Effective Identification of Vulnerabilities using Machine Learning

***Abstract*—In this article we consider the problem of exposing *unidentified* software vulnerabilities of open-source software. An unidentified vulnerability is a vulnerability that is known, as they have been reported or silently fixed, but is not publicly identified as a vulnerability, such as by being assigned a *Common Vulnerabilities and Exposure* (*CVE*) id. We use publicly-available open-source issue reports and pull requests as well as commit data to identify if a particular version of open-source software contains an unidentified vulnerability using machine learning. Similar approach has been considered before, however, we explore the utilization of these data sources further in the following directions:**

1) **For identifying vulnerability-related commits, we additionally include as features committer's id, the paths of the modified files, and code patches, which include lines added and deleted.**
2) **To improve the precision of our results in identifying vulnerability-related commits, we in addition apply self training to use the initial model to significantly increase the size of labeled data for generating another model that is more precise.**
3) **For identifying vulnerability-related issues or pull requests, we pair GitHub issues or pull requests with their commit data, and extract features from both.**

We show that our new approach, for the same recall, on average improves the precision for vulnerability-related commits by 78% when using additional features like code patches and by 106% with self training. Moreover, for vulnerability-related issues we improve the precision by 52% when compared with previous work.

***Index Terms*—vulnerability identification, machine learning, bug report, commit**

I. Introduction

As any software, open-source software may contain vulnerabilities. Identifying vulnerabilities in open-source software has become critical as there is an increasing dependency of the whole information infrastructure on open-source software. During the software development process, it is important to analyze the libraries used by an application to identify those with lurking vulnerabilities. This belongs to the domain of *software composition analysis* (*SCA*), with commercial offerings from various companies [1]–[4].

Some of the open-source library vulnerabilities are given *Common Vulnerabilities and Exposure* (*CVE*) ids in the *National Vulnerability Database* (*NVD*), and therefore the corresponding open-source software is easy to identify as vulnerable, but more often than not, the vulnerabilities are reported and/or fixed without being identified. In this paper, we consider the problem of exposing *unidentified* vulnerabilities of open-source software. An unidentified vulnerability is one that has been reported and possibly fixed, but is not publicly identified as a vulnerability, such as assigned a CVE id. According to SourceClear [5] data, close to 50% of vulnerabilities in open-source software are not disclosed publicly with a CVE. The problem of exposing unidentified vulnerability is different from *discovering* new vulnerabilities, as the former deals with vulnerabilities that are known, whereas the latter deals with vulnerabilities that are unknown. For the latter, well-known approaches of static and dynamic program analysis exist, such as *taint analysis* and *fuzzing*.

Recent years have also seen the rise of the machine learning technology, which has been widely used for program analysis [6]. Many of the works are for discovering new vulnerabilities, either automatically [7], [8] or for help in auditing [9], however, there are growing number of works that instead deal with known but unidentified vulnerabilities. The work of Perl et al. [10], for instance, classifies if commits are related to a CVE or not using machine learning, but this requires prior identification using CVE ids. Zhou and Sharma [11] explore the identification of vulnerability-relatedness of two kinds of artifacts: commit messages, and *issue reports*, which we define here to be encompassing what are known as issues and bug reports, as well as pull requests. The approach uses machine learning techniques for natural languages. This work does not require prior CVE identification and it exposes unidentified vulnerabilities in more than 5,000 projects spanning over six programming languages [11]. Although commit messages have been considered by Zhou and Sharma [11], other parts of the commit data including committer's id, changed file paths, and the source code change (patch) itself have not been considered. Differently to Zhao and Sharma, Sabetta and Bezzi [12] in addition to using commit messages, also explore the usage of source code change (patch) data. Where Zhou and Sharma demonstrated highest precision at 77% and highest recall at 34% for commit messages, Sabetta and Bezzi show an 80% precision and 43% recall for commit patch, concluding that commit patch can be useful in practice for identifying vulnerable software.

In this article, we propose a new approach to vulnerability identification. Our approach builds upon the approach of Zhou and Sharma [11], where we similarly use the publicly-available open-source bug and issue reports as well as commit data, however, in addition, our approach differs in the following respect:

1) *We use more features from the commit data*. Zhou and Sharma only consider commit messages of the entire commit data [11]. In this article, to improve the precision of the model, in addition we consider the committer's id,

changed file paths, and commit patch, which is further divided into lines added and lines deleted. As reported by Sabetta and Bezzi [12], considering commit patches can be useful in practice.

2) *We use self training to improve our results*. Commit data are easily obtained from public repositories, yet labelling them as vulnerability-related involves significant manual effort. We therefore have a large amount of commit data, yet only a small portion is labeled. To use the whole dataset for training, we apply *self training*. Self training is a widely-used semi-supervised learning technique that is useful when training data contains only a small subset of labeled data and a large subset of unlabelled data. In self training, an initial model is trained with labeled data which is then used to label the unlabelled data, in order to label the whole dataset. After labelling, the whole dataset is then used to train a better performing model.
3) *We consider the addition of of commit data into issues, bug reports, or pull requests vulnerability identification*. Zhou and Sharma treat commits and issue reports independently [11], where the identification of vulnerability-relatedness of issue reports utilizes the features only of the issue reports text themselves. Often, however, an issue report (e.g., a pull request) has corresponding code commits. In this article, we consider the use of the commit data in order to improve the precision of the identification of the corresponding issue report as vulnerability-related. For this, we explore the possibility of machine learning models generated using the features extracted from the combination of issue reports with their corresponding commit data.

In this article, we use a dataset of 20,000 labeled commits from GitHub, and we demonstrate that adding extra commit data to commit message, in particular committer's id and changed file paths, results in increasing effectiveness in identifying vulnerability-related commits, with higher precision under the same recall, when compared to both Zhou and Sharma [11] and Sabetta and Bezzi [12]. However, although adding patch information does improve the precision and recall when the recall is low, the improvement is insignificant with higher recall. We use the generated model in a self training approach to further classify 45,701 more commits from GitHub, resulting in total of 65,701 labeled commits, after which we repeat our experiment. This approach significantly improves the precision even further. Our new approach on average improves the precision for vulnerability-related commits by 78% without self training and 106% with self training.

In using the corresponding commit data with GitHub issue reports for identifying the vulnerability-relatedness of the reports, we use close to 3,000 data items, each of which is a commit and issue report pair, where each pair has a unique issue report. This number is much less than the 20,000 labeled commit data we use before, since multiple commits may correspond to only a single GitHub issue report. Even so, here also we discover a marked improvement to Zhao and Sharma [11] in precision given the same recall, signifying the effectiveness of our approach. For the same recall, our new approach improves the identification of vulnerability-related issue reports on average by 52% for the same recall.

In summary, using the approaches presented in this article, we obtain significant improvements in model precision, for both the identification of vulnerability-related commits and issue reports. These improvements are expected to significantly reduce manual work in identifying vulnerability-related commits.

We start our article by providing a discussion of our approach in Section II. We then present the evaluate our approach by presenting our results in Section III. We provide more discussions on our results in Section IV. We present related work in Section V and conclude our article in Section VI.

II. Approach

A. Automated Identification of Security Issues from Commit Messages and Bug Reports

We use two kinds of machine learning approaches:

- A supervised approach, where labeled data are used to train the machine learning models. The labeled data were manually triaged and labeled by security researchers.
- A semi-supervised approach, where we use self training to increase the amount of labeled data for training, with the expected result to increase the performance of our model. In our self training approach, we use the model of the supervised approach learned earlier to further label the unlabelled commit data. We use this approach to improve the precision of the identification of vulnerability-related commits.

Our labeled commits dataset is imbalanced, as the vulnerability-related data are much less than the non-vulnerability-related ones. For this dataset, we have positive data of only 20% of the total number of data for our manually-labeled dataset, and this number is only 7% when labeled data from self training are included. In building and validating the models, we use the same approach as Zhou and Sharma [11] called *K-fold stacking model*. In this approach, we use an ensemble of classifiers, which is a known approach to build models of imbalanced data (see a survey in [13]), and we use K-fold cross validation to separate the dataset into training and testing datasets. We illustrate the approach in Figure 1.

For the K-fold cross validation, the dataset is split into K parts, where one part is used for testing and $K-1$ parts are used for training. Where Zhao and Sharma use 12-fold stacking model, we use 10-fold stacking model instead. In our preliminary experiments, we discover that with using 10-fold, we can get better result compared with 12-fold. The K parts are used to train six different kinds of classifiers. Logistic regression is used to find optimal ensemble of these classifiers. In our ensemble, we use the same ensemble of six classifiers as Zhou and Sharma [11]. The set includes random forest (RF), Gaussian Naive Bayes, K-nearest neighbours (K-NN), linear

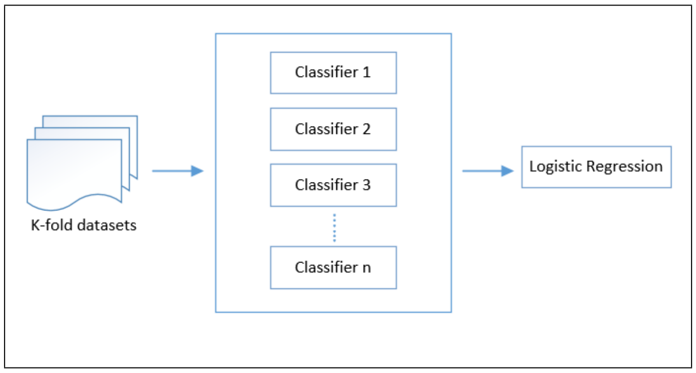


Fig. 1: K-Fold Stacking Model

support vector machine (SVM), gradient boosting (GB), and AdaBoost (Ada).

The models we generate decide on whether a commit or an issue report corresponds to a vulnerability in terms of confidence value from 0 to 1, called the *probability of vulnerability relatedness* (*PVR*). This allows for an adjustment of a threshold to distinguish vulnerable and non-vulnerable items, i.e., given a threshold τ, if the PVR is strictly greater than τ, the item can be decided to be vulnerability related (reported as a *positive*), otherwise it is not vulnerability related (reported as a *negative*).

B. Identifying Vulnerability-Related Commits

We show the sizes of our datasets in Table I. In identifying vulnerability-related commits, we use a dataset consisting of 20,000 commits. For each record in the dataset, we consider two types of data:

1) *Natural language features*. These include:
   a) Commit message.
   b) Committer's id.
   c) Changed file paths.
2) *Program code features*, which is the commit patch. These features consist of the following two:
   a) Lines added to the code.
   b) Lines deleted from the code.

We generate two models based on features used:

1) Using all natural language features.
2) Using all natural language and program code features.

In combining natural language features and the commit patch, we compute separate K-fold stacking models for natural language artifacts and the commit patches, and we combine the results using logistic regression. This is because of the lack of obvious correlation between the text of the program code artifacts and the natural language artifacts[1]. In our approach for the commit patch, similar to Sabetta and Bezzi [12], we treat the patch as natural language and use word2vec [15] to extract feature vector from word2vec model, which is built separately with natural language features. We use most of the textual content in a patch except comment symbols and brackets.

[1]In GitHub's natural language semantic code search, such correlation even needs to be modeled using machine learning [14].

Our approach results in two PVR values for each record: One for the natural language features, and another for the commit patch. We combine the results of the two models using logistic regression. The overall approach with the combining of the results is illustrated in Figure 2.

C. Enhancing Vulnerability-Related Commits Identification Using Self Training

Our precision result for the approach presented in the previous section is limited by the limited number of labeled training data that we use (20,000 commits). One approach to improve precision is by increasing the number of labeled data, however, labelling the data, being resource-intensive manual labor, is prohibitively expensive.

As mentioned, our approach described in the previous section results in two models based on the set features used, and as we shall see later (Section III), the inclusion of program code features (commit patch) results in the model with better precision on average. We use this model to label more unlabelled data, and add them in our labeled dataset, resulting in total 65,701 commit data (including the 20,000 manually-labeled commit data). In the labelling process, we use PVR low and high thresholds to decide whether a particular commit with its PVR value should be considered a positive commit (vulnerability-related) or a negative one (non-vulnerability-related). For the low and high thresholds, we use the values 0.22 and 0.88, respectively. Whenever a PVR less than 0.22 is computed by our model, the commit is considered definitely non-vulnerability-related, whereas for commits with PVR more than 0.88, the commit is considered a definite vulnerability-related. We do not include in our dataset commits with PVR from 0.22 and 0.88.

In deciding to use the values 0.22 and 0.88, we learn from the the data we obtain for deciding vulnerability-relatedness of commits in the previous section. 0.22 is the threshold such that we can decide that a commit is non-vulnerability-related with *precision* 0.93 and 0.88 is the threshold such that we can decide that a commit is vulnerable with the precision of 0.91. Hence, both 0.22 and 0.88 are thresholds for which high precision, which is $>$ 90% is obtained. The notion of precision is formalized later in Section III-B.

Using the larger 65,701 commit data, we repeat our approach in the previous section, which uses K-fold stacking model to identify vulnerability-related commits using natural language and program code features.

D. Enhancing Identification of Vulnerability-Related Issues Reports Using Commit Data

In our second approach, we consider the usage of commit data that are related to the issue reports. While Zhou and Sharma [11] consider datasets form Jira and Bugzilla deployments and GitHub, here we only consider GitHub issues and pull requests, since we only use commit data from GitHub. For Jira or Bugzilla deployments, the corresponding commit data are not identifiable as easily. Here we consider both issues and pull requests to be of the same kind (called *issue reports*),

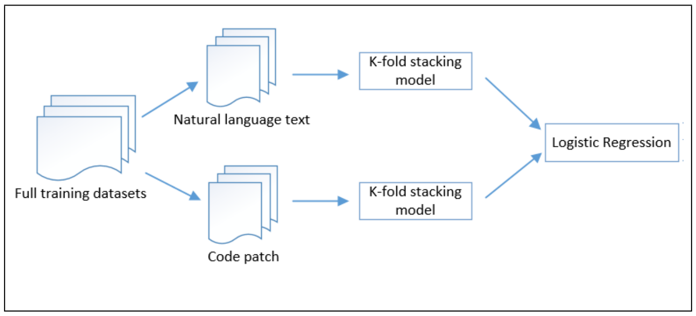


Fig. 2: Overall Model

	Commit	Commit w/ Self Training	(Issue Report,Commit) Pairs
Positive	3,989	4,838	1,396
Negative	16,011	60,683	1,548

TABLE I
SIZES OF OUR DATASETS

and we consider a dataset with 2944 records (see Table I). This dataset is built in two ways, firstly we considered all the commits that referenced a GitHub issue report, we found 2023 records this way. For the rest, we considered all the issue reports in our dataset and looked for any commits that were referenced in them. Thus, in total we ended up with 2944 pairs of issue reports and commits.

Similar to our approach with identifying vulnerability-related commits in Section II-B, here we also separate natural language features and program code features, and combine the results of two models using logistic regression (Figure 2). The difference here is that the natural language features now include issue report texts. Here it is important to note that an issue report may be related to multiple commits. For each individual feature, e.g., commit message, we append such feature from all of the commits into one contiguous text. We then use word2vec [15] to compute the feature vector from the text. For the code patch itself, we append all the lines added in one contiguous text, and separately append all the lines deleted into another contiguous text. We then apply word2vec to compute a feature vector from each of these.

III. Evaluation

A. Setup

The sizes of the dataset that we use are shown in Table I. The second column shows the numbers of positive and negative data for our 20,000 manually-labelled commits dataset. The third column shows the number of positive and negative data when self-training data are included. Finally, the fourth column shows the numbers of positive and negative data for our issue report [2] and commit pairs. As shown in Table I, our labeled dataset for the commits is an imbalanced one, having positive data of only 20% of the total number of data for the manually-labelled commit data, and only 7% of the total number of data when data labelled using self training is included, hence we use K-fold stacking model (see Section II-A). The K-fold stacking model is known to be effective for imbalanced data. For generating the models we employ algorithms provided by the scikit-learn [16] library. We use logistic regression to combine the results across all the models. The models we generate map a source data item (issue reports or commits) to a PVR value from 0 to 1.

[2]Recall that in Section I we define this to be including issues, bug reports, and pull requests.

B. Metrics Based on PVR Threshold

For deciding whether a data item is vulnerability related or not, we define threshold value on its PVR computed by a model. As mentioned in Section II-A, given a threshold τ, if the PVR is strictly greater than τ, the item can be decided to be vulnerability related (reported as a *positive*), otherwise it is not vulnerability related (reported as a *negative*). When validating our results, we obtain the values for true and false positives, and true and false negatives.

Similar to [11], to measure vulnerability prediction results, we use two metrics: *precision* and *recall*. These properties can be computed by fixing a PVR threshold. They are defined as follows:

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{1}$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{2}$$

The reasons that we target these two metrics are:

1) Precision reflects the ratio of true and false positives. In imbalanced scenario like ours, it helps us focus on the true vulnerabilities. The overall ratio of vulnerability-related items in our manually-labeled dataset is less than 20%. That is, if manual effort is devoted to checking the data, 80% of the time will be spent on false-positive items. Therefore, a high precision would save a lot of manual work in identifying false positives. Other measure that is sometimes used in the area of software bug checking is *false positive rate* (cf. [17]), however, this rate does not correlate to the amount of false positives that need to be dealt with by security researchers or developers.
2) Recall indicates the coverage of our approach wrt. existing vulnerabilities. Our aim in developing this approach is to cover all of the vulnerability-related commits and bug reports. The higher the recall value, the more actual vulnerabilities are identified by our approach.

C. Results in Identifying Vulnerability-Related Commits

We first perform experiments in identifying vulnerability-related commits by generating models using commit data only (cf. Section II-B). We evaluate our results by providing comparisons with approaches known in the literature, including Zhou and Sharma's [11] and Sabetta and Bezzi's [12]. The comparison of the features used among the approaches considered is shown in Table II.

Using our experimental results, we answer the following research questions:

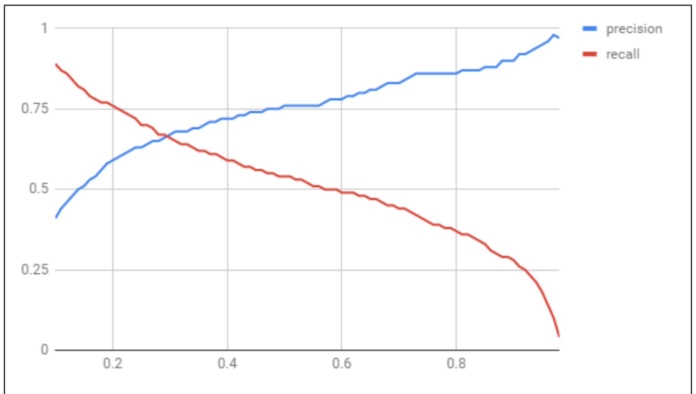
	Message	Committer	Paths	Patch
[11]	✓			
[12]	✓			✓
Ours w/o Patch	✓	✓	✓	
Ours w/ Patch	✓	✓	✓	✓
Ours w/ Patch & Self Training	✓	✓	✓	✓

TABLE II
COMMIT FEATURES USED IN THE COMPARED APPROACHES

- *RQ1. Can we effectively identify vulnerable software using the commit data only?* Here we use 20,000 commit data manually triaged by security researchers. The same question has been answered by Zhao and Sharma, and by Sabetta and Bezzi [11], [12], however, here we take a different approach by considering also the committer's id and changed file paths.
- *RQ2. Does the use of commit patch as a feature improve the identification quality of vulnerability-related commits when compared to not using commit patch?* Sabetta and Bezzi [12] consider commit patch to identify vulnerability-related commits. This motivates us to consider commit patch as well. In our approach, we consider both lines added and lines deleted as separate features.
- *RQ3. When committer's id and changed file path are already included together with commit message as features, how significant is the inclusion of commit patch as a feature to improve the identification quality of vulnerability-related commits?* Sabetta and Bezzi [12] consider commit patch to identify vulnerability-related commits in addition to commit message. They show that their approach significantly improves the results of Zhou and Sharma, however, both approaches do not consider committer's id and changed file paths as features (cf. Table II). From our own results, we can observe the effect of code patch inclusion as a feature, when committer's id and changed file paths are already included.
- *RQ4. Does self training improve the identification quality of vulnerability-related commits?* One approach to improve precision is by increasing the number of labeled data. When manual labelling is prohibitively expensive, automated labelling using previously-computed model can be used. In our setting, we can use a model previously computed for deciding vulnerability-related commit using manually-labelled dataset.

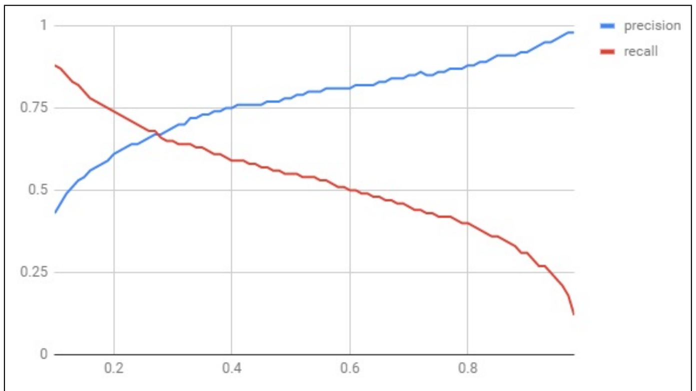
We have trained two models on our commit datasets, one of which is using the natural language features only (commit message, committer's id, changed file paths). For the 20,000 manually-labelled commit data, we show the precision and recall results for this model, with varying the PVR threshold in Figure 3. Another is to train our model using all features, including lines added and deleted in the commit patch. The precision and recall, under various PVR thresholds for the 20,000 manually-labelled commit data for this model is shown in Figure 4.

In Table III, we show the precision of our models given



Probability of Vulnerability Relatedness (PVR) Threshold

Fig. 3: Precision and Recall for the identification of vulnerability-related commits without considering commit patch



Probability of Vulnerability Relatedness (PVR) Threshold

Fig. 4: Precision and Recall for the identification of vulnerability-related commits with considering commit patch

fixed recall values. The recall values are the values selected by Sabetta and Bezzi in presenting their results in [12]. In the same table we also show the precision results of Zhao and Sharma (second column) and Sabetta and Bezzi (third column) [11], [12]. For Sabetta and Bezzi's precision results, we display their best results, which are from using both commit message and commit patch as features. As can be seen from the table, our results outperform both Zhao and Sharma's and Sabetta and Bezzi's models. The fourth and fifth columns show our precision results with and without the usage of commit patch, respectively (cf. Table II). This shows that our approach is practical, and we can answer RQ1 in the affirmative. Table IV shows the improvements when compared to the results of Zhao and Sharma [11]. On average, including more features of commit data does improve the identification of vulnerability-related commits, improving upon the results of [11] by at most 78% without self training, and 106% with self training.

Table III also provides an answer to RQ2. To answer this question, we compare the fourth and the fifth columns of Table III. The commit patch does improve the identification quality of the vulnerability-related commits when the recall

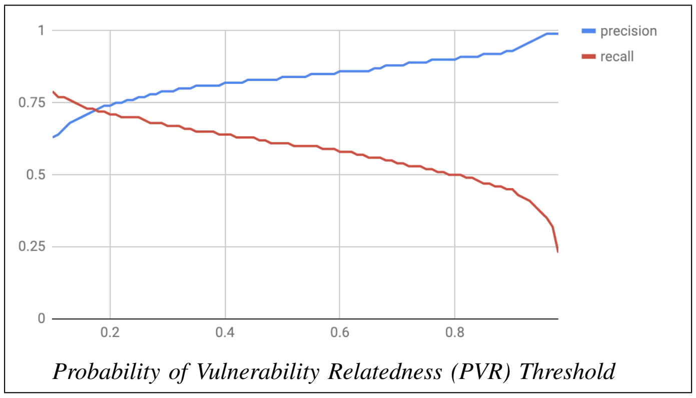


Fig. 5: Precision and Recall for the identification of vulnerability-related commits with considering commit patch and using Self Training

Recall	Precision				
	[11]	[12]	Ours w/o Patch	Ours w/ Patch	Ours w/ Patch & Self Training
0.50	0.50	0.74	0.76	0.81	0.91
0.72	0.34	0.57	0.63	0.63	0.74
0.76	0.31	0.56	0.59	0.58	0.68

TABLE III
PRECISION WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED COMMITS

Recall	Improvements as compared to [11]		
	Ours w/o Patch (%)	Ours w/ Patch (%)	Ours w/ Patch & Self Training (%)
0.50	52	62	82
0.72	85	85	117
0.76	90	87	119
Average	76	**78**	**106**

TABLE IV
PRECISION IMPROVEMENTS AS COMPARED TO [11] WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED COMMITS

Recall	Improvements as compared to [12]		
	Ours w/o Patch (%)	Ours w/ Patch (%)	Ours w/ Patch & Self Training (%)
0.50	3	9	23
0.72	11	11	30
0.76	5	4	21
Average	6	8	25

TABLE V
PRECISION IMPROVEMENTS AS COMPARED TO [12] WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED COMMITS

is low, but improvement is insignificant when recall increases. When comparing Figures 3 and 4, the precision and recall have similar curves for the cases without and with commit patch.

To answer RQ3, we note that when comparing to the results of Sabetta and Bezzi [12] (third column of Table III), which uses as features only commit messages and commit patches (cf. Table II), our results indicates that the advantage of using commit patch as feature becomes less when committer's id and changed file paths are also used as features. Table V displays the improvements to Sabetta and Bezzi's precision results, when the recall values are fixed.

Finally Table III also shows that we can answer RQ4 in the affirmative. The effectiveness of self training is even more significant than the use of commit patch alone. Figure 5 shows significantly different curves to that of Figures 3 and 4, where for Figure 5 both precision and recall are generally higher than that of Figures 3 and 4.

D. Results in Enhancing Identification of Vulnerability-Related Issue Reports Using Commit Data

Here we answer the following two research questions:

- *RQ5. Does the addition of commit data improve the quality of the vulnerability identification for the related issue reports?* The identification of issues as vulnerability related using machine learning was first shown by Zhou and Sharma [11]. Here we attempt to improve their results by incorporating commit data into the feature set. Where Zhou and Sharma use data from GitHub as well as Jira and Bugzilla deployments, we use the data from GitHub alone. This is because the related commits, being from the same GitHub repository as the issue reports, are easier to identify.
- *RQ6. Does the use of commit patch improve the identification quality of vulnerability-related issues?* We consider the identification of vulnerability-related issues, by adding more text from all of the related commits. In one experiment, we add commit messages, as well as committer's ids and changed file paths. In another experiment, we furthermore in addition use lines added and lines deleted from the patch. We compare the models generated in the two experiments.

Figure 6 shows the comparison of precision and recall of the identification of vulnerability-related issue reports when using the commit data but without using commit patch. Figure 7 shows the comparison of precision and recall of the identification of vulnerability-related issue reports when using the commit data and the commit patch. The precision in both figures drop from 1 to 0 quickly when the PVR threshold is at a very high value of 0.96. The reason is that no PVR is higher than the threshold, making the precision 0.

Table VI shows the precision, given a set of recall values, where we include the results of [11] in the second column. From Table VI, we can infer that our approach does improve the quality of the model in identifying vulnerability-related commits, however, the use of commit patch does not seem to contribute much to the improvement in precision when recall increases. As is observable by comparing Figures 6 and 7, the precision and recall without and with using commit patch have similar curves.

We discover that it is indeed the case that the usage of commits helps in the identification of vulnerability-related issue reports, and we can therefore answer RQ5 in the affirmative. Similar to the result for RQ2, for RQ6 we also discover that the commit patch does not significantly improve the identification quality of vulnerability-related issues when recall increases, although improvements are observable when recall is low. We

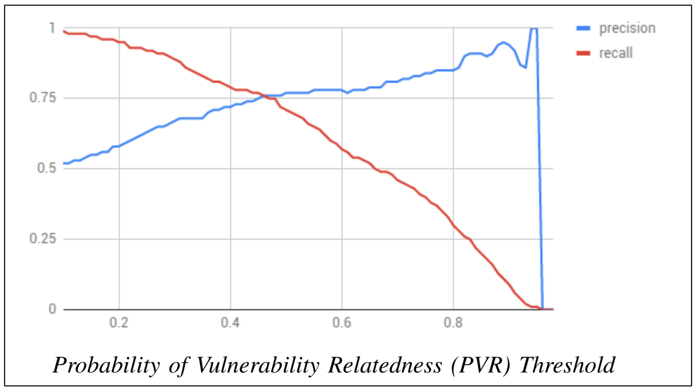


Fig. 6: Precision and Recall for identification of vulnerability-related issue reports without considering commit patch

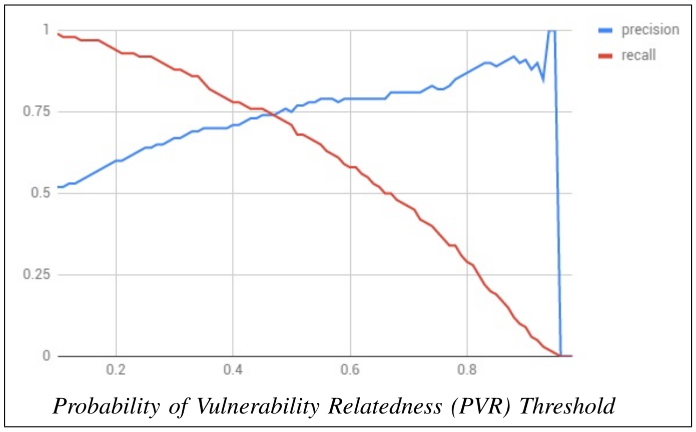


Fig. 7: Precision and Recall for identification of vulnerability-related issue reports with considering commit patch

show the improvements in precision compared to the results of [11] in Table VII, where on average, the use of patch data does not improve the identification of vulnerability-related issues. Our new techniques on average improves the identification of vulnerability-related issues in [11] by 52% at most, and that is, without commit patch data.

Recall	**Precision**		
	[11]	Ours w/o Patch	Ours w/ Patch
0.50	0.70	0.79	0.81
0.72	0.47	0.76	0.76
0.76	0.42	0.76	0.74

TABLE VI
PRECISION WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED ISSUE REPORTS

Recall	**Improvements as compared to [11]**	
	Ours w/o Patch (%)	Ours w/ Patch (%)
0.50	13	16
0.72	62	62
0.76	81	76
Average	52	51

TABLE VII
PRECISION IMPROVEMENTS WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED ISSUE REPORTS

IV. Discussion

A. *Threats to Validity*

Threats to validity of our results are as follows:

- Our results depend on the dataset we use. Since as we have shown, commit data including committer's id may contribute significantly to the identification of vulnerability-related commits, and since different persons may be involved in different projects, this suggests that the results should also depend on the projects that we have selected in our dataset.
- Similar to Zhou and Sharma [11], in building our datasets, we filtered out commits and issues reports that are clearly unrelated to vulnerabilities using regular expression. The regular expression includes common keywords related to security vulnerabilities, such as *security*, *vulnerability*, *XSS*, *CVE*, etc.. Therefore, our results depend on this filtering. Our results therefore may vary when the approach is applied without filtering or with different filtering.

B. *Committer's Id as Feature*

Previous work reports that the identification results for issue reports are better than those for commit messages [11]. This is because issue reports have richer text information than commits, and more useful features have been used to train the model for issue reports. They also decided to use only commit messages as the only feature. They provided the following reason:

- Comments are not used since only a small number of commits have comments.
- Project names are not used since the model should be applicable to various projects.
- Committer's names are not used due to concerns about the lack of accuracy.

In this article, we instead perform a real experiment with using the committer's id. There is an indication in the literature that the quality of the developer significantly affects the quality of the code [18]. It is possible the more advanced developers are more likely to be assigned the task to correct a vulnerability. We have shown experimentally that considering commit data which include committer's id is useful to improve the accuracy of the model.

C. *Treating Patch as Program Code*

We discover only marginal improvements with the use of commit patch. In our approach, similar to Sabetta and Bezzi [12], we consider program patch as natural language text. Another possible approach is to parse the patch and use the set of tokens in the result as a feature. In fact, the use of code features are common in machine learning approaches for program analysis [7]–[9]. The use of code-specific features therefore is a possible future direction to be attempted in improving our approach.

V. Related Work

A. Vulnerability Identification

Our work focuses on finding vulnerabilities that are unidentified. In this area, Perl et al. [10] classifies commits as related to a CVE or not. Wijayasekara et al. [19] propose an approach to mine bug databases to identify software vulnerabilities. Our work is most closely related to the work of Zhou and Sharma [11], which explores the identification of vulnerabilities from commit messages and bug or issue reports using machine learning for natural languages. Compared to Perl et al. [10], the work of Zhou and Sharma does not require CVE ids, and it improves upon Perl et al. by discovering hidden vulnerabilities without assigned CVE ids in more than 5,000 projects spanning over six programming languages. For the identification of vulnerability-related commits, Sabetta and Bezzi propose feature extraction also from commit patches in addition to just from commit messages [12], which is the approach taken by Zhou and Sharma [11]. We adopt the approach of Zhou and Sharma, which is a supervised machine learning approach with the usage of K-fold stacking model (described in Section II-A) due to imbalanced data, however, inspired by the work of Sabetta and Bezzi, we improve upon the work of Zhou and Sharma in the following ways:

1) We consider more commit data features to improve precision, including committer's id, changed file paths, and the commit patch itself.
2) In our approach to identify vulnerabillity-related commits, we use self learning to increase the amount of labeled data. We discovered that this self learning is highly effective.
3) We extract features from commit data for identifying vulnerability-related issue reports.

Given the same recall, we better the precision of Zhou and Sharma, as can be seen in Tables IV and VII. For vulnerability-related commit identification, we also better the precision of Sabetta and Bezzi's approach, as is shown in Table V.

B. Learning with Imbalanced Data

The K-fold stacking models that we use is based on models ensemble which can be used for imbalanced data. Handling imbalanced data is an important research area in machine learning with an early survey by Weiss [20] and a more recent surveys by He and Garcia [21], by Sun et al. [22] and by Guo et al. [13]. From these, we learn that the main approaches to imbalanced dataset include *preprocessing*, *cost-sensitive*, and ensemble methods. The preprocessing methods can be further classified into *re-sampling* methods and *feature-selection* methods. The re-sampling methods can be further categorized into undersampling, oversampling, and hybrid methods. Undersampling removes data from the majority class in the dataset to balance the data, while oversampling synthesize data for the minority class in the dataset. In the feature-selection methods, one removes some 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 methods is a popular solution for imbalanced learning [13]. It can be classified into three: bagging, boosting, and stacking [23]. In bagging method, 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 method, we serially combine weak classifiers to obtain a strong classifier. In stacking method, which includes the K-fold stacking method 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 the area of software engineering, Wang and Yao consider imbalanced data for software defect prediction [24]. The problem of software defect prediction is to predict defective modules for the next software release based on past defect logs. The data is therefore imbalanced, as the number of non-defective modules is far larger than the defective ones. The authors consider data re-sampling techniques, cost-sensitive method, and ensemble learning methods. One of the best results is provided by and boosting learning method AdaBoost.NC [25]. The work of Rodriguez et al. also reviews classifiers for imbalanced data for the software defect prediction problem [26]. They consider 12 algorithms, and with using C4.5 and Naive Bayes as base classifiers, and they use Matthew's Correlation Coefficient (MCC) as performance metric. They also discovered that ensemble methods, including SMOTEBoost and RUSBoost provide better results than sampling or const-sensitive methods.

C. Self Training

Self training is a widely-used semi-supervised learning when training data contains a small subset of labeled data, and with a large amount of unlabeled data. This situation arises when when labeled data are often expensive and time consuming to get, yet the unlabeled data are easier to collect. In self training, a model is trained with labeled data and then used to label the unlabeled data. The larger set of the labeled data is then used to train better performance model.

Nigam et al. use self training to classify text from three different real-world domains: newsgroup postings, web pages and newswire articles [27]. They use naive Bayes classifier and *expectation-maximization* (*EM*) algorithms for model training. Their experiment shows unlabeled data contains useful information about target functions, and the use of unlabeled data reduces classification error by up to 33%. Yarowsky uses self training for word sense disambiguation [28]. The algorithm is based on two constraints that words tend to have one sense per discourse and one sense per collocation. It achieves nearly the same performance with supervised algorithm given identical training contexts (95.5% vs. 96.1%) and even gets better performance when using one-sense-per-discourse constraint. Self training is used by Rosenberg et al. [29] for object detection. They point out if unlabeled data are labeled incorrectly by self-training model and added to the labeled data set, the model may be potentially corrupted. So they add the self-training labeled data into labeled data set incrementally and check the result. They use naive Bayes classifier as well. Their study demonstrates that self-training model can achieve results comparable to a model trained in the traditional supervised learning using a much larger set of fully labeled data.

The above works show self training can achieve good result for semi-supervised learning. This is also evidenced by our results in Section III-C. In our experiments, we have a large number of unlabeled commits and GitHub issue data we collected. We use stacking model combined with self training, which significantly improves the model performance a lot.

D. Vulnerability Discovery

There are many works that attempts to analyze software or related artifacts to discover vulnerabilities that are not previously known. Although this problem is different from the focus of this article, nevertheless, the subjects of the analysis, which encompass the program code or related artifact are overlapping. In this section we discuss other work in static and dynamic program analysis, symbolic execution, and machine learning.

Static analysis is a way of analyzing program code without actually executing it. This has a practical advantage of an easy setup, since it treats programs as data without the need to setup a runtime environment to execute them. Static analyses have been applied to various code artifact, from the source code to the binary, and everything in between, such as compiler intermediate language code. Static vulnerability detection at the source level include Flawfinder [30], IST4 [31] for C/C++, RATS [32] for multiple languages (C, C++, Perl, PHP, and Python), and Clang static analyzer for C and C++ [33]. However, these analyzers are language-specific, and even for supported languages may have cases where they fail to find the underlying issues due to the imprecision of the analysis. For example, RATS does not find cross-site scripting (XSS) or SQL injection vulnerabilities, which requires reasoning on the flow of data. Moreover, when applied to real-world projects, these tools raise massive false positives that are hard to reduce. Other static analysis tools work at the intermediate language level. FindBugs [34] analyzes Java bytecode and source code to discover bug patterns and Parfait [17] works on the LLVM language. Performing analysis at the lower level of abstraction potentially removes the dependency on programming languages. However, although FindBugs has a comparatively low report count [35], and Parfait has a low false positive rate [17], their precisions are unknown. Domain specific languages like SGL [36] aim to improve the precision of static analysis but require users to specify patterns of vulnerabilities or bugs.

Dynamic analysis analyzes the source code by executing it on real inputs. Basic dynamic analysis (or testing) tools search for vulnerabilities by executing the *program under test* (*PUT*) on a range of possible inputs. There are also dynamic analysis tools that do taint tracking at runtime [37]–[39]. PHP Aspis does dynamic taint analysis to identify XSS and SQL vulnerabilities [39]. ZAP [37] finds vulnerabilities in web applications. Some dynamic analysis tools are called *fuzzers*. Fuzzers work by executing the program under test using inputs that are likely to result in unexpected behavior, such as a crash or a successful exploit. One such database of inputs is FuzzDB [40] used by ZAP. Fuzzers can be categorized as *black box*, *grey box*, and *white box*. Black-box fuzzers executes the PUT without any knowledge about the PUT. Some well-known black-box fuzzers include JBroFuzz [41] used in ZAP and Peach fuzzer [42]. Grey-box fuzzers execute the PUT with a limited knowledge about the PUT. Well-known grey-box fuzzers are AFL [43] and libFuzzer [44]. They aim to achieve high coverage by detecting test inputs in the corpus that when executed reach new code region. The inputs are then prioritized to be mutated to generate new inputs. White-box fuzzers (aka. *concolic testers*) employ full semantic knowledge of the PUT and *symbolic execution* technology [45] to achieve high path coverage. However, they suffer from scalability issues. Well-known white-box fuzzers include DART [46], CUTE [47] and jCUTE [48].

Symbolic execution [45] uses the semantic of the code to translate all the instructions along an execution path into a set of constraints called the *path condition*, whose satisfiability is testable using a *constraint solver*. By mutating the path condition, the execution paths of the program can be explored. A symbolic execution engine replaces the inputs of the program with symbolic variables which represent unknown values. The path condition is a logical relation on these symbolic input variables. Whitebox fuzzers use symbolic execution where a PUT's execution path guides the construction of the path condition. New inputs are generated based on the mutation of the path condition and applying constraint solving on the mutated path condition. A well-known symbolic execution tool that does not start from an actual execution path is KLEE [49]. The symbolic execution tools that we mention here all perform path enumeration and they suffer from path explosion as well as performance penalty due to constraint solving: It is well known that each call to the constraint solver is expensive. Some well-known solvers used in symbolic execution are Z3 [50] and STP [51].

Besides the above techniques that focus exclusively on the program code, machine-learning techniques provide an alternative to assist vulnerability detection by mining context and semantic information from program code and beyond. Some work focus on detecting vulnerabilities using the program code as input data. Among these works, the work of Medeiros et al. [52] detects false positives in static program analysis bug report using machine learning. Shar et al. [53] focus on SQL injection and cross-site scripting (XSS). Others perform bug detection using non-program-code artifacts. Sahoo et al. [54] provide a survey on malicious URL detection approaches using machine learning. Ghaffarian and Shahriari provide a survey on software vulnerability analysis and discovery using machine learning [6].

VI. Conclusion

In this article we proposed an approach based on machine learning to identify vulnerabilities in open-source software. We used the publicly-available open-source issue reports, including bug reports and pull requests, as well as commit data to determine if a particular version of open-source software is vulnerable. We improved upon existing approach in the literature [11] by extracting features from more parts of the commit data, including committer's id and changed file paths. We also consider the use of self training technique to increase the size of labeled data. We experimentally demonstrated that our new approach improved the performance of previous approach, where we improve the precision for identification of vulnerability-related commits on average by 78% at most without self training, and 106% with self training. We also consider pairing commits and their corresponding issue reports to improve the identification effectiveness of vulnerability-related issue reports. Our approach on average improves the precision by 52%. In summary, we obtain significant improvements in model precision, for both the identification of vulnerability-related commits and issue reports. These improvements significantly reduce manual work in identifying vulnerability-related commits.

References

[1] "Veracode: software composition analysis," https://www.veracode.com/products/software-composition-analysis.

[2] "Vulnerability scanner," https://www.sonatype.com/appscan.

[3] "Black Duck software composition analysis," https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.

[4] "Software composition analysis," https://www.flexera.com/products/software-composition-analysis.

[5] "SourceClear," https://www.sourceclear.com/.

[6] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 56:1–56:36, 2017.

[7] G. Grieco, G. L. Grinblat, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *6th CODASPY*, E. Bertino, R. Sandhu, and A. Pretschner, Eds. ACM, 2016, pp. 85–96.

[8] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs: An initial report," in *MaLTeSQuE '17*, F. A. Fontana, B. Walter, and M. Zanoni, Eds. IEEE Computer Society, 2017, pp. 21–26.

[9] F. Yamaguchi, F. F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings*, D. Brumley and M. Zalewski, Eds. USENIX Association, 2011, pp. 118–127.

[10] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *22nd CCS*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 426–437.

[11] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *11th FSE*. ACM, 2017, pp. 914–919.

[12] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *34th ICSME*. IEEE Computer Society, Sep. 2018.

[13] H. Guo, Y. Li, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing, "Learning from class-imbalanced data: Review of methods and applications," *Expert Syst. Appl.*, vol. 73, pp. 220–239, 2017.

[14] H. Husain and H.-H. Wu, "Towards natural language semantic code search," https://githubengineering.com/towards-natural-language-semantic-code-search/, 2018.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[16] "scikit-learn: Machine learning in Python – scikit-learn 0.19.2 documentation," http://scikit-learn.org/stable/.

[17] C. Cifuentes, N. Keynes, L. Li, N. Hawes, M. Valdiviezo, A. Browne, J. Zimmermann, A. Craik, D. Teoh, and C. Hoermann, "Static deep error checking in large system applications using parfait," T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 432–435.

[18] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *ESEM '13*. IEEE Computer Society, 2013, pp. 65–74.

[19] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *5th HSI*. IEEE, 2012, pp. 89–96.

[20] G. M. Weiss, "Mining with rarity: a unifying framework," *SIGKDD Explorations*, vol. 6, no. 1, pp. 7–19, 2004.

[21] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, 2009.

[22] Y. Sun, A. K. C. Wong, and M. S. Kamel, "Classification of imbalanced data: a review," *IJPRAI*, vol. 23, no. 4, pp. 687–719, 2009.

[23] V. Smolyakov, "Ensemble learning to improve machine learning results," https://blog.statsbot.co/ensemble-learning-d1dcd548e936.

[24] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[25] S. Wang, H. Chen, and X. Yao, "Negative correlation learning for classification ensembles," in *International Joint Conference on Neural Networks, IJCNN 2010, Barcelona, Spain, 18-23 July, 2010*. IEEE, 2010, pp. 1–8.

[26] D. Rodríguez, I. Herraiz, R. Harrison, J. J. Dolado, and J. C. Riquelme, "Preliminary comparison of techniques for dealing with imbalance in software defect prediction," in *18th EASE*, M. J. Shepperd, T. Hall, and I. Myrtveit, Eds. ACM, 2014, pp. 43:1–43:10.

[27] K. Nigam, A. McCallum, S. Thrun, and T. M. Mitchell, "Learning to classify text from labeled and unlabeled documents," in *AAAI '98*, J. Mostow and C. Rich, Eds. AAAI Press / The MIT Press, 1998, pp. 792–799.

[28] D. Yarowsky, "Unsupervised word sense disambiguation rivaling supervised methods," in *33rd Annual Meeting of the Association for Computational Linguistics*, H. Uszkoreit, Ed. Morgan Kaufmann Publishers / ACL, 1995, pp. 189–196.

[29] C. Rosenberg, M. Hebert, and H. Schneiderman, "Semi-supervised self-training of object detection models," in *7th WACV/MOTION*. IEEE Computer Society, 2005, pp. 29–36.

[30] D. A. Wheeler, "Flawfinder home page," https://www.dwheeler.com/flawfinder/.

[31] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *16th ACSAC*. IEEE Computer Society, 2000, p. 257.

[32] "rough-auditing-tool-for-security," https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[33] "Clang static analyzer," https://clang-analyzer.llvm.org/.

[34] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[35] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th ISSRE*. IEEE Computer Society, 2004, pp. 245–256.

[36] D. Foo, J. Yeo, M. Y. Ang, and A. Sharma, "Sgl: A domain-specific language for large-scale analysis of open-source code," *IEEE Cybersecurity Development, SecDev*, 2018.

[37] "OWASP Zed attack proxy project," https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[38] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: preventing sql injection attacks using dynamic candidate evaluations," in *CCS '07*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 12–24.

[39] I. Papagiannis, M. Migliavacca, and P. R. Pietzuch, "PHP aspis: Using partial taint tracking to protect against injection attacks," in *WebApps '11*, A. Fox, Ed. USENIX Association, 2011.

[40] "GitHub – fuzzdb-project/fuzzdb," https://github.com/fuzzdb-project/fuzzdb.

[41] "JBroFuzz," https://www.owasp.org/index.php/JBroFuzz.

[42] "Peach fuzzer," https://www.peach.tech/.

[43] M. Zalewski, “American fuzzy lop (2.52b),” http://lcamtuf.coredump.cx/afl/.
[44] “libFuzzer – a library for coverage-guided fuzz testing,” https://llvm.org/docs/LibFuzzer.html.
[45] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[46] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *PLDI ’05*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.
[47] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *10th ESEC/13th FSE*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272.
[48] K. Sen and G. Agha, “CUTE and jcute: Concolic unit testing and explicit path model-checking tools,” in *18th CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 419–423.
[49] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *8th OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.
[50] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *14th TACAS*, vol. 4963, 2008, pp. 337–340.
[51] V. Ganesh and D. L. Dill, “A decision procedure for bit-vectors and arrays,” in *19th CAV*, vol. 4590, 2007, pp. 519–531.
[52] I. Medeiros, N. F. Neves, and M. Correia, “Automatic detection and correction of web application vulnerabilities using data mining to predict false positives,” in *23rd WWW*, C. Chung, A. Z. Broder, K. Shim, and T. Suel, Eds. ACM, 2014, pp. 63–74.
[53] L. K. Shar, H. B. K. Tan, and L. C. Briand, “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” in *35th ICSE*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 642–651.
[54] D. Sahoo, C. Liu, and S. C. H. Hoi, “Malicious URL detection using machine learning: A survey,” *CoRR*, vol. abs/1701.07179, 2017.