Automatic Identification of Security Bug Reports via Semi-Supervised Learning and CVE Mining

Shaikh Mostafa and Xiaoyin Wang
Department of Computer Science, University of Texas at San Antonio,
One UTSA Circle, San Antonio, TX 78249, USA
Email: {Shaikh.Mostafa, Xiaoyin.Wang}@utsa.edu

***Abstract*—Bugs are unavoidable in almost any software system, and open bug repositories have been widely used to collect bug reports from different users around the world. Among all categories of bug reports, security bug reports are of special interest because they usually require earlier fixes and should be hidden from public to avoid potential attacks before a corresponding patch is released and distributed. Due to the large amount of bug reports, and limited number of bug triagers, it is usually hard for bug triagers to manually identify security bug reports in time. Therefore, an automatic approach to identifying security bug reports and provide candidate security bug reports to the triagers with higher priority will help developers to fix them sooner as well as reduce their exposure time to the public. In this paper, we propose an automatic approach to identify security bug reports in open bug repositories. Specifically, to address the challenges of lacking labeled security bug reports and the data imbalance (security bug reports are typically a very small portion of all bug reports), our approach leverages semi-supervised learning, and keyword-based pre-filtering based on keywords mined from existing security-related textual descriptions in the CVE (Common Vulnerability and Exposures) database. To evaluate our approach, we constructed two data sets of security bug reports from the open bug repositories of RedHat and Mozilla. Then, we applied our approach to both datasets, and compared our results with more than 10 variants of two categories of baseline approaches. The experimental results show that our approach achieves F-scores of 86.8% and 64.9%, which outperforms the best baseline approaches by 7.4 percentage points and 9.6 percentage points, respectively.**

I. Introduction

Nowadays, bug repositories are widely used in software projects to collect bugs from users as well as to track the fixing process of bugs. A common difficulty in the management of current bug repositories is that, a small number of bug triagers (i.e. the people who filter out invalid bugs and assign bugs to developers) have to process a huge number of bugs reports received everyday. As a result, it is common that a bug report is not read and taken care of by anybody for several days or even longer time [8].

Among all types of bug reports, security bug reports especially need to be handled timely for the following two reasons. First of all, compared with other bugs, security bugs are more likely to cause severe consequences such as money loss or privacy leak. Therefore, they should be fixed earlier to avoid such potential consequences. Second, in open bug repositories where bug reports are available to the public, a security bug report may be accessed by potential attackers to the software. With the information described in the bug report, it is probable that the attacker is able to design a successful attack to exploit the security bug. Actually, both researchers and developers have been aware of such risks [10], so that in some open bug repositories, once a bug report is identified as related to security, it is hidden from the public for a long time until the bug is fixed in a release and distributed to the users.

To resolve the conflict between the timeliness requirement of handling security bug reports and the heavy workload of bug triagers, an automatic approach to identify security bug reports is desired so that a smaller number of candidate security bug reports can be provided to bug triagers with higher priority. After that, the security bug report can be labeled separately to avoid public access and be assigned to a proper developer. Note that, hiding all bug reports from public before they are triaged is typically not a desired solution, because (1) it does not accelerate the processing of security bugs, and (2) it undermines the advantage of open bug repositories (e.g., not allowing users to search pre-triaged bug reports for duplicate detection, patching, and providing extra information).

In general, the identification of security bug reports can be viewed as a classification problem that aims to classify a bug report to either the category of security bug reports or the category of non-security bug reports. However, in the specific problem of security-bug-report identification, there are two major practical challenges. The first challenge is that, there are very few labeled security bug reports in the wild, and labeling existing bug reports is time-consuming and requires software and security expertise (so that can hardly be helped by crowd-sourcing such as MechnicalTurk [4]). Therefore, it is very hard to acquire a large labeled training set for either training classification models or mining for proper keywords. The second challenge is that, security bug reports are much fewer than non-security bug reports, which results in a very unbalanced data distribution. It has been well-known that unbalanced data distribution has a large negative impact on the effectiveness of classification techniques [11] [20].

In this paper, we propose an automatic approach to identifying security bug reports. To reduce the negative impact of the two challenges mentioned above, we design our approach with the following techniques.

- To make better use of small training sets, we leverage semi-supervised learning techniques [32] as our general solution. Unlike traditional supervised learning which

trains a classification model from a labeled training set, semi-supervised learning trains a classification model from both labeled and unlabeled data.

- To handle the unbalanced data distribution, we perform a keyword-based pre-filtering process to rule out all the bug reports that do not contain any security-related keywords. After the pre-filtering, the set of left bug reports becomes much more balanced to be classified.
- To acquire a complete list of security-related keywords, we still need to mine from a large number of labeled security bug reports which is not available. In our approach, we leverage the textual descriptions in the CVE (Common Vulnerability and Exposures) [2] database which describe various security problems detected in various software projects.

To evaluate our approach, we constructed two data sets from the open bug repository of RedHat [6] and Mozilla [5]. We are able to extract labeled security bug reports from RedHat and Mozilla because RedHat has a security response team to handle security bug reports, and Mozilla has a number of security alert announcements which can be traced back to security bug reports. In the experiments, we compared our approach with classification based approaches and information-retrieval-based approaches. The experimental results show that, our approach outperforms the best baseline approach by 7.4 and 9.6 percentage points, in Redhat and Mozilla, respectively.

The main contributions of this paper are listed as follows.

- An automatic and effective approach based on semi-supervised learning and keyword mining to identify security bug reports in open bug repositories.
- A publicly available data set with labeled security bug reports for future research on the topic.
- An experimental evaluation to compare our proposed approach with classification based approaches and information-retrieval based approaches, as well between the variants of our approach.

II. Preliminary Study

In Section I, we explain that security bug reports should be identified immediately after their submission, so that they can be hidden form the public and get higher priority for fix. Therefore, it is interesting to know what is the current time gap between the submission of security bug reports and the assignment of the bug report by a bug triager in the software repository.

In 2010, Zaman et al. carried out an empirical study [31] on security bug reports of Firefox, and they found that the medium time gap between the submission of a security bug report and the assignment of the security bug report to a developer is about 4,000 minutes (66 hours) [1]. It is also reported that, this time gap is shorter than that of non-security bug reports.

To motive our research, we replicated this study on the two data set we constructed from RedHat and Mozilla. We also measured the time gap between the submission of a security bug report to the first assignment of the bug report to a developer. Our study shows that the medium triaging times of security bug reports in our RedHat data set and our Mozilla data set are 1820 minutes, and 2637 minutes, respectively. The accumulative distribution of security bugs on response time is presented in Figure 1.

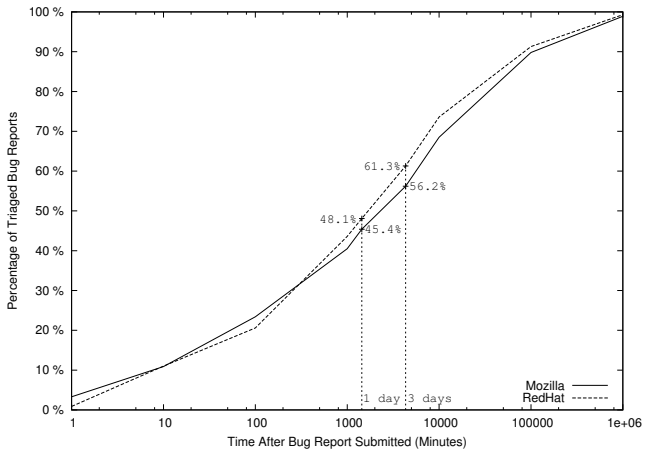

Fig. 1. Accumulative Distribution of Security Bug Reports on Response Time

From the figure, we can see that, 55% of security bug reports in Mozilla and 52% of security bug reports in RedHat are not triaged 24 hours after their submission; and 44% of security bug reports in Mozilla and 38% of security bug reports in RedHat are not triaged 3 days after their submission. Such a time gap provides enough time for potential attackers to notice and download the security bug reports, and to find out approaches to perform potential attacks later.

Therefore, with an automatic identification tool of security bug reports, it is possible to handle and hide the candidate security bug reports sooner after they are submitted, which will largely reduce the risk of exposing them to the attackers. Actually, it is possible to have all the bug reports withheld and make only non-security bug reports public after the triaging. However, as we mentioned in Section I, such a process will harm the participation of users and other potential bug fixers, and reduce the benefit of having open bug repositories. Existing open bug repositories that choose to hide security bug reports, such as Mozilla, also do not leverage such process.

III. Background

Semi-supervised learning [32] is a category of machine learning techniques that make use of not only labeled data, but also unlabeled data for training. In particular, researchers in machine learning have discovered that, unlabeled data, when used in conjunction with a small amount of labeled data, can produce considerable improvement in learning accuracy [7]. As more and more data on the Internet becomes available for training purposes every year, it is more and more difficult for data labeling, typically a manual process, to keep up with the speed of data growth. As a result, semi-supervised learning is attracting extensive attention and interest from both academia and industry [16]. Self-training [26] is a widely-used semi-supervised learning algorithm. In particular, it leverages the initial small labeled data set as the initial training set and

[1]Unfortunately, a large portion of the bug reports used in the study are hidden from public now.

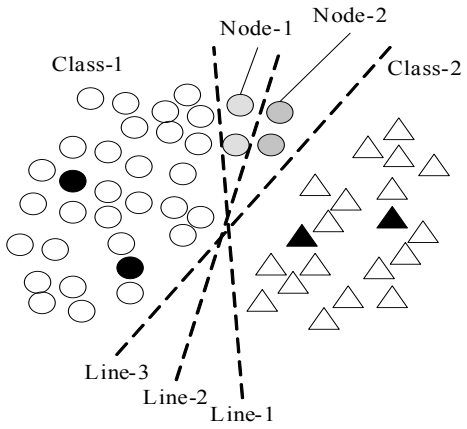

Fig. 2. Illustration of Semi-Supervised Learning

trains an initial classification model. After that, it iteratively uses the classification model to classify the unlabeled data, adds the classification results to the training set, and uses the expanded training set to train a new classification model. As more and more data on the Internet becomes available for training purposes every year, it is more and more difficult for data labeling, typically a manual process, to keep up with the speed of data growth. As a result, semi-supervised learning is attracting extensive attention and interest from both academia and industry [16]. Self-training [26] is a widely-used semi-supervised learning algorithm. In particular, it leverages the initial small labeled data set as the initial training set and trains an initial classification model. After that, it iteratively uses the classification model to classify the unlabeled data, adds the classification results to the training set, and uses the expanded training set to train a new classification model.

Figure 2 illustrates the problem of semi-supervised learning and the self training algorithm. In the figure, the instances of class-1 are marked with round nodes in the feature space, and the instances of class-2 are marked with triangle nodes. However, only a very small portion of the nodes are labeled, represented by the two solid round nodes and the two solid triangle nodes. Without the extra unlabeled instances, a learning algorithm (e.g., SVM) will learn a model based on just the labeled instances. The model is illustrated as Line-1 in the figure. However, with the extra unlabeled instances, we can see that Line-3 may be the best separation of the two classes, because it best separates two clusters with high cohesion (Note that all hollow and shaded nodes are unlabeled).

With the self training algorithm, the first iteration of training is based on the labeled data, so Line-1 in Figure 2 is generated as a classification model. Then, all nodes to the left of Line-1 are labeled as Class-1, and all nodes to the right (i.e., all triangle nodes and the 4 shaded round nodes) of Line-1 are labeled as Class-2. In the second iteration of training, a classification model is generated based on the nodes labeled in the first iteration. For the nodes between Line-1 and Line-2 (e.g., Node-1), they have more and closer Class-1 neighbors than Class-2 neighbors, so with classification models such as K-Nearest-Neighbor [15] or SVM [21] (can be viewed as continuously weighted K-Nearest-Neighbor), they will be relabeled as Class-1, and Line-2 will be generated as a new classification model. Similarly, after more iterations, Line-3 is finally reached and the algorithm converges. It should be noted that we use the SVM classification model as an example for the ease of illustration on a feature space. The idea of self-training can be applied to any learning algorithm.

In the problem of identifying security bug reports, a large number of bug reports are available from open bug repositories, but labeling security bug reports is a very time-consuming task requiring much expertise. Therefore, we believe that the accuracy of security-bug-report identification can be enhanced with semi-supervised learning. Specifically, in our approach, we choose to use the self-training algorithm, because it makes no presumptions to the distribution of data, and can be used with any existing supervised learning techniques.

Common Vulnerabilities and Exposures (CVE) is an open database that contains a large set of publicly known information-security vulnerabilities and exposures, and aims to provide a unified reference method for these vulnerabilities and exposures. In particular, when developers, security analysts, or software vendors find a vulnerability and wants to publicize and record it, they can submit an entry to the CVE database. After that, the submitted entry will be manually investigated to decide whether it should be added as a new entry, merged to an existing entry, or simply ignored. Until April 2015, there have been more than 60,000 vulnerabilities and exposures recorded in the CVE database. A CVE entry typically contains a unique ID, revision history, a short textual description of the problem, and references. As an example, in CVE-2014-9311, the textual description states that "A cross-site-scripting (XSS) vulnerability in admin.php in the Shareaholic plugin before 7.6.1.0 for WordPress allows remote authenticated users to inject arbitrary web script for HTML via ...". Note that this statement contains quite many security related keywords such as "XSS", "vulnerability", "authenticated", "inject", etc.

Figure 3 shows an example of a CVE entry. A CVE entry typically contains a unique ID, date when the entry is released and last revised, a short textual description of the problem, scores to measure the severity of the problem, ways of exploitation, and references. The reference item usually contains a list of URL targeting at relevant information of the entry, such as the original bug report in the software vendor or organization that submit the entry.

At the same time, it should be noted that the CVE-entry descriptions are summarized by professional security analysts instead of software users who submit the majority of security bug reports, so that the language style and terms of the descriptions may be different from those of security bug reports to some extent. Based on the above observation, in our approach, we use descriptions of CVE entries as a supplement data set of the small training set to mine the initial list and weights of security keywords, and use the mined keyword list in the beginning steps of semi-supervised learning.

IV. Approach

In this section, we introduce our approach in details, including how we apply self-training, how we mine the keyword list for the pre-filtering, and how we incorporate the keyword mining and pre-filtering into the process of self-training.

Vulnerability Summary for CVE-2014-9311

Original release date: 04/14/2015

Last revised: 04/15/2015

Source: US-CERT/NIST

Overview

Cross-site scripting (XSS) vulnerability in admin.php in the Shareaholic plugin before 7.6.1.0 for WordPress allows remote authenticated users to inject arbitrary web script or HTML via the location[id] parameter in a shareaholic_add_location action to wp-admin/admin-ajax.php.

Impact

CVSS Severity (version 2.0):

CVSS v2 Base Score: 3.5 (LOW) (AV:N/AC:M/Au:S/C:N/I:P/A:N) (legend)

Impact Subscore: 2.9

Exploitability Subscore: 6.8

CVSS Version 2 Metrics:

Access Vector: Network exploitable; Victim must voluntarily interact with attack mechanism

Access Complexity: Medium

Authentication: Required to exploit

Impact Type: Allows unauthorized modification

References to Advisories, Solutions, and Tools

External Source: MISC

Name: http://security.szurek.pl/shareaholic-7603-xss.html

Hyperlink: http://security.szurek.pl/shareaholic-7603-xss.html

Fig. 3. An Example CVE Entry

A. Application of Self-Training

The basic idea of our approach is to apply self-training to the identification of security bug reports. The application is straightforward. First, we use supervised learning to train a classification model from the small training set (with manually labeled tag for security and non-security). Second, with the classification model, we label all bug reports in the testing set as either security or non-security. Third, we add automatically labeled bug reports to the training set. We perform the steps 1-3 for a fixed number of iterations or until converge (i.e., no bug report in the testing set changes its label after a new iteration). The process is actually presented in the part of Algorithm 1 that are not underlined.

B. Keyword Mining

To solve the problem of unbalance data, we propose to perform keyword-based pre-filtering to exclude the bug reports that are not security related at all, so that the dataset will become more balanced. Therefore, we need to acquire a list of keywords that are most representative for security bug reports.

Such a list of keywords can be mined from the training set. However, in our scenario, the training set typically contains only a small number of security bug reports, and thus these bug reports may not cover sufficient variety of keywords. Therefore, we leverage the entries in the CVE database as the supplementary security bug reports for mining the list of keywords. In particular, we use the descriptions in the entries of the CVE database together with the security bug reports in the training set as the *security document set*, and use the non-security bug reports in the training set[2] as the *non-security document set*.

To measure how representative a term is for a document set, we leverage the adaption of standard tf-idf weights [25]. Tf-idf is a standard term weighting approach in information retrieval to measure how representative a word is for a document. Specifically, $tf(t, doc)$ is defined as the frequency of the term t in the document doc, and $idf(t)$ is defined with the formula below. The idf value measures whether t is unique enough to represent the document. The product of $tf(t, doc)$ and $idf(t)$ is viewed as the representativeness measure of term t for document doc.

$$idf(t) = \log(\frac{\text{\#all documents}}{\text{\#documents containing t}}) \quad (1)$$

Therefore, an adaptive version of tf-idf for a category of documents, is defined as below. The basic idea is to view all documents in category cat as one large document. Thus, in the formula below, for $tf(t, cat)$, the total frequency of term t in all documents in category cat is used. For $idf(t)$, the total number of documents is replaced with the number of documents not in category cat plus one (representing the one large document corresponding to the category cat), and the total number of documents containing term t is replaced with the number of documents that contain term t and are not in category cat. Similarly, 1 is added for the one large document corresponding to the category cat. The adapted formula for calculating $tf(t, cat)$ and $idf(t)$ values are presented in formulas below.

$$tf(t, cat) = \text{total frequency of } t \text{ in all documents in } cat \quad (2)$$

$$idf(t) = \log(\frac{1 + \text{\#documents not in } cat}{1 + \text{\#documents not in } cat \text{ containing } t}) \quad (3)$$

With the adapted tf and idf value, we use the product of the two values to measure the representativeness of a word for a category (denoted as $w(t, cat)$). In our approach, we want the selected top words to differentiate security bug reports and non-security bug reports. So we further define the *discriminative weight* of term t as the quotient of $w(t, security)$ and $w(t, nonsecurity)$. In comparison, we refer to $w(t, security)$ as the *non-discriminative weight* of term t.

Based on the weights calculated, we are able to rank all terms that appears in security bug reports (in the training set), and have the top ranked terms in the $toptermlist$ as the keyword list for the balancing of the datasets.

C. Incorporation of Keyword Mining in Self-Training

As we mentioned in Section I, we use self-training to address the challenge of small training sets, and use keyword-based pre-filtering to address the challenge of unbalanced

[2]The number of non-security bug reports in the training set is typically much larger than that of security bug reports, due to unbalanced data distribution.

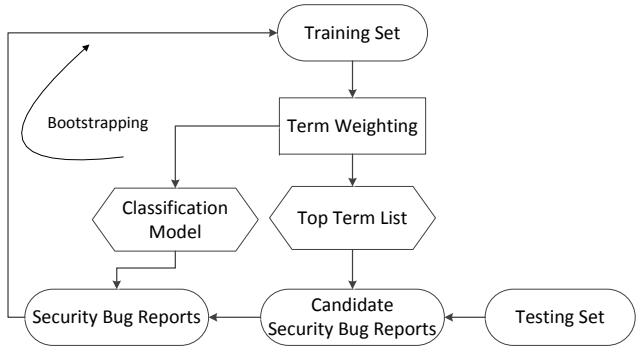

Fig. 4. Overview of our approach

datasets. One major technical challenge in our approach is how to combine the two phases. To generate a balanced data set for the classification process, the pre-filtering must be performed before the self-training phase. However, with a very small training set and the non-project-specific CVE entries, it is difficult to mine high-quality project-specific keyword list and word weights for effective pre-filtering.

In our approach, to address the above challenge and take full advantage of the two techniques, we proposed a solution that combines the two techniques seamlessly by adding a keyword-mining step and a pre-filtering step into each iteration in the semi-supervised learning process. Specifically, our revision to the basic self training algorithm is presented Algorithm 1.

Algorithm 1: Incorporation of Keyword-based Pre-filtering in Self-Training

```
   Input: Training Set R, Testing Set S, CVE Entries E
   Output: Labeled Dataset L
1  keywords = rank(R ∪ E);
2  L = R ∪ S;
3  while not converge do
4      L' = filter(L, keywords);
5      if first round then
6          model = train(R);
7      else
8          model = train(L');
9      end
10     predict(L' ∩ S, model);
11     set label as Non-Security for all in (L-L') ∩ S;
12     keywords = rank(L);
13 end
```

In Algorithm 1, the inputs are a training set R, a testing set S, and a CVE entry set E, while the output is the labeled dataset L. The lines that are not underlined (i.e., Lines 2, 3, 5-10) are the basic self-training algorithm, which continuously update the labels of the testing set (Line 10), and retrain the classification model with the training set and the relabeled testing set (Lines 5-9), until the algorithm converge. For the incorporation of keyword-based pre-filtering, we added four underlined lines (i.e., Lines 1, 4, 11, and 12).

First, at the beginning of the algorithm (Line 1), we generate an initial keyword list from the union of the training set and the CVE entry set based on the approach depicted in Section IV-B. Second, at the beginning of each iteration in self-training (Line 4), we add the pre-filtering process to filter out the relatively obvious non-security bug reports (does not contain any keywords) so that the whole dataset is more balanced. It should be noted that we do not filter out any security bug reports in the original training set. Third, since we pre-filtered the relatively obvious non-security bug reports, we need to label them as "Non-Security" during the relabel process, so we add Line 11 after the original relabel statement (Line 10). Fourth, at the end of each iteration in self-training, we redo the term-weighting and ranking, and generate a refined keyword list with the newly labeled dataset (Line 12).

With the above adaptation of the self-training process, we can take advantage of the keyword-based pre-filtering in each iteration of the self-training process, and take advantage of the refined classification results to further enhance the quality of the keyword list for pre-filtering.

V. Evaluation

To evaluate our approach, we carried out an experiment based on a data set from the RedHat bug repository [6] and the Mozilla bug repository, both hosted at Bugzilla [1]. In this section, we first introduce how we build our data set and the metrics we used in our evaluation. Then, we present the overall experimental results of our approach and comparison between our approach and existing approaches. After that, we evaluate the effectiveness of the techniques in our approach by turning off each of them. Moreover, we introduce the variants of our approach (when different parameters are used), and compare the effectiveness between them.

A. Evaluation Setup

In our evaluation, we constructed two data sets[3] from RedHat and Mozilla bug repository, respectively.

Identification of Security Bug Reports. Open bug repositories typically do not have special tags for security bug reports. However, in RedHat and Mozilla, we found information to identify security bug reports. In the RedHat repository, there is a developer team called "Security Response Team" who handles security bugs. We randomly inspect the bug reports that are handled and confirmed by the team, and finally selected 800 bug reports that we confirm to be security bug reports. In Mozilla, vulnerabilities are periodically announced to the public[4], and the vulnerability announcements contain a link to the original bug reports[5]. Therefore, we constructed the

[3]The data sets are available at http://xywang.100871.net/secbug.html

[4]https://www.mozilla.org/en-US/security/known-vulnerabilities/

[5]Only relatively old (more than 3-4 years) vulnerability announcements contain the link, while the recent ones do not (Maybe Mozilla wants to hide them in case of incomplete fix).

dataset (568 security bug reports in total) for Mozilla by tracing the links from the vulnerability announcements. Note that, one vulnerability announcement may link to multiple security bug reports. In our experiment, we deem the multiple security bug reports traced from one vulnerability announcement as separate bug reports, no matter whether they are duplicate with each other or not, because the triager typically should not know whether they are duplicate before triaging. In the experiment, we use only the summary and / or description information of the bug reports when we apply various techniques, because they are for sure to be available before a bug report is triaged.

Ratio between Security and Non-Security Bug Reports After the identification of security bug reports, we need to further decide how many non-security bug reports to be put into the data set. In RedHat, our statistics show that suspected security bug reports (i.e., any bug reports handled by the "Security Response Team") accounts for about 8% of all bugs in the time range where our security bug reports are created (i.e., 2010 to 2012). Therefore, the ratio of security bug reports to non-security bug reports is 1:11. For Mozilla, it is much more difficult to estimate a precise ratio because many security bug reports are not publicly available for the reasons mentioned above, and vulnerability announcements cover only a small portion of security bug reports (i.e., only severe ones that require users to update their software as soon as possible). Therefore, we use the same ratio 1:11 for Mozilla to provide a basis for comparison. It should be noted that different ratios may affect the effectiveness of our approach, and different software projects may have different proportion of security bug reports. We plan to further study this issue in the future.

Identification of Non-Security Bug Reports. Another challenge in our evaluation setup is how to determine that a bug report is a non-security bug report. According to the ratio of 1:11, we need 8800 non-security bug reports for Redhat, and 6000 non-security bug reports for Mozilla. It should be noted we are not able to manually inspect all these bug reports to confirm that they are true non-security bug reports. Therefore, we first randomly selected the required number of bug reports from the bug repository. Then, from the selected bug reports, we further extracted potential security bug reports (i.e., any bug reports linked to CVE, related to the "Security Response Team", or containing any security keywords in our initially mined security-word list from CVE). Then, we manually studied these potential security bug reports (by reading the descriptions and comments) to see whether they are real security bug reports (definition of security bug reports discussed in Section VI-A). If a bug report is a real security bug report from our perspective (we can not confirm because we are not developers), we remove it from our dataset, and add other randomly selected bug report to our dataset. We recursively check newly added bug reports until we collected sufficient non-security bug reports. It should be noted that we did not simply remove all potential security bug reports, otherwise the problem would be simplified due to manually removed non-security bug reports that share keywords with security bugs.

Cross-Validation. As we mentioned in Section I, a major challenge of automatic security-bug-report identification is the lack of training set. Therefore, to simulate the real-world scenario, we also evaluate our approach with a small training set. Specifically, we use a training set with about 100 security bug reports, because we believe that a training set of such size is typically the minimal requirement for machine learning [30] [33], and can be build with reasonable manual effort (i.e., manually inspection of several hundred potential security bug reports). In our evaluation, we use cross validation [23] to make sure our evaluation results in not specific to a certain training set.

Exclusion of CVE Database Entries. We noticed that, some security bug reports in our dataset are related to CVE entries. If we consider these CVE entries when mining the initial keyword list, some knowledge about the test set may be involved in the keyword list, and result in biased evaluation. Therefore, when mining CVE database, we exclude any CVE entries that have references to Mozilla or Redhat bug reports.

Measurements. To measure the effectiveness of our approach, we leverage the standard precision, recall and F-score metrics. Specifically, precision is defined as the number of real security bug reports identified as security bug reports divided by the number of all bug reports identified as security bug reports. Recall is defined as the number of real security bug reports identified as security bug reports divided by all real security bug reports. F-score is the harmonic mean of precision and recall.

B. Compared Approaches and Parameters

In this subsection, we introduce the approaches and variants that we compared in our evaluation.

1) Baseline Approaches.: In our experimental study, we consider two baseline approaches as follows to compare our approach with. The first baseline approach is supervised learning (denoted as **BS1**), and we consider two classification models: Naïve Bayes[6] and SVM[7]. For supervised learning, we consider three variants presented as below. Specifically, besides the basic supervised learning approach, we also considered oversampling [13], a widely used existing technique for handling unbalanced data, and directly using CVE reports as security bug reports in the training data.

BS1-1: In **BS1-1**, we perform supervised learning based on our standard training set (e.g., 100 / 1100 security / non-security bugs for Redhat, and 110 / 1200 security / non-security bugs for Mozilla).

BS1-2: In **BS1-2**, we perform oversampling to copy the security bug reports in our standard training set for 11 times and thus we can get a balanced training set with about the same number of security / non-security bug reports. Then, we perform supervised learning with this balanced training set.

BS1-3: In **BS1-3**, we directly use CVE reports as security bug reports. Since there are plenty of CVE reports, we should not stick to the small size of training set. Therefore, we revert our training / testing set to use the majority of our data set as the training set. For RedHat, we used 7700 non-security bug reports and 7700 randomly selected CVE reports as training set, and the rest 1100 non-security bug reports and 100 security

[6]We used the implementation of Naïve Bayes in Weka [3].

[7]We use liblinear [17] as the implementation of SVM.

TABLE I. CONSIDERED PARAMETERS

Parameter	Possible Values
Top-Term-List Size	50, 100, 200
Term-Weighting	Discriminative, Non-Discriminative
Classification	Naive Bayes, SVM
Information Source	title only, title + description

bug reports as testing set. For Mozilla, we used 4800 non-security bug reports and 4800 randomly selected CVE reports as training set, and the rest 1200 non-security bug reports and 110 security bug reports as testing set. Note that, with cross-validation, no matter how we split our data set, the whole data set is being tested during the evaluation.

The second baseline approach (denoted as **BS2**) is retrieval of security bug reports based on a keyword list mined from the training set (i.e., keyword-based filtering). For keyword-based filtering, we consider two parameters: the size of top term list, and whether using discriminative or non-discriminative term weighting. In particular, we use 50, 100, and 200 as variables of top-term-list size, and consider all six combinations of top-term-list sizes and term weighting techniques. The evaluation results are presented in Section V-C1

2) Variants of Our Approach: **Evaluation of Proposed Techniques.** In our approach, we use self-training to address the problem or small training set, and involve keyword-based pre-filtering to address the problem of unbalanced dataset. Also, we propose to involve CVE database entries when mining for the initial keyword list. To evaluate the effectiveness of these techniques, we compare our approach with the variants of our approach that turns off self-training (using just supervised learning), keyword-based pre-filtering, and CVE-database mining. The results are in Section V-C2.

Impact of Parameters. Our approach involves in a number of parameters and thus has a large number variants when combining the parameters. In this evaluation, we consider the parameters shown in Table I. The parameters include: whether use title only or title plus description of the bug reports, using what size for top term list, using discriminative term weighting or non-discriminative term weighting, and what machine learning model to use in classification(i.e., we specific consider the Naive Bayes and SVM which are widely used and based on different theocratic foundations). The default setting of our approach uses title plus description of bug reports, 100 top term list size, discriminative term weighting, and SVM model. We present the comparison between different variants of our approach in Section V-C3.

C. Evaluation Results

1) Comparison with Baseline Approaches: The comparison of our approach (default setting) with supervised-learning-based approach (**BS1**) is shown in Table II. To save space, in all tables, we abbreviate "Naïve Bayes" as "NB", "Precision" as "P", "Recall" as "R", and "F-Score" as "F", respectively. From the table, we have the following observations.

First, our approach outperforms all variants of **BS1** on F-score. Specifically, in Redhat, our approach outperforms basic supervised learning by at least 8.1 percentage points, and the best **BS1** variant (i.e., **BS1-2** with SVM) by 7.4 percentage points. In Mozilla, our approach outperforms basic supervised learning by at least 8.2 percentage points, and the **BS1** variant (i.e., **BS1-2** with SVM) by 5.7 percentage points.

TABLE II. COMPARISON WITH SUPERVISED LEARNING

Subject	Approach	P (%)	R (%)	F (%)
Redhat	Our Approach	90.4	83.5	86.8
	NB (BS1-1)	69.5	81.0	71.0
	SVM (BS1-1)	96.0	67.1	78.7
	NB (BS1-2)	22.3	96.0	36.2
	SVM (BS1-2)	83.0	76.1	79.4
	NB (BS1-3)	22.0	27.0	24.2
	SVM (BS1-3)	68.1	28.0	39.7
Mozilla	Our Approach	80.8	48.9	61.0
	NB (BS1-1)	43.7	68.0	52.8
	SVM (BS1-1)	73.5	39.2	51.1
	NB (BS1-2)	35.5	72.8	47.7
	SVM (BS1-2)	63.0	49.3	55.3
	NB (BS1-3)	6.7	4.8	5.6
	SVM (BS1-3)	7.7	2.5	3.8

Second, among all variants of **BS1**, **BS1-2** with SVM performs best on both projects, which shows that oversampling is effective to some extent on handling unbalanced data. Also, **BS1-3** performs worst in both projects, which shows that, raw CVE reports alone are not sufficient to serve as training data, while proper processing of CVE reports and a small number of project-specific security bug reports are very important.

Third, most of the techniques achieve better results on Redhat than Mozilla. With some inspections, we find that the security bug reports in Redhat have more commonalities, while the Mozilla dataset covers larger variety of security bugs. The underlying reason may be that Mozilla has more products and the bug reporters of Mozilla are less professional.

Therefore Naïve Bayes performs better on Mozilla because it has less overfitting effect on the training set. This may also explain why all approaches perform better on Redhat than Mozilla.

It should be noted that for the scenario of security bugs, recall is often more important when the precision is not very low. Compared with Naive Bayes, our approach is 13.7 percentage points higher on F-score, while 2.6 percentage points lower on recall. Actually, both the F-Scocre and recall of our approach can be further enhanced by adding bootstrapping (See Section **??**).

The Lines 2-4 in the table are the results of our approach (with default setting), the results of the Naive Bayes classifier, and the results of the SVM classifier, respectively.

The comparison of our approach (default setting) with keyword-mining-based approach (**BS2**) is shown in Table III. In the table, the rows 2 and 9 presents the results of our approach as a reference. Rows 3-5 and 10-12 present the results of using only keyword-based filtering, with different keyword-list sizes (i.e., 50, 100, and 200). Rows 6-8 and 13-15 present the results of keyword-based filtering based on non-discriminative term weighting, also combined with different top-term-list sizes. From the table, we observe that, our approach outperforms all filtering based approaches on F-score by at least 12 percentage points. Specifically, keyword-based filtering with non-discriminative term weighting results in very high recall and very low precision. By contrast, keyword-based filtering with discriminative term weighting results in higher precisions, but the precision drops sharply when a larger top-

TABLE III. COMPARISON WITH KEYWORD-BASED FILTERING

Subject	Approach	P (%)	R (%)	F (%)
Redhat	Our Approach	90.4	83.5	86.8
	Disc. + 50	75.4	74.8	74.8
	Disc. + 100	65.1	83.3	72.8
	Disc. + 200	53.0	89.0	66.1
	Non-D. + 50	12.1	98.2	21.5
	Non-D. + 100	9.5	99.4	17.4
	Non-D. + 200	8.9	99.8	16.4
Mozilla	Our Approach	80.8	48.9	61.0
	Disc. + 50	24.2	49.1	32.4
	Disc. + 100	18.6	56.6	28.4
	Disc. + 200	14.1	63.3	24.0
	Non-D. + 50	14.3	87.6	24.7
	Non-D. + 100	7.7	98.1	14.3
	Non-D. + 200	7.2	99.0	13.4

TABLE IV. EVALUATION ON DIFFERENT ROUNDS OF SELF-TRAINING

Subject	Iteration	P (%)	R (%)	F (%)
Redhat	Round 0	94.6	71.0	80.8
	Round 1	92.1	80.4	85.5
	Round 2	90.9	82.6	86.2
	Round 3	90.6	83.4	86.5
	Round 4	90.4	83.5	86.8
Mozilla	Round 0	83.7	39.0	53.2
	Round 1	81.4	47.0	59.6
	Round 2	81.8	48.2	60.7
	Round 3	80.8	48.9	61.0
	Round 4	80.8	48.9	61.0

term-list size (i.e., 200) are used to achieve higher recall.

2) Evaluation of Techniques: In this subsection, we evaluate our approach on bootstrapping. Specifically, we perform our approach for 1-4 more rounds, using the results of the previous round as the training set of the next round. We perform up to 4 iterations because we found that the results tend to be stable after that. Since the variants of our approach with different top-term-list sizes all achieve good results (F score higher than 80%), we perform experiments with all different sizes of top-term-list sizes. The results are presented in Table **??**, Table **??**, and Table **??**.

The results show that self-training is able to provide significant improvement on the F-score. Specifically, 4 iterations is able to improve the F-score of our approach by 6.0 percentage points in Redhat, and 7.8 percentage points in Mozilla. Another observation is that, the first iteration in self-training contributes the most improvement,which indicates that, with limited resource, fewer iterations of self-training are still helpful.

To evaluate the effectiveness of keyword-based pre-filtering, and CVE database mining, we compare our approach with variant (1): turning off the whole keyword-based pre-filtering process, and variant (2): exclude CVE entries when mining the initial keyword list (i.e., using only the training set). The results are shown in Table V, in which Rows 3 and 6 present the results of variant (1), and Rows 4 and 7 present the results of variant (2). The results show that, when turning off keyword-based pre-filtering, the F-scores on both Redhat and Mozilla drop significantly. By contrast, CVE mining has a large impact on the results for Mozilla, but very trivial impact on the results for Redhat. This may also due to the different similarity among security bug reports of the two projects. Since Redhat security bug reports are more similar to each other, the 100 security bug reports in the training set may already be able to cover most of security keywords that appear in the rest security bug reports. By contrast, for Mozilla, the 100 security bug reports in the training set cannot cover sufficient keywords, so CVE entries provide very good supplement.

TABLE V. EVALUATION ON KEYWORD-BASED PRE-FILTERING

Subject	Iteration	P (%)	R (%)	F (%)
Redhat	Our Approach	90.4	83.5	86.8
	Turning Off Pre-Filtering	92.5	74.3	81.5
	Turning Off CVE Mining	91.0	83.1	86.9
Mozilla	Our Approach	80.8	48.9	61.0
	Turning Off Pre-Filtering	75.8	40.8	53.0
	Turning Off CVE Mining	76.6	39.3	52.1

TABLE VI. DIFFERENT KEYWORD-LIST SIZES

Subject	Size	P (%)	R (%)	F (%)
Redhat	50	94.0	72.1	81.6
	100	90.4	83.5	86.8
	200	89.1	84.6	86.8
Mozilla	50	85.9	41.4	55.9
	100	80.8	48.9	61.0
	200	70.1	52.6	60.0

3) Impact of Parameters: In this subsection, we compare the results of different variants of our approach (when different parameter values are used). As mentioned in Section V-C1, we consider four parameters of our approach. Since it is very hard to present the results of all different combination of parameter values. We use the typical approach [28] to compare approaches with different parameters. Specifically, we change one parameter of the default setting at a time, while have all other parameters having the default values.

Table VI shows how the results of our approach change with different keyword-list sizes (i.e., 50, 100, and 200). From the table, our observation is that, our approach performs best in Mozilla for a keyword-list size of 100, and performs similarly in Redhat for a keyword-list size of 100 and 200. Actually, when the keyword-list size is very small, it may rule out many security bug reports and thus lower the recall, while when the keyword-list size is too large, it may weaken the dataset balancing, and lower the power of the classification model.

Table VII shows how the results of our approach change with different keyword-based filtering approaches (discriminative and non-discriminative). From the table, we can see that discriminative term-weighting performs better than non-discriminative weighting in Redhat, and slightly worse in Mozilla. The reason may still be the different similarity among security bug reports in the two projects, because discriminative term-weighting tends to result in a keyword list that is more specific to the training set.

Table IX shows how the results of our approach change when combining with different classification models (Naive Bayes and SVM). From the table, we can see that, when used in our approach, SVM performs better than Naive Bayes in both Redhat and Mozilla, and for both classification models, our approach is able to achieve higher F-scores than simply applying them (compared to results in Table II).

Table VIII shows how the results of our approach change when using different information sources from the bug reports.

TABLE VII. DIFFERENT TERM WEIGHTING FORMULAS

Subject	Formula	P (%)	R (%)	F (%)
Redhat	Disc.	90.4	83.5	86.8
	Non-D.	66.3	88.7	74.3
Mozilla	Disc.	80.8	48.9	61.0
	Non-D.	81.1	50.0	61.8

TABLE VIII. TITLE ONLY VS. TITLE PLUS DESCRIPTION

Subject	Info. Source	P (%)	R (%)	F (%)
Redhat	Title + Desc.	90.4	83.5	86.8
	Title Only	95.2	53.1	68.1
Mozilla	Title + Desc.	80.8	48.9	61.0
	Title Only	83.2	41.9	55.7

TABLE IX. DIFFERENT CLASSIFICATION MODELS

Subject	Model	P (%)	R (%)	F (%)
Redhat	SVM	90.4	83.5	86.8
	NB	92.8	78.4	84.7
Mozilla	SVM	80.8	48.9	61.0
	NB	45.0	70.1	54.8

From the table, we can see that, using both title and description performs better than using only title. The major reason may be that, title usually contains small amount of information and small number of terms, but the terms and information in title is are usually more representative. Thus, using only title is achieving higher precision due to less noises, but low recall due to the lack of information.

D. Threats to Validity

When building our data set, we use randomly selected bug reports from the bug repository as non-security bug reports. Since the set of known security bug reports may miss many real security bug reports, we are not able to confirm the non-security bug reports used in our study are all non-security bug reports. This is a threat to the internal validity of our experiment, because it is possible that there are some security bug reports labeled as non-security in our data set, and affect the accuracy of our experimental results. To reduce this threat, when we select non-security bug reports, we excluded the known security bug reports, and performed a semi-automatic removal of suspicious security bug reports (i.e., with keyword and meta matching and manual inspection as introduced in Section V-A). The major threat to the external validity of our experiment is that our conclusion may apply to only the data set being experimented on. To reduce this threat, we use bug reports from two large bug repositories: RedHat and Mozilla. Note that data set of security bug reports are rarely available in public, and it is very hard to build such data sets.

VI. DISCUSSIONS

A. Definition of Security Bug Reports

Although the word "security bug reports" is widely referred to, it is not a well-defined word representing a well-defined set of bug reports. Based on different understanding of "security", and "security bugs", a bug report can be judged as security bug report or not. For example, a display error of a password prompt is related to security, because the bug is about a security-related component, and users may mistakenly input some information (even privacy) to the prompt. However, such a bug is not a vulnerability that allows an attacker to break in the system or steal important personal information.

In our paper, when referring to security bug reports, we indicate bug reports reporting vulnerabilities of software (so that it may be attacked by malicious users). Actually, according to our motivation, these are exactly the bug reports that should be hidden from the public. Also, both our data sets of security bug reports are built based on this criteria. In particular, the Red Hat security bug reports are handled by the "security response team" which respond to potential attacks as well as reporting vulnerabilities to the CVE database. The Firefox bug reports are linked to reported vulnerabilities in Mozilla Foundation Security Advisory.

It should be noted that, our definition of security bugs is relatively narrow, which is consistent with our motivation, but this definition will make the problem of identifying security bug reports more difficult because those security-related non-security bug reports (such as the display error example above) often contain security-related terms and features (e.g., "password" in the example above).

B. The Size and Unbalance of the Data Set

In our evaluation, we used a small training set (i.e., about 100 labeled security bug reports), and our data set is unbalanced (the ratio of security bug reports by non-security bug reports is 1:11). The reason for using such settings is to simulate the actual scenario of using our approach.

As we mentioned in Section V-A, we choose 100 labeled security bug reports to form our training set because it is often mentioned as the smallest size of training set for most mining and learning tasks, and it is often not prohibitively expensive to accumulate 100 labeled security bug reports. Also, we use the 1:11 ratio according to the proportion of security bug reports of Red Had during the period these bug reports are submitted. Although it is obvious that the data set should be unbalanced due to the sparsity of security bug reports, the ratio of security bug reports by non-security bug reports may vary in different projects. The reason is that different software projects may have different amount of code and number of features related to security. At the same time, it is very hard to estimate the proportion of security bug reports in the open bug repository of a software project, because security bug reports are not well labeled in most current software projects. However, we believe that, 1:11 is a relatively high estimation of the ratio of security bug reports to non-security bug reports, and the real data distribution may be more unbalanced. Since we design our approach to handle such sparsity and data unbalance, we conjecture that our approach may bring more benefit when the data distribution becomes more unbalanced, and we plan to carry out experiments to evaluate our approach on lower ratios (requiring more non-security bug reports).

C. Evaluation Metrics

In our evaluation, we use the widely applied F-score to measure the effectiveness of our approach. The benefit of using F-score is that, it combines precision and recall to a single value and makes it easy to compare different approaches. However, depending on how our approach is used, its real effectiveness may not be precisely measured with F-score. Low recall may result in later handling and longer exposure of security bug reports (because the missed security bug reports are triaged at lower priority), and low precision may result in the same thing (because the triager needs to spend more time on false positives). Given different strategies of hiding potential

security bug reports (hide before or after the bug reports are triaged) and efficiency of triaging, precision and recall may be of different importance. Additionally, security bug reports may have different severity, and miss-classification of more severe security bug reports should affect the effectiveness more, while this also cannot be measured with F-score.

VII. Related Works

In this section, we discuss the previous research efforts related to our work.

A. Studies on Security Bug Reports

We are aware of two previous research efforts on identifying security bug reports. Gegick et al. [19] proposed an approach to identifying security bug reports based on keyword mining, and performed an empirical study based on an industry bug repository. More recently, Dehl et al. [12] developed a similar approach to identify security bug reports with keyword mining, and they found that their approach performs better than the Naive Bayes classification model. Both approaches require manually generated start word lists and synonym lists to help the mining, so their effectiveness can be largely affected by the quality of lists (we are not able to access the lists). Furthermore, both approaches are evaluated on bug data sets that are not public available, so we are not able to directly compare our results with them. Actually, Gegick et al.'s work uses a similar word mining strategy with the Non-discriminative version of our baseline approach **BS2**, but their mining process is performed by a commercial tool which we do not have access to. The discriminative version of our baseline approach **BS2** is the same with the word mining strategy of Dehl et al. [12]'s work. However, some parameters (e.g., word list size or word weight threshold) are missing from their paper, so we are not able to build a variant of **BS2** that exactly simulates their approach. Actually, our experimental results show that our technique outperforms all evaluated variants of **BS2**. Besides technical difference, our work in this paper makes two extra contributions. First, we construct a public available data set with labeled security bug reports from open bug repositories which can be used in future research in the area. Second, our evaluation is performed with a minimal training set which is often the only training set available.

B. Classification of Bug Reports

The general problem of classifying bug reports according to certain criteria has been widely researched. The early works in the area mainly try to assign bug reports to different developers. Specifically, Anvik et al. [9], Cubranic and Murphy [14], and Lucca [18] all proposed approaches to automatically assign bug reports to software developers. Recently, Kanwal and Maqbool proposed an approach to prioritize software bugs, and use priority information to help the bug report assignment. Menzies and Marcus also suggested a classification based approach to predict the severity of bug reports [27]. Additionally, Hooimeijer and Weimer suggested a statistics based model to predict the quality of bug reports [22].

C. Semi-Supervised Learning and Handling Data Imbalance

Semi-supervised learning [32] has become more and more popular as the speed of data labeling falls further and further behind the speed of data growth. Though not widely used, it has been adopted for solving several software engineering problems and achieved good results. Specifically, Seliya and Khoshgoftaar [29] proposed to use semi-supervised learning for the estimation of software quality. Li et al. [24] proposed to use semi-supervised learning for defect detection based on sparse historical data. Actually, due to the difficulty of labeling software artifacts and the sparsity of data for new software projects, we believe that semi-supervised learning has great potential on enhancing the practical effectiveness of machine-learning based software engineering techniques.

VIII. Conclusions

In this paper, we proposed an approach to automatically identifying security bug reports based on semi-supervised learning and keyword mining. We constructed two datasets with labeled security bug reports from the RedHat and Mozilla bug repositories, and evaluated our approach on the data sets. The experimental results show that, on Redhat and Mozilla, our approach outperforms the best baseline approaches by 7.4 and 5.7 percentage points on F-score, respectively.

References

[1] Bugzilla, http://www.bugzilla.org/, 2013.

[2] CVE, http://cve.mitre.org/, 2013.

[3] Weka: Data mining software in java, 2013.

[4] Amazon mechanical turk, 2015.

[5] Mozilla bugzilla, 2015.

[6] Redhat bugzilla, 2015.

[7] R. K. Ando and T. Zhang. A high-performance semi-supervised learning method for text chunking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 1–9, 2005.

[8] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[9] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, 2006.

[10] A. Barth, S. Li, B. I. P. Rubinstein, and D. Song. How open should open source be? Technical Report UCB/EECS-2011-98, EECS Department, University of California, Berkeley, Aug 2011.

[11] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29, 2004.

[12] D. Behl, S. Handa, and A. Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pages 294–299, 2014.

[13] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[14] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[15] S. A. Dudani. The distance-weighted k-nearest-neighbor rule. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-6(4):325–327, April 1976.

[16] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(912):1194 – 1213, 2007.

[17] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[18] L. G. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 93, 2002.

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

[20] H. He and E. Garcia. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, 2009.

[21] M. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, Jul 1998.

[22] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, 2007.

[23] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.

[24] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[25] Y. Liu, H. T. Loh, and A. Sun. Imbalanced text classification: A term weighting approach. *Expert Systems with Applications*, 36(1):690 – 701, 2009.

[26] D. McClosky, E. Charniak, and M. Johnson. Effective self-training for parsing. In *Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 152–159, 2006.

[27] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM08: Proceedings of IEEE International Conference on Software Maintenance*, pages 346–355, 2008.

[28] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *proceedings of the International Conference on Software Engineering*, 2007.

[29] N. Seliya and T. Khoshgoftaar. Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Quality Journal*, 15(3):327–344, 2007.

[30] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *J. Mach. Learn. Res.*, 2:45–66, Mar. 2002.

[31] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, 2011.

[32] X. Zhu. Semi-supervised learning. In *Encyclopedia of Machine Learning*, pages 892–897. 2010.

[33] X. Zhu, C. Vondrick, D. Ramanan, and C. Fowlkes. Do we need more training data or better models for object detection? In *Proceedings of the British Machine Vision Conference*, pages 80.1–80.11, 2012.