A Machine Learning Approach for Vulnerability Curation

Yang Chen
Veracode
ychen@veracode.com

Andrew E. Santosa
Veracode
asantosa@veracode.com

Ang Ming Yi
Veracode
mang@veracode.com

Asankhaya Sharma
Veracode
asharma@veracode.com

David Lo
Singapore Management University
davidlo@smu.edu.sg

ABSTRACT

Central to software composition analysis is a database of vulnerabilities of open-source libraries. Security researchers curate this database from various data sources, including national vulnerability database, bug tracking systems, commits, and mailing lists. In this article, we report the design and implementation of a machine learning system to help the curation by automatically predicting the vulnerability-relatedness of each data item. Our system supports a complete pipeline from data collection, model training and prediction, to the validation of new models before deployment. It is executed iteratively to generate better models as new input data become available. It is enhanced by a self-training process that significantly and automatically increases the size of the training dataset, opportunistically maximizing the improvement in the models quality at each iteration. We devised new *deployment stability* metric to evaluate the quality of the new models before deployment into production. We experimentally show the improvement in the performance of the models in one iteration, with 27.59% maximum PR AUC improvements, and the effectiveness of self-training, with 10.50% PR AUC improvement. We discover that the addition of the features of the corresponding commits to the features of issues/pull requests improve the precision for the recalls that matter, and that there is no uniform ordering of word2vec parameters sensitivity across data sources. The deployment stability metric has also helped to discover an error.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

application security, open-source software, machine learning, classifiers ensemble, self-training

ACM Reference Format:
Yang Chen, Andrew E. Santosa, Ang Ming Yi, Asankhaya Sharma, and David Lo. 2019. A Machine Learning Approach for Vulnerability Curation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

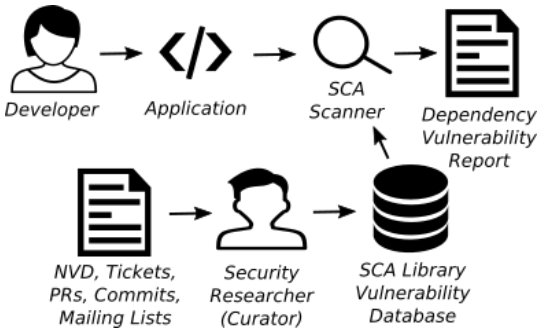

Figure 1: Software Composition Analysis

1 INTRODUCTION

Modern information infrastructure relies heavily on open-source libraries provided by Maven central, npmjs.com, PyPI, etc. Unfortunately, they may contain vulnerabilities that compromise the security of the applications using them. *Software Composition Analysis* (*SCA*) automatically reports vulnerabilities in the open-source libraries used by an application, so that they can be remediated by the application's developers. SCA is widely used in industry, with commercial offerings from various companies [1, 6, 7, 9]. Figure 1 shows its typical workflow. A SCA system scans an application to report the vulnerable dependencies (libraries) used, using the data from a library vulnerability database. Security researchers curate this database using various sources, including *National Vulnerability Database* (*NVD*) [3] which lists vulnerabilities with *Common Vulnerabilities and Exposure* (*CVE*) ids, however, according to SourceClear data [8], close to 50% of vulnerabilities in open-source software are not disclosed publicly in the NVD. Therefore, other sources are also used. Veracode SCA, in particular, uses issue reports, *pull requests* (*PRs*), and commit messages, as often the vulnerabilities are reported and/or fixed through these media without being identified in the NVD. The design of a state-of-the-art SCA product is discussed in an article [13].

In this article, we report the work done at Veracode for the design and implementation of a machine learning system to help with the curation of the library vulnerability database. This system predicts each input data item as either *vulnerability related* or not, with the result used as a recommendation for a team of security researchers who curate the database. We obtain input data from a variety of data sources, and we train one model for each data source, using

a stacking ensemble of classifiers, which are reported to perform well for unbalanced datasets and natural language processing in the literature [40]. Our system provides a complete iterative pipeline from data collection, model training and prediction, to the validation of new models before deployment. We use scikit-learn 0.20 [5] and word2vec [19] from gensim 3.6.0 [2] for word embedding.

Existing approach [40] uses manually-labeled data from the curated database for training new models, however, in this way, a large amount of data remain unused. This unused set includes data items that fail our initial keyword-based filtering, and data items that are predicted as not related to vulnerability by the current production models. To maximize the utilization of these data items, our system performs *semi-supervised* learning [10], in particular *self-training* [20], in which we re-apply our production models to the unlabeled data in order to significantly increase the amount of labeled data for training new models. Self-training significantly and automatically increases the size of labeled dataset used in training new models, opportunistically maximizing the improvement in the quality of models at each iteration of our pipeline.

Our pipeline produces a suite of new models at each iteration. Before deploying these models into production, we need to guarantee that they will perform better than the current production models. For this, we conduct the following two validations:

- *Performance validation.* We deploy a new model into production only when the new model has a better performance than the production model it is replacing.
- *Stability validation.* We check for each new model that its true positives and true negatives do not vary too much to that of the production model. To measure this, we compute a new metric called *deployment stability*, which is the sum of the numbers of the common true positives and common true negatives of the new and production models, divided by the total number of data items.

In summary, our contribution is on the design and implementation of a machine learning system to support manual curation of a library vulnerability database. Our system supports the whole pipeline of machine learning, including data collection, training, prediction, and validation. The pipeline is executed iteratively to improve the model continually, and has been deployed in production at Veracode. We use the k-fold stacking ensemble of classifiers from existing work [40], but we advance further in these directions:

(1) From the technical perspective, we designed and implemented a complete system which uses self-training [20]. In our experiment, self-training improves PR AUC by 10.50%.
(2) We introduce a complete model validation methodology with a novel deployment stability metric.
(3) We add emails and *reserved* CVEs (NVD entries that are not yet confirmed as vulnerabilities) into the suite of input data sources, and we consider more commit features. The existing work [40] uses only commit message as the feature due to concerns about accuracy, however, the literature indicates that the quality of the developer significantly affects the quality of the code [18]. We hypothesize that the developers more familiar with security issues are more likely to fix a vulnerability and therefore include Github user name into our commits feature set. We also in addition include patched files paths, and the commit patch itself. For the recall of 0.72 reported by the existing work [40] we more than doubled the precision from 0.34 to 0.69.

In this article, we detail our pipeline, evaluate its performance for one iteration as a case study, with PR AUC improvements at most 27.59%, and experimentally demonstrate the effectiveness of self-training for a data subset, with 10.50% PR AUC improvement. We discover that the addition of the features of the corresponding commits to the features of Github issues/PRs improve the precision for the recalls that matter, and that there is no uniform ordering of word2vec parameters sensitivity across data sources. We also show how our deployment stability metric is useful to discover an error.

We provide an overview of our whole system in Section 2. We discuss our data collection and feature engineering in Section 3, and our model training including self-training in Section 4. We discuss our validation methodologies in Section 5, and provide an evaluation of our approach in Section 6. We present related work in Section 7, and conclude our paper in Section 8.

2 OVERVIEW

Figure 2 shows the data flow diagram (DFD) of our system. It is an iterative pipeline centering on the use of machine learning models to predict the labels of new data (B) for helping the manual labeling by security researchers. The labeled data are in turn used to train new models using supervised learning (C), after which the resulting models are validated against existing production models (D) to ensure that they are better. The newly-validated models are then deployed in production for predicting the labels of newer data (B), as well as being used in a self-training subsystem (E), which automatically labels residual data not manually labeled to increase the amount of labeled data used in the next training iteration, opportunistically maximizing the performance of the new models at each iteration. Each iteration of the pipeline accumulates more labeled data and gradually improves the performance of the models.

We collect data from six sources (A) including Jira tickets, Bugzilla reports, Github issues and PRs, Github commits, emails from relevant mailing lists, and reserved CVEs from NVD. Here, the data are highly unbalanced, where only a small proportion of the input data are vulnerability-related. To mitigate this issue, we use keyword-based filtering to select data which are more likely to be related to security (detailed in Section 3.1).

To bootstrap the whole process, we created a suite of initial models. Here, the security researchers reviewed the filtered-in data and labeled them as vulnerability-related or not, denoted in Figure 2 using the dotted line (bootstrap 1), directly producing the labeled data. On these data, we used supervised learning to obtain the initial models (bootstrap 2), one for each data source. In this initial phase, we directly used these models in production (bootstrap 3).

The security researchers review only those data that are predicted by the models as vulnerability-related and label them. Although we may miss some true vulnerability-related data due to the imprecision of the production models, but with using the prediction results, we reduce the manual review and labeling time. We then use the new labeled data to generate new models using the same learning approach (C). We note that our system has been gradually evolving over time, and at present we use a set of known-good fixed

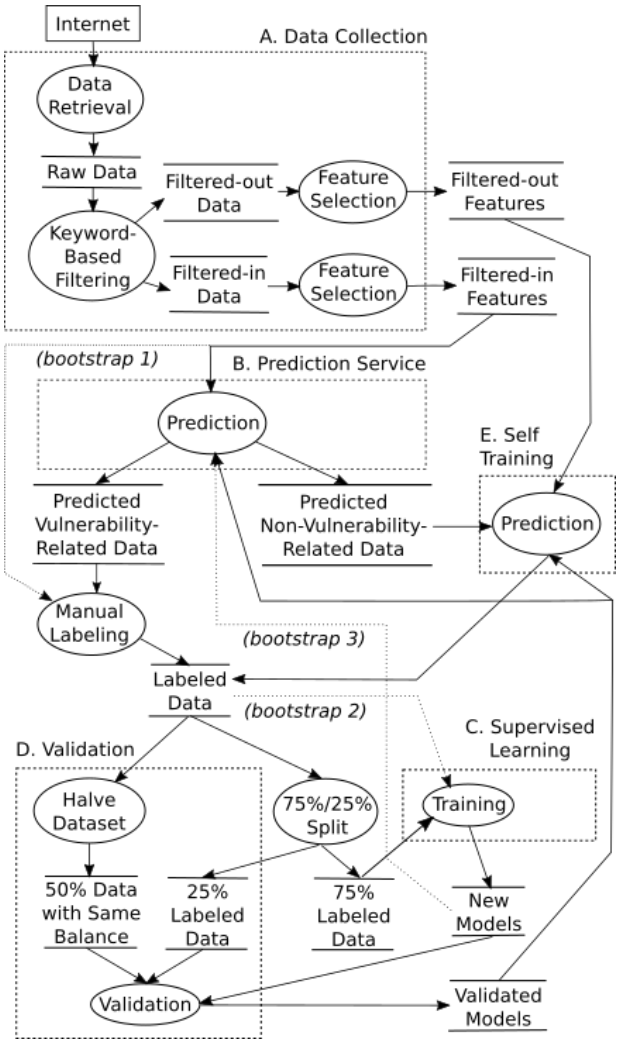

Figure 2: System Pipeline Data Flow Diagram

hyper-parameters in the training process, such as we no longer reserve test data for hyper-parameter tuning. Once there are new models, we validate them against the production models (D), to ensure that the new models to be deployed actually perform better than the production models.

Manually labeling data for training is expensive, and hence there is a large amount of data that cannot be manually labeled. The first dataset consists of the data that are predicted as vulnerability-unrelated by the production models, and second dataset consists of the data that fail the keyword-based filtering. Their sizes can be much larger than the labeled data, particularly for commit data. Such is the situation where semi-supervised learning [10] can be applied. In particular, here we use a class of semi-supervised learning called self-training [20] to automatically label the data from both sources (E) for training new models.

Table 1: Dataset Sizes

Data Source	Collected Size	No. Positive	Positive Ratio
Jira Tickets	17,427	911	5.23%
Bugzilla Reports	39,801	20,250	50.88%
Github Issues/PRs	50,895	5,147	10.11%
Commits	157,450	5,181	3.29%
Emails	20,832	11,756	56.43%
Reserved CVEs	31,056	7,245	23.33%

3 DATA COLLECTION

3.1 Data Sources

Our data sources are those that the security researchers use to find information on vulnerabilities in their day-to-day tasks. These include both textual data sources as well as code sources. The data sources are Jira tickets, Bugzilla reports, Github issues and PRs, Github commits, mailing lists, and *reserved* CVEs (a CVE entry with an id, but unconfirmed as a vulnerability) from NVD. Table 1 summarizes of the data we collected for each data source for the first iteration of our pipeline, including the total collected data sizes (2nd column), the positive labeling counts, which is the number of data finally labeled as vulnerability-related (3rd column), and the ratios of the positively-labeled data (4th column). Although we collect the same set of features from Jira Tickets, Bugzilla reports, and Github Issues/PRs, we separate them as different sources as they have different characteristics. Table 1 also shows that Bugzilla reports are likely to be vulnerability related when compared to the other two. A possible reason is that Bugzilla reports are specifically about bugs which include vulnerabilities, whereas Jira tickets and Github issues/PRs in addition also discuss feature requests and implementations. The notion of issue severity and the typical number of attachments may also differ between the three.

At the time of writing we have collected Bugzilla reports, Jira tickets, and Github issues/PRs from 24 Bugzilla servers, 82 Jira servers, and 20,447 open-source library repositories hosted on Github. We build our list of servers and repositories based on the open-source libraries that we frequently encounter in customer scans. The set Bugzilla and Jira servers spans the areas of software security, library, platform and infrastructure, and utilities. Due to lack of space, we list only their representatives in the first two columns of Table 2. The commit data we use are from the same 20,447 Github repositories. Since one Github issue or PR may be related with multiple commits, the number of commit data is much more than the number of issues or PRs (see Table 1). We use the commit data both to improve the prediction results for the Github issue and PR models, as well as independently to train models used in predicting vulnerability-related commits. To obtain email data, we subscribe to 18 open-source software projects mailing list. We only list some of them in the third column of Table 2 due to lack of space. For reserved CVEs, we retrieve the data from NVD.

Some data may not provide any useful information. The data for most of our sources are highly unbalanced, with a large proportion of them to be irrelevant to security vulnerability (see the fourth column of Table 1). For all of the data sources except emails and

reserved CVEs, we perform a keyword-based filtering with security-related keywords, including *security*, *advisory*, *authorized*, *NVD*, etc. Only data items whose text contains any of the keywords are used as inputs to the next stage. Those that do not contain any of the keyword are to be used for self training. The numbers shown in the second column include those that pass as well as fail the filtering. We do not apply this filtering to the email data since the ratio of imbalance is low. There are also other possible ways of filtering, such as based on the timeliness of the data item, however, such complex filterings are unnecessary.

3.2 Feature Engineering and Selection

We show the features of the data that we use in Table 3. Most of the features are textual and used as inputs to word2vec (see Section 4.1), however, there are some numerical features that are directly included in the input vectors. From Jira tickets and Bugzilla reports, we extract the same sets of features. We use the Github issue and PR data to train two different models, depending on whether a Github issue or PR has associated commits or not. We name the data source without commits as *Github_Basic*, and the data source with commits as *Github_Combined*. For Github_Basic, we extract the same sets of features as for Jira tickets and Bugzilla reports. For Github_Combined, we in addition extract commit message, Github user name, patched files paths, patch lines added, and patch lines deleted (we treat program code as text). For Github issues and PRs, we use the Github_Basic model for prediction when the data has no commits information, otherwise we use the Github_Combined model. Section 6.1 shows that Github_Combined has a better precision than Github_Basic for the recalls that matter. For reserved CVEs, each item typically only contains a number of web page URLs. Although a reserved CVE is more relevant to vulnerabilities than some other types of sources, the web pages may not describe a vulnerability, and therefore we select their content as a feature.

Our feature selection is based on our experiences on whether the feature actually helps in improving performance. For instance, we avoid using date features since they reduce the prediction precision. At one time we observed a low prediction performance due to Jira tickets, Bugzilla reports, Github issues and PRs were predicted as positive (vulnerability-related) with a newly-trained suite of models. If a date value was of the year 2019, the prediction result would be positive, and if we changed its value to 2017 or 2018, the prediction result would mostly be negative. Intuitively, the date features should not be as relevant to vulnerability compared to other features, and we thus exclude them as features.

4 MODEL TRAINING

4.1 Word Embedding

Most of the data features that we consider are textual (e.g., commit messages and commit patches for commits data; see Table 3), models training requires input vectors. To map a textual feature to an input feature, we use word2vec [19], a popular approach for word embedding. Here we use the implementation of word2vec in gensim 3.6.0. Before inputting to word2vec, we clean each textual feature by removing all non-alphabetic characters, except for commit patches, where we remove brackets and comment delimiters such as #, //, /**/ from the code. Word2vec also requires a *corpus* for each data source. Generally, larger corpus results in better vector model. For non-commit data sources, we train word2vec model based on the collected data alone, however, for commits, we in addition expand our corpus by using an existing unlabeled extra commit data already collected from various sources such as Github. The number of these extra commit data items is 3,212,690.

Word2vec employs neural network to learn vector representation of words in one of two ways, either by predicting central word from context word, called *continuous bag of words* (*CBOW*) (which is the approach that we select), or by predicting context words from central word, called continuous *skip-gram*. *Window size*, *minimal count* and *vector size* are three important parameters to train word2vec models. *Window size* is the maximum distance between the current and predicted word within a sentence. *Minimal count* is the count to ignore all words if their total frequency is lower than this value. *Vector size* is the vector size for one word. We list the parameter values in Table 4 by data source, as the result of a tuning process. We show in Section 6.3 that the performance of the model is not significantly sensitive to any one of the parameters in Table 4. The output of word2vec is a mapping of each word into a vector. Given a textual feature, we compute a vector average of all the vectors that the words are mapped into. The vector averages of the textual features are combined with numerical values of non-textual features to construct a composite vector that is input to a learner. We train a word2vec model separately for each textual feature as each textual feature may have unique characteristics.

1 for each $L \in LabeledDataSources$:
2 for $i := 1, \ldots, k$:
3 $L_i :=$ Random subset of L of size $|L|/k$
  s.t. $\forall j \cdot 1 \leq j < i \Rightarrow L_i \cap L_j = \emptyset$
4 for $i := 1, \ldots, k$:
5 $L_{test}, L_{train} := L_i, L - L_i$
6 for each $t \in ClassifierTypes$:
7 $basicModel_L^{t,i} := t.fit(L_{train}, labels(L_{train}))$
8 $\overline{v}_t := basicModel_L^{t,i}.predict(L_{test})$
9 $M_i := (\overline{v}_{t_1} \ldots \overline{v}_{t_6})$ s.t. $[t_1, \ldots, t_6] = ClassifierTypes$
10 $M := \begin{pmatrix} M_1 \\ \vdots \\ M_k \end{pmatrix}$
11 $LRModel_L := logisticRegression.fit(M, labels(L))$

Figure 3: k-Fold Stacking Ensemble Pseudocode. The input is *LabeledDataSources*, which is the set of labeled datasets, where each dataset corresponds to a data source. *ClassifierTypes* is the sequence of six basic classifier types: RF, Gaussian Naive Bayes, k-NN, SVM, gradient boosting, and AdaBoost. The procedure outputs a sequence of $LRModel_L$ for all $L \in LabeledDataSources$ and a sequence of $basicModel_L^{t,i}$ for all $L \in LabeledDataSources$, $t \in ClassifierTypes$, and $1 \leq i \leq k$.

Table 2: Example Data Sources

Bugzilla Servers	Jira Servers	Mailing Lists
https://bugzilla.suse.com	https://issues.apache.org/jira	announce@apache.org
https://bugzilla.mozilla.com	https://jira.sakaiproject.org	dev@jspwiki.apache.org
https://bugs.eclipse.org	https://java.net/jira	security@apache.org
https://bugzilla.novell.com	https://jira.sonarsource.com	replies@oracle-mail.com
https://bugs.webkit.org	https://issues.jboss.org	rubyonrails-security@googlegroups.com
https://bugzilla.gnome.org	https://0xdata.atlassian.net	security@suse.de
https://bugs.kde.org	https://jira.mongodb.org	debian-security-announce@lists.debian.org

Table 3: Features per Data Source. $*$ = non-textual numeric feature. We input only the textual features to word2vec.

Data Source	Features Used
Jira Tickets, Bugzilla Reports, Github_Basic	title, body, source, severity, state, labels, comments, attachment number*, comment number*
Github _Combined	Github_Basic features, commit message, user name, patched files paths, patch lines added, patch lines deleted
Commits	commit message, user name, patched files paths, patch lines added, patch lines deleted
Emails	email subject, body, sender, and status
Reserved CVEs	URL, content of the URL link, URL length*, content length*

Table 4: Word2vec Parameters

Data Source	Window Size	Minimal Count	Vector Size
Jira Tickets	20	50	200
Bugzilla Reports	20	50	200
Github Issues/PRs	20	50	200
Commits	10	50	200
Emails	10	20	200
Reserved CVEs	10	10	200

4.2 Training

We use supervised learning ((C) of Figure 2) to build vulnerability-relatedness prediction models using scikit-learn 0.20 [5], one model for each data source. For this task, there are different types of possible classifiers to use, such as *random forest* (*RF*) or *support vector machine* (*SVM*), but here we follow previous work [40] to use an ensemble of classifiers which has a better performance than each individual classifier. Our *stacking* ensemble consists of six *basic* classifiers, which are RF, *Gaussian Naive Bayes*, *k-nearest neighbor* (*k-NN*), SVM, *gradient boosting*, and *AdaBoost*, and employs a logistic regression as the meta learner.

We use the value 1 to label a vulnerability-related data item, and 0 otherwise. Since the labels that we use to fit the model are represented by values of 0 or 1, when a model is used to predict a vulnerability relatedness of a data item, the output is therefore a value from 0 to 1, which we call the *probability of vulnerability relatedness* (*PVR*). A PVR is a degree of confidence on whether the data item is related to a vulnerability.

We randomly select 75% of the labeled dataset for model training and the remaining 25% for performance testing (see Section 5.1). Figure 3 shows our *k-fold stacking ensemble* [40] pseudocode for training. Here, we further randomly split the 75% labeled data part into k disjoint subsets (Lines 2–3 of Figure 3). (We use $k = 10$, which we discovered to perform better than $k = 12$ used in the literature [40].) We choose one of the k subsets as the testing set, and the union of the remaining k-1 subsets as the training set (Line 5 of Figure 3). For each one of the k such split, and for each of the six basic classifiers that we use, we fit a model on the training set (Line 7), and then apply the model for prediction to the testing set (Line 8). Now, for each data item in the testing set, the prediction outputs a PVR from 0 to 1. Since we apply the model to the whole testing dataset, the result of the prediction is a vector of PVRs whose length is the size $|L|/k$ of the testing set. In Figure 3, this is the column vector $\overline{v}_t$ (Line 8). We combine all the column vectors for all basic classifiers into a $(|L|/k) \times 6$ matrix M_i (Line 9). After performing k fittings and predictions for a basic classifier, we obtain a $|L| \times 6$ matrix M whose elements are PVRs. Line 10 of Figure 3 shows the construction of matrix M from $M_1, \ldots, M_k$. We use this matrix, and the column vector of the labels of each data item (denoted as *labels*(L) in Figure 3) as inputs to a logistic regression to generate a model $LRModel_L$ (Line 11 of Figure 3). We generate one such model for each data source $L \in LabeledDataSource$, and this logistic regression model combined with all of the $6 \times k$ basic classifier models are the outputs of the algorithm. We call this output suite as an *ensemble* model, or simply a *model*. We initially trained new models for all data sources once a week, however, we discovered that their performance does not change significantly on a weekly basis, and therefore we presently only train new models monthly.

When using the models for prediction, we apply a threshold on the PVR called *PVR threshold* to predict if the data item is vulnerability-related. Given PVR threshold τ, when a PVR of a data item is strictly greater than τ, the data item is predicted as vulnerability-related (positive), otherwise it is predicted as vulnerability-unrelated (negative). This prediction is given to the security researchers to finally label the data item.

4.3 Self-Training

Our security researchers only label the data predicted vulnerability-related (see Section 2). Besides these manually-labeled data, there are a large amount of unlabeled data which are made up of two parts. One is the dataset that fail the keyword-based filtering (*dataset A*), and another is the dataset predicted as unrelated to vulnerability by the current production model (*dataset B*). Table 5 shows the labeled and unlabeled data sizes for each data source. The existence

Table 5: Labeled and Unlabeled Datasets Sizes

Data Source	Collected Data Size	Labeled Data Size	Unlabeled Data Size
Jira Tickets	17,427	13,028	4,399
Bugzilla Reports	39,801	22,553	17,253
Github Issues/PRs	50,895	17,230	33,665
Commits	157,450	22,856	134,594
Emails	20,832	16,573	4,259
Reserved CVEs	31,056	18,399	12,657

of a relatively large amount of unlabeled data is a typical scenario where semi-supervised learning [10] is applicable, especially for the commit data source where the number of unlabeled data items is about six times that of labeled data items. Here we label the unlabeled data automatically, and use them for training. This is self-training, a widely-used technique in semi-supervised learning (see Section 7.3).

For the automated labeling, we first need to obtain the PVRs of the datasets A and B. For dataset A, we apply the production models to compute their PVRs, and for dataset B, we already have their PVRs computed by the production models, but here we use special PVR thresholds for their labeling. We set a *high* PVR threshold τ_h and a *low* PVR threshold τ_l. Given a threshold τ currently used for prediction in production, we have that $\tau_l < \tau < \tau_h$. If the PVR of a data item is greater than τ_h, we label it as vulnerability-related. If the PVR is less than τ_l, we label the data item as a vulnerability-unrelated. We do not label the data item if its prediction score is between the high threshold and low threshold. We determine the values for τ_h and τ_l based on our previous experiences for different data sources, where the chosen values we expect to result in high prediction accuracy. We note that in this way, the dataset labeled as vulnerability-unrelated by the self-training mechanism is a subset of the dataset predicted as such by the current production model, as $\tau_l < \tau$. We combine the automatically-labeled dataset with the manually-labeled dataset for use in training our models for the next iteration. We use self-training data labels for model training only and not inputting them to the security research team since the labels may have a low precision.

5 MODEL VALIDATION

5.1 Performance Validation

The first validation step ((D) in Figure 2) ensures that any new model replacing a production model has a better performance. Note that here we have a labeled dataset that keeps increasing in size for every iteration of our pipeline. We randomly select 75% of this dataset for training, and use 25% of it for validation (see Section 4.2). Using k-fold validation instead of 75%–25% split is doable, however, whether further split on top of the k-fold split already done on the 75% part for the model training (Section 4.2) would improve performance significantly is a subject of future research. For validation we collect *precision* and *recall* metrics, where:

$$precision = \frac{|TP|}{|TP| + |FP|} \qquad recall = \frac{|TP|}{|TP| + |FN|}$$

with TP, FP, and FN are respectively the sets of true positives, false positives, and false negatives. Precision is the ratio of true positives vs. all predicted positives. A high precision saves manual work in removing false positives. Recall indicates the coverage to identify all truly vulnerability-related items. The higher the recall, the more likely any real vulnerability-related item gets predicted as such (less false negatives). Here our models are not optimized on one of the metrics, as both precision and recall are important.

Given the same model and dataset, each PVR threshold (0.01, 0.02, . . ., 0.99) fixes a precision and recall pair. Using each pair as a coordinate, we plot a *precision-recall* (*PR*) curve in a Cartesian system with the recall values as the abscissa and precision values as the ordinate. Higher curve indicates better performance. This we use to compare the performance of a new model with another model already deployed in production. There is an alternative diagnostic tool called *receiver operating characteristic* (*ROC*) curve, however, ROC is not suitable to us since our data have a high ratio of imbalance [12] (see Table 1).

5.2 Stability Validation

The second validation step tests the commonality of the prediction results of the new model versus those of the production model. Reports that are too different may indicate problems with the new model. For this, we build a test dataset for each data source from half of the positively-labeled data, and half of the negatively-labeled data, such that the test dataset has half the size of the labeled data, but with the same label balance. For the production model, we use the PVR threshold value that is used in the deployment, and for the new model, we select a threshold based on model performance report, where both the precision and recall are higher than those of the production model (when the new model is deployed, this will be the new PVR threshold used in production). Here we define and use a new metric called *deployment stability* as follows:

$$deployment\ stability = \frac{|TP_n \cap TP_p| + |TN_n \cap TN_p|}{test\ dataset\ size}$$

where TP_n and TP_p refer to respectively the true positives of the new and the production models, and TN_n and TN_p refer to the true negatives of the new and production models, respectively. This metric measures the coverage of the correct prediction in both models, encompassing both positive and negative data. A higher value indicates less risk in deploying the new model as both the new and the production models agree on more of the predictions.

6 EVALUATION

6.1 Models Deployment Case Study

Here we use the testing methodologies of Section 5 for illustrating the performance improvements in an iteration. The case study is from the initial iteration of our pipeline when we introduced self-training into the system. This iteration improves the models performance for most of the data sources. Here, we use production models that are trained using a set of manually-labeled data only. The new models are on the other hand, trained using both the manually-labeled and the self-training-labeled data from our data sources introduced in Section 3. The manually-labeled dataset used to generate the production models is a subset of the manually-labeled dataset for training the new models.

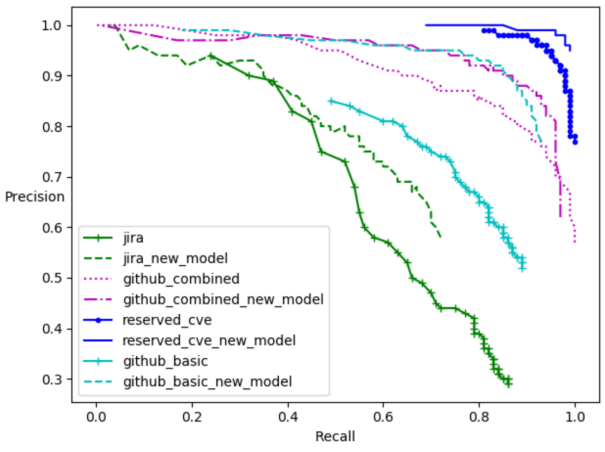

Figure 4: PR Curve for Jira, Github_Combined, Github_Basic, and Reserved CVEs

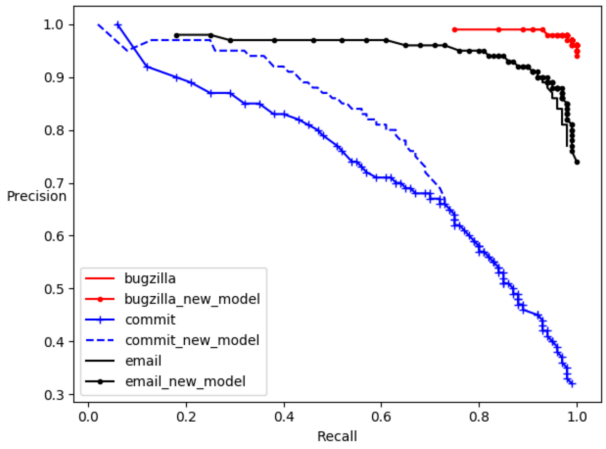

Figure 5: PR Curve for Bugzilla, Email, Commit

Performance Validation. Here we use the performance validation technique of Section 5.1 which employs 25% of the available labeled data. Figures 4 and 5 combined compare the PR curves for all the data sources (we separate them into two figures for clarity). In both figures, each data source has two curves: one for the new model and another for the production model. For each data source, the curve for the new model is mostly above that of the production model. Table 6 shows for each data source how much the new model improves the area under curve of the PR curves (PR AUC) for the recalls where we have precision data. The data show no improvement in PR AUCs for Bugzilla reports, but they show improvements of up to 27.59% for Github_Basic. For this case study, the new models improve the performance of the production models.

The PR curves of Figures 4 and 5 show differences in the performance across data sources. Although we use the same set of features for Jira tickets, Bugzilla reports, and Github_Basic (see Table 1), their curves show significant differences. Assuming our feature selection already covers the most informative parts of these data sources, we can conclude that the differences are mainly due to the actual proportion of the vulnerability-related data in these

Table 6: New Models Improvement over Production Models

Data Source	Recall Range	% PR AUC Inc.
Jira Tickets	0.24–0.72	8.50
Bugzilla Reports	0.90–0.94	0.00
Github_Basic	0.49–0.89	27.59
Github_Combined	0.01–0.97	2.88
Commits	0.06–0.73	8.01
Emails	0.92–0.98	0.95
Reserved CVEs	0.81–0.99	2.52

Table 7: Github_Combined Improvement over Github_Basic

Recall Range	% PR AUC Increase
0.18–0.93	-0.40
0.80–0.93	0.34

Table 8: Stability Validation Dataset Sizes

Data Source	50% of Positively Labeled	50% of Negatively Labeled	Total Size
Jira Tickets	455	6,059	6,514
Bugzilla Reports	10,125	1,151	11,276
Github Issues/PRs	2,573	6,042	8,615
Commits	2,590	8,838	11,428
Emails	5,878	2,408	8,286
Reserved CVEs	3,622	5,577	9,199

sources. Bugzilla reports and reserved CVEs, for example, have very high precision for all recalls; this is because the input data themselves are mostly vulnerability-related (see Section 3.1). Figure 5 also shows a markedly good performance of the email models. Recall from Section 3.1 that there is already a high proportion of vulnerability-related data within the email dataset such that we do not apply keyword-based filtering for it (irrelevant emails in the dataset include release announcements, etc.). Therefore, we see that the more the vulnerability-unrelated data in the input, the lower the prediction quality. Given the same recall, Github_Combined has a better precision than Github_Basic for the production model, however, the result is not as straightforward for the new model. Table 7 shows the PR AUC increase for the Github_Combined vs. Github_Basic PR curves. Overall, for all of the recall values for which we have prediction data (0.18–0.93), Github_Combined has a worse performance; however, for the recall ≥ 0.80, it has a better precision than Github_Basic. We discovered that **addition of commit features improves the precision for the recalls that matter**. A possible reason is that the commit messages and code comments in security-related commits have characteristics that are useful for the identification of vulnerability-related issues/PRs.

Stability Validation. Here we apply the validation method of Section 5.2. Recall that the test is performed on half of the labeled data for each data source, chosen such that they have the same balance of positives and negatives with the original dataset. Table 8 shows the dataset size details. We input this same test dataset to both the production model and the new model. Table 9 shows the

Table 9: Stability Validation Results

Data Source	Deployment Stability
Jira Tickets	0.93
Bugzilla Reports	0.93
Github_Basic	0.93
Github_Combined	0.77
Commits	0.88
Emails	0.91
Reserved CVEs	0.73

Table 10: Dataset Sizes for Self-Training Experiment

	Labeled	Unlabeled	Total
Positive	3,989	849	4,838
Negative	16,011	44,672	60,683
Total	20,000	45,701	65,701

deployment stability results. We observe that most of the deployment stability values are around 0.9 which means in most cases, both the production model and the new model agree on 90% of the prediction results, so there is not too much risk to deploy the new model. Github_Combined's deployment stability at 0.77 is lower than most others. This indicates that the new model behaves somewhat differently to the production model on the same input dataset. The likely cause of this is the change in the input data for training, where the new models are trained with more training data, including self-training data. However, although the deployment stability is low, in this case the new model has a much better quality (see Figure 4), and we still replace the current production model with the new model. Reserved CVEs' deployment stability at 0.73 is also low. We discovered that both models consistently mispredict some amount of positive data as negative. In this case, we do not have a confidence to deploy the new model into production. We discovered the reason is that in this initial iteration, there is an error in our implementation where the features that are used for Reserved CVEs model training become reordered in the prediction stage. This demonstrates that **the deployment stability metric has been effective in helping to discover an error**.

6.2 Self-Training Experiment

In the case study above, the performance and stability results can be attributed to either the increasing amount of the manually-labeled data used, or the introduced self-training-labeled data. Here instead we present an experiment where the self-training-labeled commit data alone is used to increase the amount of labeled commit data. For this experiment, we use Github commit data alone. The sizes of the datasets that we use are shown in Table 10. The second column shows the numbers of positive and negative data for the 20,000 manually-labeled commits dataset. The third column shows the sizes of the unlabeled datasets, divided into their positive and negative labelings by self-training. Again, this situation is suitable for the application of self-training due to the disproportionately larger amount of the unlabeled data.

We first train a k-fold stacking ensemble model (see Section 4.2) in a supervised manner and using the 20,000 pre-labeled data. For labeling, we use PVR low and high thresholds (see Section 4.3)

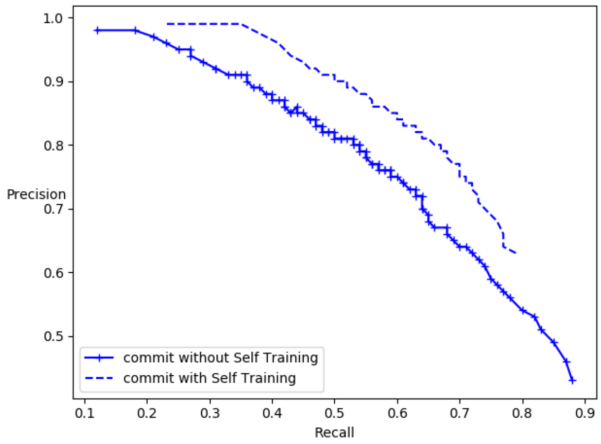

Figure 6: PR Curve for Self-Training Experiment

to predict whether a particular commit with its PVR is a positive (vulnerability-related) or a negative (vulnerability-unrelated). For the low and high thresholds, we use the values 0.22 and 0.88, respectively. 0.22 and 0.88 are the thresholds such that we can use our model to decide that a commit is negative, and respectively positive, with the high precisions of 0.93 and 0.91, respectively. We next use the supervised training model to label the unlabeled data, and add them into our labeled dataset, resulting in total 65,701 labeled commit data. Using these data, we train another k-fold stacking ensemble model. Figure 6 shows the PR curves of both models. For any of the recall in the range 0.23 to 0.79 with complete precision data, **the model with self-training consistently has a better precision**, with 10.50% PR AUC improvement of the PR curves. This result demonstrates the effectiveness of self-learning for commits data.

6.3 Sensitivity to Word2vec Parameters

Here we measure the sensitivity of the model's performance to the values of the word2vec parameters (see Section 4.1). We experiment using Jira tickets, Bugzilla reports, and commits data sources. To compute the sensitivity, we use the parameter values of Table 4 as a baseline. From the baseline, we change 50% of the value of one parameter in both positive and negative directions, while fixing the values of the other parameters, and we record the performance of the resulting models (one for each change in the positive or negative direction). As the performance measure, we use the PR AUC of the trained models for a uniform recall range for each data source where we have precision data available (0.1–0.61 for Jira tickets, 0.88–1.0 for Bugzilla reports, and 0.13–0.67 for commits). The sensitivity is computed as the percentage of the absolute change in the PR AUC for each percentage of absolute change in the input value, which is then averaged for both the positive and negative changes. From the results in Table 11, we see that the performance for Jira tickets is most sensitive to the change in window size, minimal count, and vector size, in that order. The prediction performance for Bugzilla reports is already very high (see Figure 5) and is therefore hard to change by tuning the word2vec parameters. For commits, the result is different to that of Jira tickets, where the performance

Table 11: Sensitivity to Word2vec Parameters

Data Source	Parameter	Sensitivity
Jira Tickets	Window Size	0.102
	Minimal Count	0.056
	Vector Size	0.016
Bugzilla Reports	Window Size	0.003
	Minimal Count	0.002
	Vector Size	0.002
Commits	Window Size	0.003
	Minimal Count	0.016
	Vector Size	0.007

is most sensitive to minimal count, followed by vector size, and lastly window size. Our results indicate that **there is no uniform ordering of sensitivity across data sources**.

7 RELATED WORK

7.1 Vulnerability Identification

There is a large number of works in the area of vulnerability *discovery*, where the problem is in the discovery of *unknown* vulnerabilities, with a variety of approaches including static and dynamic program analyses, and even machine learning [4, 11, 14, 15, 30, 35, 37, 39], as well as attack surface approximation [28, 29]. In contrast, our work is on the *identification* of vulnerabilities, where the vulnerabilities are *known* in various forms such as fix commits, tickets, and mailing list discussions, but not explicitly identified as such. In this area, machine learning techniques are prominent. Wijayasekara et al. [36] point to the criticality of unidentified vulnerabilities, which they call *hidden impact* vulnerabilities. They show that 32% and 64% of Linux kernel and MySQL vulnerabilities discovered from January 2006 to April 2011 respectively were unidentified vulnerabilities, and that their numbers have increased in both software 53% and 10% respectively. Perl et al. [22] classify commits as related to a CVE or not using SVM model, but this requires prior identification of the vulnerabilities in the NVD. Zhou and Sharma [40] explore the identification of vulnerabilities in commit messages, and issue reports/PRs. Their approach does not require CVE ids, and it discovers hidden vulnerabilities without assigned CVE ids in more than 5,000 projects spanning over six programming languages. Although we use the same k-fold stacking ensemble, we implement a complete iterative pipeline that improves model precision at each iteration with more data sources and features. Our work also uses self learning [20] and a novel deployment stability metric. For the identification of vulnerability-related commits, Sabetta and Bezzi propose feature extraction also from commit patches in addition to commit messages [25]. As ours, they consider patches as natural language text, however, their work is limited in scope to commits data only. Wan considers vulnerability identification from commit messages using deep learning [31]. The work shows higher F1 scores compared to Zhou and Sharma's k-fold stacking technique [40], however, the models are trained using a dataset labeled without keyword-based filtering. This alone includes 167K Linux kernel commits. In contrast, our approach considers more data sources, and has been aimed at practical applicability by employing keyword-based filtering and self learning. This only requires the manual labeling of a small proportion of the data.

7.2 Learning with Unbalanced Data

Handling unbalanced data is an important research area in machine learning, with a number of literature surveys [16, 17, 27, 34]. The main approaches include *preprocessing*, *cost-sensitive learning*, and ensemble. Preprocessing is further classified into *re-sampling* and *feature-selection*. Re-sampling is further categorized into undersampling, oversampling, and hybrid. To balance the data, undersampling removes data from the majority class in the dataset, while oversampling synthesizes data for the minority class in the dataset. Feature-selection removes irrelevant features from the feature space, resulting in more balanced data with only features that are relevant. Cost-sensitive learning assumes higher costs for the misclassification of minority class samples compared to majority class samples, with the algorithm optimizes towards lower cost.

Ensemble is a popular solution for unbalanced learning [16]. It can be classified into three: bagging, boosting, and stacking [26]. In bagging, the dataset is split into disjoint subsets, and a different classifier is applied to each subset. The results are then combined using either voting for classification, or averaging for regression. In boosting, we serially combine weak classifiers to obtain a strong classifier. In stacking, which includes the k-fold stacking ensemble that we use, the classifiers are coordinated in parallel and their results are combined using a meta classifier or meta regressor, which in our case is logistic regression.

In software engineering, Wang and Yao consider the prediction of defective modules for the next software release based on past defect logs [33]. The data is therefore unbalanced, as the number of non-defective modules is far larger than the defective ones. They consider data re-sampling, cost-sensitive, and ensemble learning methods. One of the best results is achieved by AdaBoost.NC ensemble [32]. Rodriguez et al. also review classifiers for unbalanced data for the software defect prediction problem [23]. They consider 12 algorithms, with C4.5 and Naive Bayes as base classifiers. They discovered that ensemble, including SMOTEBoost and RUSBoost provide better results than sampling or cost-sensitive learning. Different to these approaches, we use stacking instead of boosting ensemble to be more adaptive to variations in the input data sources, nevertheless, the literature justifies our usage of ensemble.

7.3 Self-Training

Self-training is a widely-used semi-supervised learning [10] approach. It is applied when the training dataset disproportionately includes only a small amount of labeled data. This situation arises when labeled data are expensive and time consuming to get, yet the unlabeled data are easier to collect. Nigam and Ghani defines self-training as an algorithm that initially builds a single classifier using the labeled training data. It then labels the unlabeled data and converts the most confidently-predicted data item of each class into a labeled training example, iterating until all data are labeled [20]. Nigam et al. use *expectation-maximization* (*EM*), a related algorithm which iteratively uses the initial Naive Bayes classifier to label the unlabeled text data from the web to learn new parameters for the classifier until there is no change in the parameters [21]. The use

of EM reduces classification error by up to 33%. Yarowsky uses an algorithm akin to EM for word sense disambiguation [38], however, here problem-specific constraints are applied to build the initial training data as well as filtering the automatically-generated labels at each iteration. The author reports testing accuracy exceeding 95%. Rosenberg et al. [24] use self-training as defined by Nigam and Ghani [20] (which stops when all data are labeled) for object detection with results comparable to traditional supervised learning using a much larger set of fully-labeled data. Our work differs from these that we do not apply confidence metric or filtering constraint to the automatically-labeled data, and that ours does not aim to converge in iterations as our pipeline iterates in perpetuum. Nevertheless, self training is effective even for our application.

8 CONCLUSION AND FUTURE WORK

We described a design and implementation of a machine learning system to support the curation of library vulnerability database by a team of security researchers. Our system uses publicly-available data sources including NVD, bug tracking systems, commits, and mailing lists, and using these, it trains machine learning models that predict the vulnerability-relatedness of the input data. The system supports a complete pipeline from data collection, model training and prediction, to the validation of new models before deployment. It is executed iteratively to generate better models as new input data become available. We employ self-training to significantly and automatically increase the size of input dataset used in training new models to maximize the improvement in the quality of the generated models at each iteration. We also proposed a new deployment stability metric to aid the decision in deploying the newly-trained model at each iteration. We presented a case study to illustrate how our approach improves the quality of the models in an iteration, with PR AUC improvements at most 27.59%. Here we discovered that the addition of the related commit features to issues/PRs improve the precision for the recalls that matter, and that there is no uniform ordering of word2vec parameters sensitivity across data sources. We also showed how self-training results in better precision, with 10.50% PR AUC improvement, and how the deployment stability metric was useful to help discover an error.

For future work, firstly, we can try to include more keywords from our labeled dataset into our keyword-based filter to reduce input data imbalance. Secondly, the use of code features are common in machine learning approaches for program analysis [11, 15, 37]. Similarly, we can try using the library's code as another data source. Lastly, the applicability of deep learning may be worth studying.

ACKNOWLEDGMENT

We thank, in alphabetical order, Darius Foo and Spencer Hao Xiao for their comments on our draft.

REFERENCES

[1] [n.d.]. Black Duck Software Composition Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.
[2] [n.d.]. gensim: Topic Modelling for Humans. https://radimrehurek.com/gensim/index.html.
[3] [n.d.]. NVD - Home. https://nvd.nist.gov/.
[4] [n.d.]. rough-auditing-tool-for-security. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
[5] [n.d.]. scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/.
[6] [n.d.]. Software Composition Analysis. https://www.flexera.com/products/software-composition-analysis.
[7] [n.d.]. Software Composition Analysis — Veracode. https://www.veracode.com/products/software-composition-analysis.
[8] [n.d.]. SourceClear. https://www.sourceclear.com/.
[9] [n.d.]. Vulnerability Scanner. https://www.sonatype.com/appscan.
[10] O. Chapelle, B. Schölkopf, and A. Zien. 2006. *Semi-Supervised Learning*. MIT Press.
[11] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay. 2017. Machine learning for finding bugs: An initial report. In *MaLTeSQuE '17*. IEEE Comp. Soc., 21–26.
[12] J. Davis and M. Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *23rd ICML (ACM International Conference Proceeding Series)*, Vol. 148. ACM, 233–240.
[13] D. Foo, J. Yeo, X. Hao, and A. Sharma. 2019. The Dynamics of Software Composition Analysis. *CoRR* abs/1909.00973 (2019). arXiv:1909.00973
[14] S. M. Ghaffarian and H. R. Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4 (2017), 56:1–56:36.
[15] G. Grieco, G. L. Grinblat, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier. 2016. Toward Large-Scale Vulnerability Discovery using Machine Learning. In *6th CODASPY*. ACM, 85–96.
[16] H. Guo, Y. Li, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing. 2017. Learning from class-imbalanced data: Review of methods and applications. *Expert Syst. Appl.* 73 (2017), 220–239.
[17] H. He and E. A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284.
[18] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *ESEM '13*. IEEE Comp. Soc., 65–74.
[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781
[20] K. Nigam and R. Ghani. 2000. Analyzing the Effectiveness and Applicability of Co-training. In *CIKM '00*. ACM, 86–93.
[21] K. Nigam, A. McCallum, S. Thrun, and T. M. Mitchell. 1998. Learning to Classify Text from Labeled and Unlabeled Documents. In *AAAI '98*. AAAI Press / The MIT Press, 792–799.
[22] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *22nd CCS*. ACM, 426–437.
[23] D. Rodríguez, I. Herraiz, R. Harrison, J. Javier Dolado, and J. C. Riquelme. 2014. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *18th EASE*. ACM, 43:1–43:10.
[24] C. Rosenberg, M. Hebert, and H. Schneiderman. 2005. Semi-Supervised Self-Training of Object Detection Models. In *7th WACV/MOTION*. IEEE Comp. Soc., 29–36.
[25] A. Sabetta and M. Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *34th ICSME*. IEEE Comp. Soc.
[26] V. Smolyakov. [n.d.]. Ensemble Learning to Improve Machine Learning Results. https://blog.statsbot.co/ensemble-learning-d1dcd548e936.
[27] Y. Sun, A. K. C. Wong, and M. S. Kamel. 2009. Classification of Imbalanced Data: a Review. *IJPRAI* 23, 4 (2009), 687–719.
[28] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. A. Williams. 2015. Approximating Attack Surfaces with Stack Traces. In *37th ICSE*, Vol. 2. IEEE Comp. Soc., 199–208.
[29] C. Theisen, K. Herzig, B. Murphy, and L. Williams. 2017. Risk-Based Attack Surface Approximation: How Much Data Is Enough?. In *ICSE-SEIP '17*. IEEE Comp. Soc., 273–282.
[30] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th ACSAC*. IEEE Comp. Soc., 257.
[31] L. Wan. 2019. *Automated Vulnerability Detection System Based on Commit Messages*. Master's thesis. Nanyang Technological University.
[32] S. Wang, H. Chen, and X. Yao. 2010. Negative correlation learning for classification ensembles. In *IJCNN '10*. IEEE, 1–8.
[33] S. Wang and X. Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Trans. Reliability* 62, 2 (2013), 434–443.
[34] G. M. Weiss. 2004. Mining with rarity: a unifying framework. *SIGKDD Explorations* 6, 1 (2004), 7–19.
[35] D. A. Wheeler. [n.d.]. Flawfinder Home Page. https://www.dwheeler.com/flawfinder/.
[36] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *5th HSI*. IEEE, 89–96.
[37] F. Yamaguchi, F. Lindner, and K. Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *5th WOOT*. USENIX Assoc., 118–127.
[38] D. Yarowsky. 1995. Unsupervised Word Sense Disambiguation Rivaling Supervised Methods. In *33rd ACL*. Morgan Kaufmann Publishers / ACL, 189–196.
[39] M. Zalewski. [n.d.]. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl/.
[40] Y. Zhou and A. Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *11th FSE*. ACM, 914–919.