This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

SAIS: Self-Adaptive Identification of Security Bug Reports

Shaikh Mostafa, Bridgette Findley, Na Meng *Member, IEEE,*, and Xiaoyin Wang *Member, IEEE,*

***Abstract*—Among various bug reports (BRs), security bug reports (SBRs) are unique because they require immediate concealment and fixes. When SBRs are not identified in time, attackers can exploit the vulnerabilities. Prior work identifies SBRs via text mining, which requires a *predefined keyword list* and trains a classifier with *known* SBRs and non-security bug reports (NSBRs). The former approach is not reliable, because (1) as the contexts of security vulnerabilities and terminology of SBRs change over time, the predefined list will become outdated; and (2) users may have insufficient SBRs for training. We introduce a semi-supervised learning-based approach, SAIS, to adaptively and reliably identify SBRs. Given a project's BRs containing some labeled SBRs, many more NSBRs, and unlabeled BRs, SAIS iteratively mines keywords, trains a classifier based on the keywords from the labeled data, classifies unlabeled BRs, and augments its training data with the newly labeled BRs. Our evaluation shows that SAIS is useful for identifying SBRs.**

***Index Terms*—Security bug reports, self learning, bug triaging**

I. INTRODUCTION

Bug tracking systems, such as JIRA [6], are widely used to track various bug reports (BRs) in software development. A common challenge for bug report maintenance is that a small number of bug triagers have to sort a huge number of BRs everyday, leaving many BRs unread or unresolved [9]. On the other hand, among all types of BRs, security bug reports (SBRs) are unique in two aspects. First, it requires triagers with high expertise in the security domain to accurately sort out SBRs. Second, SBRs especially need immediate identification once being created, because the exposed security bugs may cause severe consequences (e.g., money loss or privacy leakage), and thus require for early fixes to avoid potential security attacks. Furthermore, when bug tracking systems are open and SBRs are publicly available, bug report maintainers should hide these reports from any potential attackers, who may exploit the security vulnerabilities [11].

Although SBRs require almost immediate recognition by bug triagers, a study by Zaman et al. [56] showed that the existing triage process is far away from satisfying the requirement. In particular, the average triage time (i.e., the time between an average SBR's submission and its assignment to a developer) of SBRs in Firefox project is 4,000 minutes (66 hours). The study result indicates a strong need for automatic tool support that can identify SBRs efficiently and accurately. With such tool support, software developers can quickly hide the SBRs from public, focus their effort on resolving SBRs and fixing security bugs timely.

Shaikh Mostafa, Bridgette Findley, and Xiaoyin Wang are from the Department of Computer Science, University of Texas at San Antonio. Contact: xiaoyin.wang@utsa.edu

Na Meng is with Virginia Tech. Contact: nm8247@cs.vt.edu

Gegick et al. [20] built an approach that applies text mining to known SBRs and NSBRs, and trains a classification model that automatically decides whether a given BR is SBR or not. Nevertheless, Gegick's approach puts two requirements on its usage scenario, both of which are difficult to satisfy in reality. First, the leveraged text mining technique requires users to manually define a start list—a list of keywords that are directly or indirectly relevant to security, such as "attack" and "excessive". However, developers may not have enough security domain expertise to properly define a such list. More importantly, due to new findings and language features, the vocabulary of SBRs is always expanding (i.e., more and more security-related terms are created and should be added over time). For example, attacks to "md5" hashing mechanism were first found around 2009, so corresponding vulnerabilities started to appear since then. The first "OAuth" protocol was published in 2010, so attacks to vulnerabilities in ?OAuth? implementations are found only after 2010. Therefore, even if developers manage to initially define a high-quality start list, this initial list can gradually become outdated and insufficient. Second, the approach requires for a sufficient number of known SBRs and NSBRs to train a useful classification model, but developers may only have a few SBRs recognized for their own projects. Consequently, developers cannot train a good model with the limited SBR data.

This paper presents our new approach SAIS, which reliably identifies SBRs by overcoming two realistic challenges: *the evolving vocabulary of SBRs and a small set of identified SBRs for specific software projects*. SAIS has two novelties, which were intentionally designed to overcome the two challenges. First, by mining keywords from both provided labeled data and the database of Common Vulnerabilities and Exposures (CVE), SAIS can automatically construct an up-to-date security-relevant keyword list without requiring users of any domain expertise. Second, with semi-supervised learning, SAIS initially trains a classification model with a limited number of project-specific SBRs, uses the model to classify unlabeled data, and then includes the newly labeled data into training to iteratively improve its model training. Such bootstrapping strategy enables SAIS to adaptively identify SBRs even with a small training set initially provided.

We evaluated SAIS on the BRs of two open source projects: RedHat and Mozilla. SAIS identifies SBRs with 87% F-score (the harmonic average of the precision and recall [45]) for RedHat and 61% for Mozilla. To understand how each component of SAIS contributes to the overall performance, we also developed several variants of SAIS by removing either its iterative execution, keyword mining, or model training.

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

In summary, we have made the following contributions:

- We designed and implemented SAIS to automatically identify SBRs by iteratively mining security-relevant keywords and training a classification model for BR labeling. SAIS obtains 87% F-score when labeling RedHat SBRs, and achieves 61% for Mozilla data.
- We conducted a comprehensive empirical study to systematically investigate how each component in SAIS affects the overall performance. Compared with others, the keyword-mining component plays the most important role to improve SAIS.
- We investigated the sensitivity of SAIS to changes in the keyword list length and machine learning algorithm. We found that SAIS worked best when the top 100 mined keywords were included, and we use Support Vector Machines (SVM) to train a classifier in each iteration.
- We publicized our data set[1] with labeled security bug reports to facilitate future comparison between automatic SBR identification techniques.

II. Background

This section first presents a study on triaging time of BRs, and then introduces background knowledge on semi-supervised learning and the CVE database.

A. Study on Triaging Time

To better motivate our research, we conducted a study similar to the prior study [56] on the BRs of two other open source projects: RedHat [4] and Mozilla [3]. As with prior study, we also observed that the SBRs' triage time is long. Fig. 1 presents the cumulative distribution of SBRs' triage time observed in our study. The X-axis shows the triage time ranging from 1 to 1,000,000 minutes with a base-10 log scale. The Y-axis shows the percentage of triaged SBRs. Overall, the medium SBR triage time periods of RedHat and Mozilla are separately 1,820 minutes (30 hours) and 2,637 minutes (44 hours). For RedHat, only 48% SBRs were triaged within 24 hours (1 day), and 61% SBRs were identified within 72 hours (3 days). For Mozilla, even lower percentages of SBRs were triaged in 1 or 3 days (45% and 56% respectively). The considerable time delay between an SBR's creation and its recognition by triagers can offer sufficient opportunities to potential attackers, who learn security vulnerabilities from the exposed SBRs and implement attacks exploiting those vulnerabilities.

B. Concepts of Semi-Supervised Learning

Semi-supervised learning [57] is a category of machine learning techniques that use both labeled and unlabeled data for training. Researchers found that when a labeled data set is insufficient to train a useful model, using a much larger unlabeled data set in conjunction with the labeled data could considerably improve the learning accuracy [8]. There are various semi-supervised learning techniques, including self-training [36] [24], S3VM [32], and Min-Cut [34]. For our project, we use self-training due to its popularity, and we use the unlabeled BRs in the bug repositories to support semi-supervised learning.

[1]The data set and SAIS implementation are available at our anonymous website https://sites.google.com/site/securebugs/

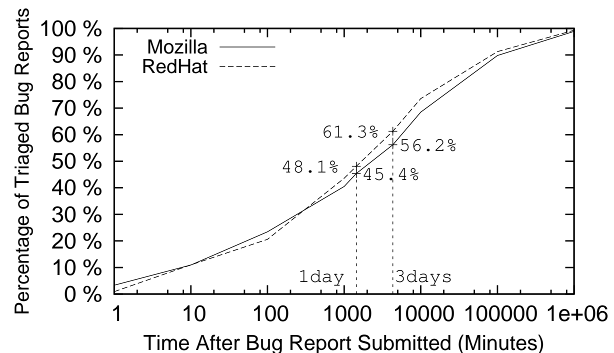

Fig. 1: Cumulative Distribution of SBRs' triage time

Self-training leverages a provided small labeled data set to train an initial classification model. Then it iteratively uses the model to classify unlabeled data, adds the classification results to the training set, and continues using the expanded training set to create a new classification model.

As illustrated in Fig. 2, when users want to label a set of nodes as two classes (Class-1 with circle nodes and Class-2 with triangle nodes), they may provide a small set of 4 labeled data points (2 solid circle nodes and 2 solid triangle nodes), expecting the unlabeled data points to be automatically labeled accordingly. Without self-training, a naïve approach is to directly apply an off-the-shelf machine learning (ML) algorithm, such as SVM, and train a model purely with the labeled data. The trained model may correspond to Line-1 in Fig. 2, which wrongly classifies some nodes (e.g., Node-1 and Node-2) due to the small training set. With self-training, a more advanced approach can still obtain a naïve classification model (e.g., Line-1) within its first iteration by applying an ML algorithm to the labeled data. Nevertheless, for the same iteration, the approach further classifies unlabeled data using the model, and expands its training set with the newly labeled data to prepare for the second iteration of model training. Such iterative process continues until finally the trained model converges (e.g., Line-3), or all nodes' classification labels become stable.

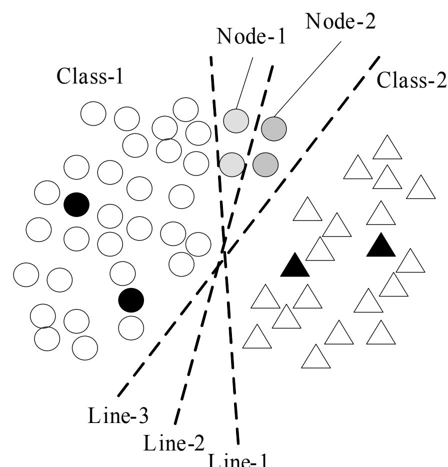

Fig. 2: Illustration of Semi-Supervised Learning

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

CVE-2006-0884 Detail

Current Description

The WYSIWYG rendering engine ("rich mail" editor) in Mozilla Thunderbird 1.0.7 and earlier allows user-assisted attackers to bypass javascript securit settings and obtain sensitive information or cause a crash via an e-mail containing a javascript URI in the SRC attribute of an IFRAME tag, which is executed when the user edits the e-mail.

Impact

CVSS v2.0 Severity and Metrics:
Base Score: 9.3 HIGH
Vector: (AV:N/AC:M/Au:N/C:C/I:C/A:C) (V2 legend)
Impact Subscore: 10.0
Exploitability Subscore: 8.6

Fig. 3: An Example CVE Entry

We chose to use semi-supervised learning, because prior works [17], [42] show that by combining it with any supervised ML algorithm (e.g., SVM), we can improve the learning accuracy especially when the initial training set is small.

C. CVE Database

CVE[2] [2] is an open database that lists information-security vulnerabilities and exposures, and aims to provide common names for publicly known problems. Developers, security analysts, and software vendors can report the vulnerabilities they found by submitting entries to the CVE database. Once an entry is submitted, the CVE editorial board manually investigates the submission and decide whether to accept it. If a submission is accepted, the CVE editorial board (i.e., security experts) further rephrase the original security problem description with security terminology. Until April 2015, the database has contained over 60,000 approved entries.

We chose to include CVE entries for keyword mining, because they contain the up-to-date vocabulary to describe various vulnerabilities. This information resource ensures the mined keywords to be up-to-date with the description of security problems. On the other hand, we also include software projects' BRs as another information resource for mining, because compared with security experts, developers may use a different vocabulary to describe their project-specific security problems.

> **Mozilla Bug 319858: javascript execution when forwarding or replying (CVE-2006-0884).** *It is possible to inject javascript in thunderbird 1.0.7 and 1.5rc. the js is executed if the luser selects "forward as inline" (1.5rc) or "forward as inline" or "reply" (1.0.7)...*

Figure 3 shows the CVE entry (CVE-2006-0884) for the bug above as an example. Comparing the CVE entry and the bug report, we can see that the CVE entry describes the bug from security experts' perspective and in a more general way, while the bug report describes the bug from user's perspective and is specific to the context of discovery. Also, the bug report contains informal words such as "js", typos such as "luser" (should be "user"), and more software-specific words such as "forward".

[2]https://cve.mitre.org/

III. Approach

As shown in Fig. 4, SAIS executes mainly four steps iteratively, until labels of all BRs no longer change. To mine security-relevant keywords, step ① takes in the labeled SBRs and NSBRs from a project's bug tracking system together with posts in the CVE) database [2]. The project-specific labeled data contain terms (i.e., words) used in SBRs and NSBRs, while the CVE entries include terms used in security problem descriptions. By applying text mining to the documents from these two resources, SAIS can automatically reveal the state-of-the-art security-relevant vocabulary, and thus identify the most frequently used keywords. For clarity, we name the initial labeled data as $\boldsymbol{L_0}$. Note that we use CVE database to supplement our dataset because usually only a small set SBR set is available. Our later evaluation shows that, although not as helpful as project-specific keywords, CVE-based keywords can enhance the performance of SAIS on Mozilla, but they are only helpful when iterative execution is performed in SAIS.

Based on the keywords, step ② converts each labeled BR to a numeric vector, in which each element corresponds to an identified keyword, and the element's value (1 or 0) indicates whether the BR contains the keyword or not. Note that the length of the vector is the same as the total number of CVE-based keywords and project specific keywords. This step takes effect ever since the second iteration of steps (1) through (4). Specifically, in the nth iteration ($n > 1$), this step refines the newly labeled SBRs from the $(n-1)^{th}$ iteration. As some of the automatically labeled SBRs may be false alarms, SAIS implements this intuitive filter to remove falsely labeled SBRs before using the data for training in the next step. For instance, given an automatically labeled SBR, this step checks whether the numeric vector purely contains 0's (i.e., whether the SBR contains no security-relevant keyword). If so, the SBR is regarded as a false alarm, and relabeled with" NSBR".

Step ③ takes the numeric vectors of refined labeled data as input. It leverages a machine learning algorithm—support vector machine (SVM) [21]—to train a classification model with the numeric vectors. For clarity, we call the labeled data used in this step as $\boldsymbol{L_i}$.

Finally, step ④ uses the classification model to classify the unlabeled BRs, creating a new labeled data set $\boldsymbol{L_n}$. For any BR commonly shared between L_n and L_i, SAIS checks whether the BR is labeled divergently. If yes, SAIS adds L_n to L_i (the data initially labeled from the last iteration) to iteratively retrain a classifier for data relabeling; otherwise, it outputs the identified SBRs in L_n.

This section will first explain the keyword-mining step (Section III-A), and then describe how the mined keywords are used to refine labeled data (Section III-B). Next, we will discuss how to train a classification model with the refined data (Section III-C), and will expound on the labeling process and the transition to the next iteration of all steps (Section III-D). Finally, we present and explain the fully integrated algorithm combining the four steps in Section III-E.

A. Keyword Mining

Given an unbalanced labeled data set (e.g., containing m SBRs and n NSBRs, where $m < n$), SAIS first randomly

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

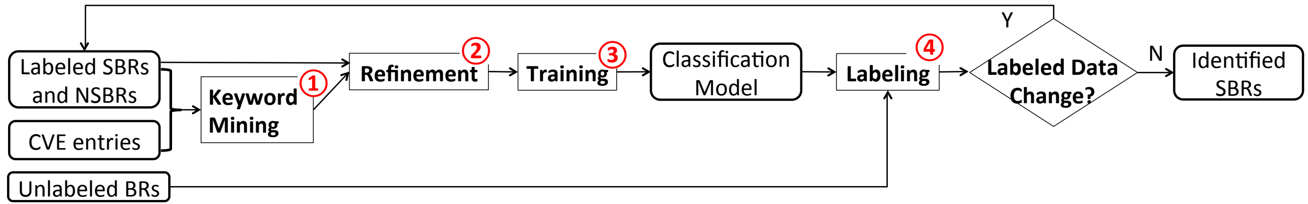

Fig. 4: SAIS has four steps. Step ① mines keywords from labeled data and CVE entries. Step ② represents the labeled data as keyword vectors and refines the data accordingly. Step ③ trains a model based on the refined data. Step ④ classifies unlabeled BRs with the trained model, and augments the original training data with newly labeled BRs. SAIS repeats the four steps until all unlabeled BRs are stably labeled.

selects n CVE entries from the over 60,000 entries stored in our local machine. SAIS then treats the sampled entries as BRs to uniformly mine and rank security-relevant keywords by extending a classical tf-idf term weighting approach [33]. For simplicity, we generally call the SBRs together with the sampled CVE entries as *security documents (SD)*, where the cardinality of SD is $|SD| = m + n$. We similarly call NSBRs as *non-security documents (NSD)*, where $|NSD| = n$.

For each document d, we first extract the terms or words from only the summary or description of the problem (either security-relevant or irrelevant), because such information is always accessible to bug triagers when they sort BRs. We then perform automatic stemming to normalize the term representation (e.g., "executed"→"execute"). Next, for each extracted term t, we further count the number of its occurrence in the document d, denoting the term frequency with $tf(t, d)$. Given the two categories of documents (SD vs. NSD), we similarly compute t's frequency in a category cat as below:

$$\boldsymbol{tf(t, cat)} = \sum_{\boldsymbol{d \in cat}} \boldsymbol{tf(t, d)}. \tag{1}$$

Inverse document frequency (idf) was originally defined to measure the specificity of term t in the following way [1]:

$$idf(t) = \log(\frac{\text{\# of all documents}}{\text{\# of documents containing } t}). \tag{2}$$

Intuitively, the higher $idf(t)$ is, the more special term t is, and thus the more important t is to characterize certain type of documents. In our research, as we are more interested about how well each term characterizes SDs or NSDs, we adapted the above idf formula in the following way:

$$\boldsymbol{idf(t, cat)} = \log(\frac{\mathbf{1 + \text{\# of all documents not in } \boldsymbol{cat}}}{\mathbf{1 + \text{\#documents not in } \boldsymbol{cat} \text{ containing } \boldsymbol{t}}}). \tag{3}$$

Intuitively, if a term t_1 purely occurs in every NSD document, then $idf(t_1, SD) = log(\frac{1+n}{1+n}) = 0$, $idf(t_1, NSD) = log(\frac{1+m+n}{1+0}) = log(1 + m + n)$. The values indicate that t_1 can well characterize NSDs.

The classical td-idf metric was defined to measure term t's representativeness for a document d in the following way:

$$tfidf(t, d) = tf(t, d) \cdot idf(t). \tag{4}$$

We adapted the original formula to correspondingly measure term t's representativeness for a document category cat as below:

$$\boldsymbol{tfidf(t, cat) = tf(t, cat) \cdot idf(t, cat)}. \tag{5}$$

As we aim to rank terms based on their capabilities to differentiate SDs from NSDs, we further define *discriminative weight* of term t as below:

$$\boldsymbol{descrimitive_weight(t)} = \frac{\boldsymbol{tfidf(t, SD)}}{\boldsymbol{tfidf(t, NSD)}}. \tag{6}$$

Theoretically, the more weight a term t has, the more important it is to differentiate between different kinds of documents.

Based on the weights of each term, SAIS ranks the terms that appear in SDs, and then selects a list of top 100 terms as the security-relevant keywords used in the following steps (see Table VI for results of using different list length). Notice that although this step is executed in every iteration, SAIS only uses the CVE sampled data in the first iteration to complement the limited SBRs provided by users. In the follow-up iterations, as sufficient automatically labeled SBRs are generated and included, SAIS does not need the CVE data to mine keywords. Furthermore, 100 is set as the default size of keyword list according to our experiments presented later in Section IV.

B. Keyword-Based Refinement

This step aims to refine the training data based on mined keywords from the second iteration of the whole process, when the newly labeled data may incur noisy information.

Our keyword-based refinement algorithm is presented in Algorithm 1. Given a labeled BR in the original training set, if it is an NSBR, SAIS will directly add it to the refined training set. Otherwise, SAIS will check whether it contains any keyword in the list of 100 keywords mined from step ①. If an SBR does not contain any of the keywords, it means that the SBR does not have any known security-relevant keyword. Therefore, it is possible that the SBR is incorrectly labeled, and SAIS further relabels the document with "NSBR" and adds it to the refined training set (see more information on such cases in the explanation of Table II). The SBRs with at least one security-related keywords are also added to the training set without changing their labels.

Notice that we intentionally skip this step's relabeling part for the first iteration, because we assume that the manually labeled SBRs given by users are always correct. Even if a manually labeled SBR does not contain any top-ranked security-relevant term, it may be because the mined CVE data fails to reveal the actual security problems specific to users' software project(s). Therefore, we trust manually labeled SBRs without validating their correctness. However, ever since the second iteration, as new labeled SBRs are generated by the

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

Algorithm 1: Keyword-based Refinement of Training Data

```
Input: Labeled Training Set T, Keyword List KEY
Output: Re-Labeled Training Set T'
1  T' = ∅ for Bug report r in T do
2      if r is NSBR then
3          Add r to T'
4      end
5      for Keyword k in KEY do
6          if k ∈ r then
7              Add r to T'
8          end
9      end
10     Add r to T' as NSBR
11 end
12 return T'
```

model created in the previous iteration, it is possible that some SBRs are incorrectly identified. Therefore, our refinement process filters out the obviously wrongly classified SBRs.

C. Model Training

To train a binary-class classifier for SBR identification, we prepare BRs in the training data as feature vectors. For a given BR, we first remove all the stop words [44], and perform stemming [39] to unify words with the same morphological roots. Then, all the remaining words (morphological roots) are considered features of the BR, and the corresponding feature values are the their frequencies in BR. The training set includes both positive and negative data points. Each data point has the following format: $< numeric_vector, label >$, in which label is either 1 to represent "SBR", or 0 to represent "NSBR". We leverage liblinear [18]—a software library implementing a collection of machine learning algorithms—to conduct machine learning. By default, SVM [21], an off-the-shelf machine learning algorithm, is used in this step to train a classification model based on the training data. We used SVM's default parameter settings in liblinear to train classifiers.

D. Labeling

With the trained classification model, SAIS further classifies unlabeled BRs into SBRs and NSBRs. Given a BR, the model outputs the document's likelihood of being an SBR. By default, we set the likelihood threshold as 0.5, meaning that if the output likelihood is greater than 0.5, SAIS automatically labels the document with "SBR"; otherwise, the document is labeled with "NSBR".

After classifying all unlabeled BRs in the $n^{th}(n > 0)$ iteration, SAIS further checks for each BR, whether its newly assigned label is different from the original label (either "UNLABELED" or the label assigned in the previous iteration). If any BR's label is changed, SAIS augments the user-provided labeled data with all automatically labeled data, and continues a new iteration of all four steps to train a better classification model. On the other hand, if all BRs' labels are unchanged, SAIS concludes that the classification results converge, and stops iteration to output all identified SBRs.

E. Integrated Algorithm

Based on the four steps detailed above, we present how we combine them in the self-training framework in the following algorithm.

Algorithm 2: 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 = mine_1(R ∪ E);
2  L = R ∪ S;
3  while ∃ br ∈ L ∧ br's label is updated do
4      L' = refine_2(L, keywords);
5      if first round then
6          model = train_3(R);
7      else
8          model = train_3(L');
9      end
10     L = label_4(L' ∪ S, model);
11     keywords = mine_1(L);
12 end
```

In Algorithm 2, 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. Our four steps are denoted in the algorithm as $mine_1$, $refine_2$, $train_3$, and $label_4$. 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 our first two steps, keyword mining and keyword-based refinement, we added three underlined lines (i.e., Lines 1, 4, 11).

IV. Evaluation

In this section, we first introduce our data set for evaluation (Section IV-A), and then describe our evaluation methodology (Section IV-B). Based on the data set and methodology, we evaluated SAIS (Section IV-C), and SAIS's several variants (Section IV-D). We further compare SAIS with some existing related works in Section IV-E. After that, we investigated different machine learning algorithms (Section IV-F) and different lengths of the created keyword list (Section IV-G) to explore the best way of performing keyword mining and machine learning for SBR identification. Finally, we present a study of security-relate keyword distribution over our data set and a chronological study of SAIS in Section IV-H.

A. Data Set Construction

We created two data sets based on the bug tracking systems of RedHat [4] and Mozilla [3]. As shown in Table I, our RedHat data set includes 9,600 labeled BRs mined from the original bug system, while the Mozilla data set covers 6,568 labeled BRs. To construct these data sets, we identified SBRs and NSBRs in different ways.

Identification of Security Bug Reports. One big challenge of extracting SBRs from open bug tracking systems is that

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

TABLE I: Evaluation data sets

Project	# of SBRs	# of NSBRs	# of labeld BRs
RedHat	800	8,800	9,600
Mozilla	568	6,000	6,568

these systems usually do not have any special tag to indicate security bugs, neither is it feasible for us to manually inspect every BR to discover the SBRs. Fortunately, we managed to leverage some unique information in the BRs of RedHat and Mozilla to quickly recognize SBRs. Specifically for RedHat, there is a developer team called "Security Response Team" that especially handle security bugs. Through sampling and inspecting the BRs handled and confirmed by this team, we finally obtained 800 SBRs.

In Mozilla, vulnerabilities were periodically announced to the public [5], and the vulnerability announcements contain links to the original BRs. Based on the mentioned BR links, we obtained 568 SBRs for Mozilla. In particular, when one vulnerability announcement refers to multiple BRs, we treated the documents as separate SBRs, because in reality, bug triagers only care about how to sort BRs (e.g., SBRs vs. NSBRs), instead of checking whether a newly reported BR is a duplicate or supplement of an already-reported issue.

Identification of Non-Security Bug Reports. In dataset construction, we try to make the ratio of SBR/NSBR in our dataset similar to that of reality. So we estimated the ratio between the occurrence rates of SBRs and NSBRs, and then sampled NSBRs accordingly. In this way, we ensure our labeled data sets to realistically reflect the two types of BR's distributions. One big challenge here is that open bug systems do not have any precise security-relevant category information for any BR. Therefore, we can only roughly estimate the ratio by taking the best effort. Especially for RedHat, we observed that the BRs handled by the "Security Response Team" counted for 8% of the overall BRs in the raw data. Assuming that the reports touched by the "Security Response Team" are security-relevant and those untouched are security-irrelevant, we assessed the rough ratio between SBRs and NSBRs to be 1:11. Based on the estimated ratio, for the 800 sampled SBRs of RedHat, we decided to sample 8,800 NSBRs correspondingly. For the 568 sampled SBRs of Mozilla, we determined to sample 6,000 NSBRs. To sample NSBRs, based on the above-mentioned procedure to identify SBRs, here we label a sampled BR with "NSBR" if it does not meet the SBR criteria mentioned above.

B. Evaluation Methodology

K-fold cross validation (CV) is a widely adopted method to assess how well a classification approach performs on independent data sets [29]. CV splits a data set into k subsets, and executes the classification approach k times. For each execution, CV uses some of the subsets for training, and the remaining subset(s) for testing. The overall evaluation result is the average among the k executions.

In our project, we split RedHat's data set so that each subset includes 100 SBRs, and 1,100 NSBRs to maintain the 1:11 ratio. Similarly, we split Mozilla's data so that every subset includes 110 SBRs and 1,200 NSBRs. To simulate the real-world scenario, when classifying BRs for individual projects, we used one subset for training and the remaining subsets for testing. We intentionally configured every subset to cover at least 100 SBRs, because prior work shows that 100 is the minimal data size for meaningful machine learning [50], [58].

To measure an SBR identification approach, we defined and used the following three metrics: precision, recall, and F-score.

Precision (P) measures among all the labeled SBRs, how many reports are true SBRs:

$$P = \frac{\text{\# of true SBRs}}{\text{\# of all labeled SBRs}} * 100\%. \quad (7)$$

Recall (R) measures among all true SBRs, how many reports are labeled as SBRs by an automatic approach:

$$R = \frac{\text{\# of labeled SBRs}}{\text{\# of all true SBRs}} * 100\%. \quad (8)$$

F-score (F) is the harmonic mean of precision and recall as below:

$$F = \frac{2 * P * R}{P + R} * 100\%. \quad (9)$$

Precision, recall, and F-score all vary within [0%, 100%]. The higher F-scores are desirable, because they demonstrate better accuracy of SBR identification.

C. SAIS's Evaluation Results Over Iterations

Table II presents how SAIS's evaluation result varies with the number of iterations executed. As shown in the table, with the initial labeled data set and CVE entries, SAIS only obtained 81% F-score for RedHat and 53% F-score for Mozilla. After six or five iterations of execution, the results stabilized, indicating that the trained classification models converged. SAIS acquired 87% F-score as its best result for RedHat and 61% F-score as the best one for Mozilla.

Furthermore, as more iterations were involved, SAIS's recall rates and F-scores for both projects monotonically increased, while the precision rates generally decreased. One possible reason to explain this phenomenon is that in the first iteration, with the limited provided labeled data (L_i), the trained classifier could only identify the SBRs which shared many terms with L_i. As a result, the initial precision rate was high, while the recall was relatively low. Ever since the second iteration, the training set L_r included automatically classified data in addition to L_i. The extra data enabled the trained model to retrieve some SBRs that were less similar to L_i, and thus improved the recall rate. Meanwhile, the extra data also introduced noise by also including some wrongly classified data, and thus lowered the resulting precision rate. During the iterations, for Red Hat, averagely 139 BRs initially labeled as NSBR were relabeled as SBR, and 10 BRs initially labeled as SBR were relabeled as NSBR. For Mozilla, averagely 74 BRs initially labeled as NSBR were relabeled as SBR, and 9 BRs initially labeled as SBR were relabeled as NSBR.

Finding 1: SAIS*'s F-score monotonically increased with the number of iterations, indicating that semi-supervised learning helps improve* SAIS*'s classification accuracy.*

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

TABLE II: SAIS's evaluation results with different numbers of iterations

Project	# of Iterations	P (%)	R (%)	F (%)
RedHat	1	95	71	81
	2	92	80	86
	3	91	83	86
	4	91	83	87
	5	90	84	87
	6	90	84	87
Mozilla	1	84	39	53
	2	81	47	60
	3	82	48	61
	4	81	49	61
	5	81	49	61

D. SAIS*'s Variant Approaches*

There are multiple steps and various data sets involved in SAIS. To investigate whether the steps or data sets were necessary, we developed the six variants of SAIS shown below:

(1) SAIS$_{mu}$ applies SVM to L_i, without keyword mining, CVE data, or iterative execution. This variant naïvely trains a classification model based on the initially provided labeled data L_i. To create a feature vector for each BR, SAIS$_{mu}$ first extracts terms from the security problem description, removes stop words, and conducts stemming. As SAIS$_{mu}$ does not calculate terms' weights, it then leverages all the remaining terms to construct a vocabulary, based on which each document is converted to a numeric vector.

(2) SAIS$_{mb}$ applies SVM to a balanced training set created from L_i, without keyword mining, CVE data, or iterative execution. This variant is similar to the above-mentioned variant, except that it creates a balanced training set from L_i by making 11 copies of each original SBR.

(3) SAIS$_{mc}$ applies SVM to a balanced training set consisting of labeled NSBRs and an equal number of CVE entries. No keyword mining, labeled SBRs for training, or iterative execution is involved. This variant is similar to SAIS$_{mb}$, except that it creates the balanced training set by using the NSBRs of L_i's complementary set and CVE entries. For instance, the L_i of RedHat contains 100 SBRs and 1,100 NSBRs, so L_i's complement set includes 700 SBRs and 7,700 NSBRs. Instead of using the project-specific SBRs, SAIS$_{mb}$ randomly samples 7,700 CVE entries, and uses them together with the 7,700 NSBRs to train a classification model. Finally, SAIS$_{mb}$ uses L_i for testing.

(4) SAIS$_k$ solely relies on keyword mining to identify SBRs, without machine learning, CVE data, or iterative execution. After mining the top 100 keywords based on L_i, SAIS$_k$ uses the keywords to check whether any BR has any mined keyword. If so, the document is labeled as an SBR; otherwise, it is labeled as an NSBR.

(5) SAIS$_{nr}$ disables the refinement step in SAIS.

(6) SAIS$_{nc}$ disables the CVE data used in SAIS.

(7) SAIS$_{ht}$ disables keyword-based filtering, but uses higher threshold in SVM classification. Here we want to check whether higher classification threshold (i.e., 0.8) can achieve similar results as keyword-based filtering.

TABLE III: Comparison between SAIS and its variants

Project	Approach	P (%)	R (%)	F (%)
RedHat	SAIS	90	84	87
	SAIS$_{mu}$	96	67	79
	SAIS$_{mb}$	83	76	79
	SAIS$_{mc}$	68	28	40
	SAIS$_k$	65	83	73
	SAIS$_{nr}$	93	74	82
	SAIS$_{nc}$	91	83	87
	SAIS$_{ht}$	98	65	78
Mozilla	SAIS	81	49	61
	SAIS$_{mu}$	74	39	51
	SAIS$_{mb}$	63	49	55
	SAIS$_{mc}$	8	3	4
	SAIS$_k$	19	57	28
	SAIS$_{nr}$	76	41	53
	SAIS$_{nc}$	77	39	52
	SAIS$_{ht}$	80	35	49

1) Supervised Learning without Keyword Mining: Variants (1-3) (i.e., SAIS$_{mu}$, SAIS$_{mb}$, and SAIS$_{mc}$) apply supervised learning without keyword mining. Table III presents their evaluation results. These results are worse than SAIS's. Furthermore, we also compared the data with SAIS's first iteration results shown in Table II to evade any potential influence caused by SAIS's iterative execution. From the comparison, we learnt that SAIS$_{mu}$ and SAIS$_{mb}$ achieved comparative performance. For instance, both SAIS$_{mu}$ and SAIS$_{mb}$ obtained 79% F-score for RedHat, while SAIS's first iteration derived 81% F-score. Similarly, SAIS's first iteration acquired 53% F-score for Mozilla, which value is close to SAIS$_{mu}$'s 51% and SAIS$_{mb}$'s 55%. Nevertheless, SAIS$_{mc}$ worked much worse than SAIS's first iteration for both data sets.

The above comparison indicate three insights. First, keyword mining is not important for supervised learning. Although neither SAIS$_{mu}$ nor SAIS$_{mb}$ conducted keyword mining as SAIS's did, their results were comparable to SAIS's first iteration. Second, SAIS$_{mb}$ worked slightly better than SAIS$_{mu}$, meaning that oversampling is helpful to some extent on handling unbalanced data. Third, the project-specific labeled SBRs are crucially important to train a good classification model. When SAIS$_{mc}$ replaced the project-specific labeled SBRs with CVE entries for training, even though more CVE entries were used, SAIS$_{mc}$'s evaluation results were much poorer than SAIS's first iteration. This may be because CVE entries contain so diverse security-relevant information that they do not help much when identifying the SBRs of a particular software project.

Finding 2: *For the first iteration, keyword mining does not quite help supervised learning to identify SBRs, but project-specific labeled SBRs are vitally important for model training.*

2) Keyword-based Labeling: SAIS$_k$ replaces the supervised learning approach investigated via SAIS$_{mu}$ with a naïve keyword-based labeling. The comparison between SAIS$_{mu}$ and SAIS$_k$ shows that supervised learning outperforms keyword-based labeling. This is understandable, because with supervised learning, SAIS$_{mu}$ models the possible co-existence relationship among terms for SBRs and NSBRs. Consequently,

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

SAIS$_{mu}$ can selectively label SBRs with higher precisions but relatively lower recalls, obtaining higher F-scores.

Finding 3: *Compared with supervised learning, keyword-based labeling identifies SBRs with lower precisions, higher recalls, and lower F-scores.*

3) SAIS without Refinement: By disabling the refinement step in SAIS, SAIS$_{nr}$ loses the functionality to refine training data with an intuitive filter, which can relabel SBRs as NSBRs if the SBRs do not include any mined keyword. As expected, SAIS$_{nr}$ worked worse than SAIS, because it cannot get rid of some obviously wrongly classified data. With more noisy data used for training, SAIS$_{nr}$'s trained model further deviates from the ideal model.

Finding 4: *Disabling the refinement step in* SAIS *can compromise the overall evaluation results, because this step is helpful to correct those obviously wrong training data.*

In Finding 2, we found that applying keyword-based refinement in the first iteration does not help much. One reason is that the SBRs in the first iterations are manually labeled so less noise is involved. In contrast, in Finding 4, we found hat when keyword-based refinement is used in late iterations, it is helpful to correct the wrongly classified BRs.

4) SAIS without CVE data: While ignoring the CVE data used in SAIS, SAIS$_{nc}$ worked equally well with SAIS for RedHat by achieving the same F-score: 87%, and performed much worse than SAIS for Mozilla (52% vs. 61%). This comparison corresponds to some of our observations in Section IV-D1.

One possible reason to explain why SAIS$_{nc}$ worked as well as SAIS for RedHat is that RedHat share more common terms with the CVE than Mozilla's SBR do. Consequently, even though we do not use the CVE data in SAIS$_{nc}$, the F-score is unaffected. This may be due to the fact that RedHat has a professional "Security Response Team" to manage SBRs. Therefore, the SBRs are better organized and contain more security-relevant terms.

SAIS$_{nc}$ worked worse than SAIS for Mozilla, perhaps because Mozilla's SBRs are very different from each other. Our statistics show that the average frequency of top 20 security-related keywords in Mozilla dataset (7.6) is much lower than that of RedHat (13.5). As a result, the CVE entries can help retrieve more SBRs whose security keywords do not appear in the training set.

Finding 5: *Disabling the CVE data in* SAIS *caused worse results for Mozilla, but produced no change for RedHat. This may be because Mozilla contains a more variety of security bugs, with SBRs described in professional or unprofessional ways.*

5) SAIS with High Classification Threshold: Our evaluation shows that using higher classification threshold to replace keyword-based refinement does not perform as well as our default technique. One potential reason is that, in semi-supervise learning, the original training set may contain noises, adding keyword filtering brings more information and may neutralize the classification errors caused by the noises, but simply using a higher threshold does not have this benefit.

TABLE IV: Comparing SAIS with Existing Work

Project	ML	P (%)	R (%)	F (%)
RedHat	SAIS	90	84	87
	SVM	96	67	79
	NBM	70	81	75
	Random Forest	95	65	77
	Thung et al.	93	73	82
Mozilla	SAIS	81	49	61
	SVM	74	39	51
	NBM	41	50	45
	Random Forest	80	39	52
	Thung et al.	75	41	53

E. Comparison with Existing Techniques

We are aware of two existing works on the classification of BRs using supervised or semi-supervised learning approaches. In particular, Tyo [51] performed studies on using various standard supervised learning approaches to identify SBRs and to classify them into different categories. The study shows that Naïve Bayes Multi-Nominal (NBM) [28], Random Forest [22], and SVM [21] achieved best results in the three datasets used. Therefore, we compare SAIS with all three algorithms on our two datasets and present the results in Table IV (middle three rows of each section). It should be noted that Tyo's work also classify SBRs into categories such as memory access bugs, design phase bugs, etc. We believe that such categorization can be very useful in many scenarios but our work does not further categorize SBRs for the following reasons. First, we focus on the risk of SBRs not being hidden from the public before they are resolved, and identifying SBRs is sufficient to address such a risk. No matter what bug type the SBR belongs to, The triager can remove the SBR from public visibility as long as it is found to be an SBR. Second, the dataset used in Tyo's work is from NASA and all bugs are well categorized. In open source software such as RedHat and Mozilla, security bugs are not labeled with their types and the ground truth of SBR types is never clear. So we compare our techniques with Tyo's work only on SBR identification. Thung et al. [49] proposed to use semi-supervised learning to classify BRs. Instead of identifying SBRs, their approach categorizes BRs to control / data flow bugs, structural bugs, and non-code bugs. However, their approach can still be applied to our usage scenario. For semi-supervised learning, they also use self-training, but they do not perform keyword-based refinement and CVE word mining because they did not target SBRs.

We re-implemented the techniques in Tyo's work and Thung et al.'s work, and compare their results with SAISand present the results in Table IV. The results of three supervised learning algorithms in Tyo's work are in Rows 2-4 of each section, and the results of Thung et al.'s work are in Row 5 of each section. From the table, we can see that for binary classification of SBRs and NSBRs (i.e., identification of SBRs) our approach outperforms all three supervised learning algorithms in Tyo's work on both RedHat and Mozilla data sets. SAIS also outperforms Thung et al.'s work on both RedHat and Mozilla data sets.

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

TABLE V: SAIS's results with different ML algorithms

Project	ML	P (%)	R (%)	F (%)
RedHat	SVM	90	84	87
	Naïve Bayes	93	78	85
	Random Forest	93	79	86
Mozilla	SVM	81	49	61
	Naïve Bayes	45	70	55
	Random Forest	84	46	59

F. Sensitivity to Machine Learning Algorithms

By default, SAIS leverages SVM to train a classification model in each iteration. We are curious how sensitive SAIS's overall results is to the selected ML algorithm. Therefore, in addition to SVM, we also investigated integrating other ML algorithms into the training step, such as Naïve Bayes [26] and Random Forest [22]. These two ML algorithms are widely used and based on different theoretical foundations.

As shown in Table V, SVM achieved the highest F-scores: 87% for RedHat and 61% for Mozilla. Random Forest obtained slightly lower F-scores: 86% for RedHat and 59% for Mozilla. Naïve Bayes worked worst, deriving 85% F-score for RedHat and 55% for Mozilla. Especially, Random Forest achieved higher precisions but lower recalls than SVM for both projects, indicating that the classifiers trained by Random Forest were usually more selective when identifying SBRs. Nevertheless, Naïve Bayes had better precision and worse recall than SVM when working on RedHat data, but worse precision and better recall when working on Mozilla data. We do not observe any obvious trend from these comparison results.

Finding 6: *When being used by* SAIS*, SVM worked slightly better than Random Forest, but much better than Naïve Bayes.* SAIS*'s results are sensitive to the ML algorithm* SAIS *leverages.*

G. Sensitivity to the Mined Keyword List's Size

The default keyword-list size used in SAIS is 100, meaning that SAIS leverages the top 100 ranked keywords to represent BRs with numeric vectors and to refine labeled data. In this section, we aim to explore how SAIS's results vary when different numbers of keywords are used.

TABLE VI: SAIS's results with different keyword-list sizes

Project	Size	P (%)	R (%)	F (%)
RedHat	50	94	72	82
	100	90	84	87
	200	89	85	87
Mozilla	50	86	41	56
	100	81	49	61
	200	70	53	60

Table VI presents SAIS's results with different settings of keyword-list sizes. According to the table, when $size = 100$, SAIS worked best. Compared with the other settings, $size = 100$ made SAIS to obtain the highest F-scores: 87% for RedHat and 61% for Mozilla.

When $size = 200$, SAIS worked equally well with $size = 100$ on RedHat (87% vs. 87%), but slightly worse on Mozilla (60% vs. 61%). Specifically, SAIS's precisions were lower but recalls were higher in both projects. This may be because as more keywords are included, some less discriminative keywords commonly exist in both SBRs and NSBRs. When such less discriminative terms are used as features, they cannot distinguish well between the two types of BRs. When being used as a refinement filter, these terms cannot sensitively identify some wrongly classified BRs, neither can they correct these wrong classification results. As a result, $size = 200$ led to higher recalls and lower precisions, because more keywords are likely to label more SBRs.

When $size = 50$, SAIS's F-scores went down for both projects. The F-score decreased from 87% to 82% for RedHat, and from 61% to 56% for Mozilla. In particular, $size = 50$ improved precisions but worsened recalls. One possible reason to explain the phenomena is that as fewer highly ranked keywords are included, the keywords put higher constraints on the documents to be recognized as SBRs: the documents must contain the discriminative keywords in certain ways. Consequently, the SBRs identified based on these keywords are usually similar to the original labeled set, and thus are more likely to be true SBRs. However, the downside of using fewer keywords is that although some true SBRs are less similar to the provided training data, they cannot be retrieved or recognized, so recall rates suffer.

Finding 7: *When setting the keyword-list size as 100,* SAIS *obtained the best performance. It indicates that* $size = 100$ *leads to the best trade-off between precision and recall compared with other experimented settings.* SAIS *is sensitive to the keyword-list size.*

H. Studies on Keyword Distribution

Finally, we studied the distribution of keywords in different datasets and over time. In table VII, Column 1 indicates the size of considered top keyword list, and Columns 2 and 3 present the number of common keywords between CVE and RedHat data set, and between CVE and Mozilla data set. From the table, we can see that CVE can provide many new keywords, and the common keywords between CVE and Redhat data set is more than the common keywords between CVE and Mozilla data set.

TABLE VII: Keyword Commonality between Datasets

Size	CVE-RedHat	CVE-Mozilla
10	5	4
50	23	15
100	51	36
200	87	62

We further studied how keyword distribution changes over time. In particular, we compared the top keyword lists extracted from the chronological earlier half (on or before 05/03/2006 for Red Hat, and on or before 05/26/2010 for Mozilla) of the SBR sets with those extracted from the later half of SBR sets, and counted the common keywords. The results are shown in Table VIII. From the table, we can see that the Mozilla dataset has less common keywords in the first

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

and second half of BRs. This also shows that Mozilla SBRs are more different from each other and keywords in Mozilla project are changing faster. One potential reason is that Mozilla was frequently releasing new versions during the submission time (2004-2012) of Mozilla BRs in our dataset.

TABLE VIII: Keyword Change Over Time

Size	RedHat	Mozilla
10	7	5
50	39	23
100	77	52
200	141	89

To investigate whether SAIS is able to handle evolving keyword sets, we further performed a longitudinal study to train the classification model with older data but apply it to newer data. In particular, instead of using cross-validation, we used chronologically earliest 100 BRs and NSBRs during the corresponding time in each dataset to train the model, and use the rest BRs as the testing set. To further check how the size of initial SBR set affects results, we also used the earliest 50 and 200 BRs in the evaluation. The results are presented in Table IX.

TABLE IX: Chronological Study of SAIS

Project	SBR Set Size	P (%)	R (%)	F (%)
RedHat	50	92	67	77
	100	93	83	88
	200	94	85	89
Mozilla	50	81	40	54
	100	80	47	59
	200	81	52	63

From the table, we can see that SAIS can achieve similar results under chronological evaluation settings. Furthermore, when the size of initial training SBR set size decreases, SAIS has lower recall and F-score. Therefore, for the projects with very small number of SBRs (e.g., 50) serving as training set, we may need to further consider other approaches such as transfer learning to learn from other projects at the beginning.

V. 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 (e.g., the prompt does not show or the label of password input box is missing) 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 IV-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 Hat 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 and Threats to Validity

In our evaluation, we use the widely applied F-score to measure 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 usefulness 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).

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

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 usefulness more, while this also cannot be measured with F-score.

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 IV-A). Another threat to internal validity is our implementation of existing works [51], [49] may not be exactly the same with theirs. As both existing works use certain combinations of standard machine learning algorithms and pre/post-processing steps, we expect the threat is not severe. To reduce this threat, we try use standard implementation of machine learning algorithm and read the two papers carefully to make sure we caught all pre/post-processing steps in their techniques. 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 often not available in public, so it is difficult to build such data sets.

D. Practical Usefulness

Although SAIS (and prior techniques for SBR identification) does not achieve very high F-score in some projects (e.g., Mozilla), they are still practically useful in the actual scenario of bug triaging. We demonstrate this with the exemplar triaging scenario below.

Consider a bug triager who gets 100 bug reports during the night and needs to handle them in the 10 hours of the coming day. Among the bug reports, 10 are security bugs (a reasonable proportion as evidenced by our datasets). Without any technique (bug reports are randomly ordered), averagely the triager will find one security bug report every hour, so after one hour there will be 9 security bug reports unhanded and thus available to the public. After 5 hours, 5 security bug reports are still not handled and will be available to the public.

If the triager uses a keyword-based baseline technique, with an assumed precision, recall, and F-score of 30% (considering that a keyword-based approach achieved 28% F-score on Mozilla), the triager will find 3 security bug reports in the first hour (among the first 10 bug reports that are labeled as "security-related"), so only 7 security bug reports will be left after 1 hour, and averagely $10 - (3 + (7/9) \times 4) = 3.9$ bugs will be available to the public after 5 hours.

If the triager uses SAIS, with an assumed precision, recall, and F-score of 60% (considering that SAIS achieved 61% F-score on Mozilla), the triager will find 6 security bug reports in the first hour (among the first 10 bug reports that are labeled as "security-related"), so only 4 security bug reports will be left after 1 hour, and averagely $10 - (6 + (4/9) \times 4) = 2.3$ bugs will be available to the public after 5 hours.

To sum up, in the concrete triaging scenario described above, a tool with 60% precision and recall (and 60% F-score) will reduce more than 50% of the security bug reports being available to the public after one hour or 5 hours of triaging. So although 60% F-score is far from perfect, it can be practically useful.

VI. 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. [20] 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 are evaluated on bug data sets that are not public available, so we are not able to directly compare our results with them. Gegick et al.'s approach is similar to SAIS_{mb}, one of SAIS's variant approaches investigated in our evaluation. Their approach requires manually generated start word lists and synonym lists to help the mining, and the mining process is performed by a commercial tool. We have no access to either the two lists of words or the tool. Dehl's approach [12] is similar to SAIS_{k}, another variant approach we explored in our evaluation. As some parameters (e.g., word list size or word weight threshold) are not mentioned in their paper, we are unable to reproduce their approach purely based on the paper description. Tyo [51] studied vulnerability reports from NASA dataset, and evaluated how different machine learning algorithms perform on identifying SBRs. They also performed multi-class classification to put SBRs in different security bug categories. In our work, we use semi-supervised learning to handle the sparsity of security bug reports, and further develop keyword-based refinement to address data unbalance.

According to our evaluation, SAIS outperformed SAIS_{k}, and the best machine learning algorithms indicated in Tyo's work [51] on identifying SBRs. In addition to technical novelty, our research makes two extra contributions. First, we construct a public available data set containing labeled security bug reports from open bug repositories so that the data set can be used in future research in the area. Second, our evaluation is performed with a small training set which is often the only training set available.

There have been several other research efforts on security bug reports. Zaman et al. [56] performed an empirical study on how performance bug reports and security bug reports

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

are handled in open bug repositories. Their key findings include that fixing security bug reports usually requires more experienced developers and more complex code commit, and security bug reports are usually fixed faster than other bug reports. These findings actually support our assumptions that security bug reports needs to be identified earlier and handled separately. Barth et al. [11] proposed an approach to attack open source software projects by identify the fixes of recent security bug reports. Since the distribution of new versions of software typically require a long time, it is highly likely that the attack of this fixed security bug can affect a lot of users in real world. The authors suggest that the open bug repositories to hide security bug reports as well as bug fixes until a long time after the fix. These research efforts are not on the identification of security bug reports, and they actually provide motivations to automatic identification of security bug reports.

We are also aware of some research efforts on mining the CVE database. Huang et al. [25] proposed to use text clustering to identify categories and patterns of CVE entries. Neuhaus and Zimmermann studied the trends of topics in the CVE database with topic modeling. Bozorgi et al. [13] proposed to train a prediction model from the CVE database to predict whether and when a vulnerability may be exploited. Anbalagan and Vouk [7] proposed a tool to automatically link CVE entries to bug reports in open bug repositories. Although these efforts also make use of the textual and meta data in CVE entries, they do not target on the problem studied in this paper.

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. [10], Cubranic and Murphy [15], and Lucca [19] 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 [37]. Additionally, Hooimeijer and Weimer suggested a statistics based model to predict the quality of bug reports [23].

Identification of duplicate bug reports is another subarea attracting many researchers. Runeson et al. [41] and Wang et al. [55] carried out some early works in the area based on the cosine similarities of textual information or execution information. Later, Chengnian et al. [48] [47] reported two pieces of research efforts for more precise identification of duplicate bug reports with more sophisticated mining techniques. Specifically, they also used the discriminative weighting of terms, and the weighting is performed automatically by mining the training set [48]. However, they used the weights to calculate a more precise similarity between single bug reports, while in our approach, we use a different weighting formula because we needs the weights to rank terms according whether the terms are representative for a bug category. Furthermore, we combine the term ranking approach with classification and bootstrapping.

More recently, Thung et al. [49] proposed to use semi-supervised learning to classify BRs into functional bugs and structural bugs, and finer categories. Compared with their approach which targets classification of general BRs, SAIS is specific for identifying SBRs, so we developed the CVE keyword-mining and keyword-based refinement to handle the sparsity of SBRs and data unbalance. We further evaluated SAIS and Thung et al.'s approach in the scenario of SBR identification and SAIS performs better.

C. Semi-Supervised Learning and Handling Data Imbalance

Semi-supervised learning [57] 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 [42] proposed to use semi-supervised learning for the estimation of software quality. Li et al. [31] 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 machine-learning based software engineering techniques.

There have also been a number of research efforts on learning from unbalanced software engineering data. Jing et al. [27] proposed to use dictionary learning to handle the sparseness of historical features, and class imbalance in defect prediction. Wang et al. proposed to apply various class imbalance learning techniques such as over-sampling, under-sampling, and bootstrapping to defect prediction [54] and bug detection [53].

The identification of security bug reports can be viewed as a specific instance of the automatic content tagging problem [43], which aims to add meaningful tags to contents on the Internet. The large amount of research efforts to tackle this problem mainly fall into two categories: (1) the approaches based on term weighting, ranking, and retrieval of documents using a list of terms as queries; (2) the approaches based on classification which leverages various standard or adapted machine learning techniques. The first category of approaches mainly focus on more advanced term weighting formulae [46] [33], ranking strategies [14] [52], and the adaptation of these techniques in different usage scenarios [16]. The second category of approaches, mainly focus on the selection of proper features in different usage scenarios [40] [35]. Recently, there have been some research efforts [30] [38] on combining the term ranking and classification approaches. Our work basically leverages the existing scheme of combination, and incorporated the bootstrapping process considering the small training set available for the problem.

VII. FUTURE WORKS

In the future, we plan to extend our work in the following three directions.

First, we currently use only the CVE database as our external keyword-mining set. In our observation, although it

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

enhances the final results (e.g., F-score), it may be still insufficient, especially for projects with larger variety of security keywords (e.g., Mozilla) in their relatively unprofessional BRs. In the future, we plan to collect more confirmed security bug reports from different software domains (e.g., browsers, mobile apps), so that our technique can be further improved in dealing with such projects.

Second, we are currently using statistical measurements (e.g., precision, recall) to measure different techniques. Due to the limitation of these measurements, they may not precisely measure the actual usefulness of SAIS. Therefore, we plan to reach out to BR triagers of real world projects, and have them to try out SAIS in practice, so that we can better evaluate its usefulness. Furthermore, we may compare the number of SBRs the triager can identify with SAIS during certain period, with that of other approaches.

Third, although leveraging semi-supervised learning helps to reduce the size of required training set, we still need about 100 BRs in the initial training set, and it may still too large for some smaller projects. We plan to explore the feasibility of further reducing the size of training set with more advanced techniques such as transfer learning.

VIII. Conclusion

In this paper, we proposed SAIS, an approach to automatically identify SBRs based on keyword mining, semi-supervised learning, and the CVE database.

We summarize our findings from the following three aspects: necessity of automatic SBR identification, the comparison of different techniques, and the comparison of parameters within different techniques.

Necessity. Our motivation study shows that more than half of the security bug reports are not triaged within 24 hours after their submission, which left sufficient time for potential attackers to read them and develop attacks based on them. Therefore, automatic identification of security bug reports will be very helpful for the security of software projects.

Technique-wise comparison. We developed and evaluated different categories of techniques for automatic SBR identification under the realistic setting of unknown security-relevant keywords and small training set.

- Among the investigated techniques, SAIS worked best while keyword-based labeling (SAIS_k) worked worst, because the latter one intuitively retrieved SBRs purely based on keyword matching.
- Semi-supervised learning with iterative execution outperforms supervised learning without any iteration, because SAIS's result continuously getting better with the number of iterations.
- Although CVE entries can help SAIS identify SBRs, they cannot replace project-specific SBRs when being used to train classifiers, as those SBRs importantly characterize the SBRs to label.
- As demonstrated by SAIS_{mb}, over-sampling can help improve the learning accuracy for imbalanced training sets, although the improvement is limited.

Configuration-wise comparison. Among the experimented ML algorithms, SVM outperformed Naïve Bayes and Random Forest when the trained model was integrated into SAIS's training step. Among the investigated different sizes of the mined keyword list, $size = 100$ outperformed the other size settings, because it led to the best trade-off between SAIS's precision and recall.

References

[1] A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

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

[3] Mozilla bugzilla, https://bugzilla.mozilla.org/, 2015.

[4] Redhat bugzilla, https://bugzilla.redhat.com/, 2015.

[5] Known vulnerabilities in mozilla products, https://www.mozilla.org/en-us/security/known-vulnerabilities/, 2017.

[6] Jira: Issue and project tracking software. https://www.atlassian.com/software/jira, 2018.

[7] P. Anbalagan and M. Vouk. On mining data across software repositories. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 171–174, May 2009.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 105–114, 2010.

[14] C. Carpineto, G. Romano, and V. Giannini. Improving retrieval feedback with multiple term-ranking function combination. *ACM Trans. Inf. Syst.*, 20(3):259–290, July 2002.

[15] 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.

[16] D. Das and S. Bandyopadhyay. Word to sentence level emotion tagging for bengali blogs. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, ACLShort '09, pages 149–152, 2009.

[17] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9–12):1194 – 1213, 2007.

[18] 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.

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

[20] 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.

[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] T. K. Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, 1995.

[23] 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.

[24] W. Hu, J. Chen, and Y. Qu. A self-training approach for resolving object coreference on the semantic web. In *Proceedings of the 20th International Conference on World Wide Web*, pages 87–96, 2011.

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2939132, IEEE Transactions on Dependable and Secure Computing

[25] S. Huang, H. Tang, M. Zhang, and J. Tian. Text clustering on national vulnerability database. In *Computer Engineering and Applications (ICCEA), 2010 Second International Conference on*, volume 2, pages 295–299, March 2010.
[26] L. Jiang, Z. Cai, H. Zhang, and D. Wang. Naive bayes text classifiers: a locally weighted learning approach. *J. Exp. Theor. Artif. Intell.*, 25(2):273–286, 2013.
[27] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 414–423, 2014.
[28] A. M. Kibriya, E. Frank, B. Pfahringer, and G. Holmes. Multinomial naive bayes for text categorization revisited. In *Australasian Joint Conference on Artificial Intelligence*, pages 488–499. Springer, 2004.
[29] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.
[30] M. Lan, C. Tan, J. Su, and Y. Lu. Supervised and traditional term weighting methods for automatic text categorization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(4):721–735, April 2009.
[31] 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.
[32] Y.-F. Li, J. T. Kwok, and Z.-H. Zhou. Semi-supervised learning using label mean. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 633–640, 2009.
[33] Y. Liu, H. T. Loh, and A. Sun. Imbalanced text classification: A term weighting approach. *Expert Systems with Applications*, 36(1):690 – 701, 2009.
[34] Y. Liu, S. Zhou, and Q. Chen. Discriminative deep belief networks for visual data classification. *Pattern Recognition*, 44(10–11):2287 – 2296, 2011.
[35] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *J. Mach. Learn. Res.*, 2:419–444, Mar. 2002.
[36] 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.
[37] 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.
[38] R. Nallapati. Discriminative models for information retrieval. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 64–71, 2004.
[39] C. D. Paice. An evaluation method for stemming algorithms. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, pages 42–50, 1994.
[40] H. Qu, A. L. Pietra, and S. Poon. Automated blog classification: Challenges and pitfalls. In *AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs*, pages 184–186, 2006.
[41] 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.
[42] 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.
[43] A. Sheth, C. Bertram, D. Avant, B. Hammond, K. Kochut, and Y. Warke. Managing semantic content for the web. *Internet Computing, IEEE*, 6(4):80–87, Jul 2002.
[44] C. Silva and B. Ribeiro. The importance of stop word removal on recall values in text categorization. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 3, pages 1661–1666 vol.3, July 2003.
[45] M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In *Australasian joint conference on artificial intelligence*, pages 1015–1021. Springer, 2006.
[46] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W.-C. Lee, and C. L. Giles. Real-time automatic tag recommendation. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 515–522, 2008.
[47] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, pages 253–262, 2011.
[48] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE (1)*, pages 45–54, 2010.
[49] F. Thung, X.-B. D. Le, and D. Lo. Active semi-supervised defect categorization. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 60–70. IEEE Press, 2015.
[50] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *J. Mach. Learn. Res.*, 2:45–66, 2002.
[51] J. P. Tyo. Empirical analysis and automated classification of security bug reports. 2016.
[52] J. Wang and B. D. Davison. Explorations in tag suggestion and query expansion. In *Proceedings of the 2008 ACM Workshop on Search in Social Media*, SSM '08, pages 43–50, 2008.
[53] S. WANG, L. L. MINKU, and X. YAO. Online class imbalance learning and its applications in fault detection. *International Journal of Computational Intelligence and Applications*, 12(04):1340001, 2013.
[54] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *Reliability, IEEE Transactions on*, 62(2):434–443, June 2013.
[55] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information. In *proceedings of the International Conference on Software Engineering*, 2008.
[56] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: A case study on firefox. In *MSR*, pages 93–102, 2011.
[57] X. Zhu. Semi-supervised learning. In *Encyclopedia of Machine Learning*, pages 892–897. 2010.
[58] 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.

1545-5971 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.