ***clnifarsec*** had better rankings. Another example in WPP is Camel, where ***farsectwo*** is out-ranked by the other FARSEC ranked results. For

the best TPP results with high FPs, Chromium and Camel are out-ranked by ***farsectwo*** and ***farsecsq*** respectively. This shows that FARSEC ranking can increase the number of actual SBRs at the top of a ranked list of predicted bug reports with less FPs. This blending of results can sometimes yield a better outcome than one *best* result.

6 Discussion and Threats to Validity

In this section we discuss our findings and how they relate to results from related studies. We also discuss the threats to validity of our work.

6.1 Discussion

The findings from our experiments show that removing NSBRs with security cross words mitigates the class imbalance issue. This is supported by the results of our text-based prediction models which produced useful results for security engineers.

The enhanced prediction models also alleviate the issue of mislabelled SBRs. Although we have not verified the results with security engineers, an examination of the top ranked bug reports of Ambari has revealed two SBRs labelled as a NSBRs. Recall that Ambari is one of the four datasets collected and manually labelled by graduate students and faculty [35]. At the time the data was downloaded for our experiments (May 5, 2017), bug report #3292[8] was manually labelled as a NSBR, however it was labelled as a SBR by a prediction model built with the *farsecsq* filter and `multilayer_perceptron` algorithm. In addition, the report is also at position 3 in the first of the 10 deciles in the FARSEC ranked results for Ambari (Table 7k). An excerpt of the report is as follows:

> *Earlier dfs.web.authentication.kerberos.keytab field was being used for NameNode and SNameNode component.*

Even when the models failed to detect mislabelled SBRs, the ranking capability of FARSEC proved useful. Another bug report (#3153 shown in Figure 1) was manually mislabelled as a NSBR, but FARSEC ranked it at position 30, in the first of the 10 deciles in the ranked results for Ambari (Table 7k). Below is an excerpt from the description of the report:

> *After configuring web authentication yarn client that is assigned yarn service check needs to negotiate 401 HTTP authentication response received while using REST api.*

This shows that FARSEC can detect mislabelled SBRs through prediction and ranking.

However, mislabelling of SBRs is not completely eliminated in models built with FARSEC filters. We found that some SBRs had low scores (below the set threshold of 0.75), while others had scores, which matched those of NSBRs. Therefore, if we forego the machine learning algorithms in favour of report scores, it results in either low FNs with high FPs or low FPs and high FNs (Table 8). When we removed all NSBRs with at least one security cross word, none were left. This meant that the training data would only contain SBRs leaving us with a one-class classification problem [50]. Since we use traditional classification to build prediction models in this work, where the presence of both SBRs and NSBRs are required in the training data, all bug reports in the test data would be labelled as SBRs. In future work we can use one-class classification solutions which would define a boundary around the SBRs, such that it correctly labels as much of them as possible, while minimizing the chance of accepting NSBRs [50].

Developing and applying Transfer Project Prediction (TPP) models for security bugs has been recognized as a challenging problem [3]. Our study has found that TPP can be effective, and with relatively better results that Within Project Prediction (WPP) especially when SBRs are scarce. This result justifies the purpose of TPP, which is to provide useful models for projects with little or no labelled data. In this paper Wicket, Ambari and Camel have the fewest number of SBRs to build prediction models and both the filtered and non-filtered TPP models (Table 7 right column) helped to generate ranked lists of bug reports with higher MAP values than WPP models (Table 7 left column). This suggests that SBR prediction can be generalized, and that there exists a core set for security related keywords which can predict SBRs for any project.

TABLE 8
Predicting SBRs in test data using the bug report scores with different **Support**(x) functions and a threshold of 0.75. The x represents the frequency of each word present in SBRs.

Target	Support(x)	TN	TP	FN	FP	f-measure	g-measure
Chromium	x^2	13370	104	11	7485	2.7	75.0
	$x \times 2$	20854	0	115	1	0.0	0.0
	x	20855	0	115	0	0.0	0.0
Wicket	x^2	140	5	1	354	2.7	42.3
	$x \times 2$	151	4	2	343	2.3	41.9
	x	286	4	2	208	3.7	62.0
Ambari	x^2	87	5	2	406	2.4	28.3
	$x \times 2$	215	5	2	278	3.4	54.2
	x	441	2	5	52	6.6	43.3
Camel	x^2	66	18	1	415	8.0	24.0
	$x \times 2$	142	15	4	339	8.0	43.0
	x	419	3	16	62	7.1	26.7
Derby	x^2	9	42	0	449	15.8	3.9
	$x \times 2$	134	37	5	324	18.4	43.9
	x	454	12	30	4	41.4	44.4

6.2 Threats to Validity

With any empirical study, biases can affect the final results. Therefore, conclusions drawn from this work must be considered with threats to validity in mind. We now report on the external, construct and internal validity of our work.

6.2.1 External Validity

Sampling bias threatens any classification experiment because what happens in one domain may not happen in another. Generalizing our results require multiple projects from different domains with labelled bug reports. Replication experiments are needed in order to repeat, refute, or improve our results.

In addition, the choice of machine learning algorithms for classification could be an issue. Classification is a large and active field and any single study can only use a small

8. https://issues.apache.org/jira/browse/AMBARI-3292

subset of the known classification algorithms. The subset chosen in this study is drawn from a benchmark study of 22 algorithms [37] where the results showed that there was no significant difference in the performance of the top 17. We used four algorithms from the top 17, while the fifth (`ib-k`) was the top performer of the remainder.

6.2.2 Construct Validity

Mislabelling in the training data is an issue for prediction models although results could be improved with manual correction by domain experts [3]. It is possible that some of the reports used to create the prediction models could be mislabelled. To mitigate this, our experiments included multiple datasets, which were labelled by different groups of people.

The use of security cross words that are independent of and specific dataset also helps deal with the issue of mislabelling. As described in related work (Section 2.2.1), there are several methods for creating and expanding keyword lists. This is mitigated somewhat by the transfer learning, where different security related keywords are used on different projects to build prediction models.

6.2.3 Internal Validity

In this study, we did not consider the context in which security cross words appear in both SBRs and NSBRs, and how it may affect prediction results. More work is needed to determine if the use of context as a pre- and/or post-processing mechanism will improve the filtering and ranking capabilities of FARSEC and therefore further reduce the number of mislabelled SBRs by text-based prediction models.

In addition, based on the tf-idf values of each term, we choose the top 100 terms as security related keywords for our experiments. A more exhaustive exploration of whether better results can be achieved by a different number of terms warrant further research.

7 Conclusions and Future Work

When mislabelled security bug reports are publicly disclosed in bug tracking systems before security engineers can address them, it presents attackers with a window of opportunity to exploit the security vulnerabilities. With bug tracking systems containing thousands of bug reports with only a tiny fraction of those described as security bugs, security engineers are faced with the problem of trying to find a needle in a haystack [5]. In other words, to find actual SBRs, security engineers would have to comb through hundreds or thousands of bug reports. To help ease the burden of the search, they employ prediction models, usually based on text mining methods.

This paper presented FARSEC, a novel approach to reduce the mislabelling of security bug reports by text-based prediction models. Our approach is based on the observation that it is the presence of security related keywords in both security and non-security bug reports, which leads to mislabelling. Based on this observation we developed a method for automatic identification of these keywords and for scoring bug reports according to how likely they are to be labelled as SBRs. The 'marriage' of filtering and ranking presents an effective and usable solution for reducing the mislabeling of SBRs in both WPP and TPP models.

There are a number of directions in which this work can be extended. In future work, we plan to:

- incorporate additional information contained in the bug reports such as who were assigned bug reports, time stamps showing how much time it took to fix bug reports as well as comments and feedback on bug reports;
- identify the context of security keywords in SBRs as a post processing step to see if they can also improve on the ranking capability in FARSEC;
- find an automatic way to determine the threshold for the removal of NSBRs with high scores (this threshold could vary with different projects);
- find an automatic way to determine the best number of security related keywords to use when creating data matrices for each project;

Acknowledgment

The authors would like to thank Keerthi Thomas, Arosha Bandara and Blaine Price for their initial contributions to this work. We would also like to thank the reviewers for their insights on the previous version of this paper. This work was funded supported, in part, by Science Foundation Ireland grant 13/RC/2094 and by the European Research Council (Advanced Grant 291652 - ASAP).

References

[1] I. Chawla and S. Singh, "Automatic bug labeling using semantic information from LSI," in *Seventh International Conference on Contemporary Computing (IC3)*, Aug 2014, pp. 376–381.

[2] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: Learning to classify vulnerabilities and predict exploits," in *16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '10. New York, NY, USA: ACM, 2010, pp. 105–114.

[3] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *7th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2010, pp. 11–20.

[4] D. Wijayasekara, M. Manic, J. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *5th International Conference on Human System Interactions (HSI)*, June 2012, pp. 89–96.

[5] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Third International Conference on Software Testing, Verification and Validation (ICST)*, April 2010, pp. 421–428.

[6] 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, Nov 2011.

[7] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[8] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[9] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2014, pp. 23–33.

[10] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:9.

[11] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, Sept 2009.

[12] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on github," in *11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 348–351.

[13] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of software maintenance activities," in *8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 163–172.

[14] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, Oct 2010.

[15] 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 *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595713

[16] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, pp. 540–578, 2009.

[17] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012, pp. 61:1–61:11.

[18] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248 – 256, 2012.

[19] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, pp. 167–199, 2012.

[20] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 822–834, June 2013.

[21] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *International Conference on Software Engineering (ICSE)*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 382–391.

[22] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013, pp. 45–54.

[23] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with morph," in *International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 189–199.

[24] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1054–1068, Aug 2013.

[25] R. Feldman and J. Sanger, *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2007.

[26] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, ser. Prentice Hall series in artificial intelligence. Pearson Prentice Hall, 2009.

[27] C. D. Manning, P. Raghavan, and H. Schtze, *Scoring, term weighting, and the vector space model*. Cambridge University Press, 2008, p. 100123.

[28] E. Rochester, *Clojure Data Analysis Cookbook - Second Edition*, ser. EBL-Schweitzer. Packt Publishing, 2015.

[29] D. Heckerman, E. Horvitz, M. Sahami, and S. Dumais, "A bayesian approach to filtering junk e-mail," in *Proceeding of AAAI-98 Workshop on Learning for Text Categorization*, 1998, pp. 55–62.

[30] P. Graham, *Hackers & Painters: Big Ideas from the Computer Age*. O'Reilly Media, 2004.

[31] O. Jalali, T. Menzies, and M. Feather, "Optimizing requirements decisions with keys," in *4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '08. New York, NY, USA: ACM, 2008, pp. 79–86.

[32] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[33] J. R. Quinlan, "Bagging, boosting, and c4.5," in *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*. AAAI Press, 1996, pp. 725–730.

[34] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, pp. 321–, Jul. 1961.

[35] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 518–521.

[36] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *International Conference on Software Engineering (ICSE)*, May 2011, pp. 481–490.

[37] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485 –496, july-aug. 2008.

[38] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces," in *37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 199–208.

[39] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[40] H. Zhang, "The optimality of naive bayes," in *Proceedings of the Seventeenth Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2004, pp. 562–56.

[41] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2 –13, jan. 2007.

[42] W. Afzal, "Using faults-slip-through metric as a predictor of fault-proneness," in *Asia Pacific Software Engineering Conference*, ser. APSEC '10, 2010, pp. 414–422.

[43] E. Weyuker, T. Ostrand, and R. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, pp. 539–559, 2008.

[44] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[45] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21 –27, january 1967.

[46] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[47] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[48] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[49] W. Su, Y. Yuan, and M. Zhu, "A relationship between the average precision and the area under the roc curve," in *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ser. ICTIR '15. New York, NY, USA: ACM, 2015, pp. 349–352.

[50] D. Martinus and J. Tax, "One-class classification: Concept-learning in the absence of counterexamples," Ph.D. dissertation, PhD thesis, Delft University of Technology, 2001.