Effective Identification of Security Issues using Machine Learning

Yang Chen
CA Technologies
Singapore
yang.chen@ca.com

Andrew E. Santosa
CA Technologies
Singapore
andrewedward.santosa@ca.com

Ang Ming Yi
CA Technologies
Singapore
mingyi.ang@ca.com

Asankhaya Sharma
CA Technologies
Singapore
asankhaya.sharma@ca.com

***Abstract*—In this article we consider the problem of identifying *unidentified* software vulnerabilities of open-source software. An unidentified vulnerability is a vulnerability that is known, as they have been reported and possibly silently fixed, but is not publicly identified as a vulnerability, such as by being assigned a CVE id. We use the publicly-available open-source issue reports and pull requests as well as commit data to identify if a particular version of an 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 id, changed file paths, and patches, which include lines added and deleted.**
2) **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 on average improves the precision for vulnerability-related commits by 78% and for vulnerability-related issues by 50% for the same recall when compared with previous work.

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

I. Introduction

In recent years, there is an increasing dependency of the whole information infrastructure on open-source software. Open-source software may contain vulnerabilities. Sometimes these vulnerabilities are widely known and assigned CVE ids. However, often the vulnerabilities are reported and fixed without being identified as such. In this paper, we consider the problem of identifying *unidentified* software 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 [1] data, close to 50% of vulnerabilities in open-source software are not disclosed publicly with a CVE.

Recent years have also seen the rise of the machine learning technology, which has been widely used for program analysis [2]. Many of the works are for discovering new vulnerabilities, either automatically [3], [4] or for help in auditing [5], however, there is a growing number of works that instead deal with unidentified vulnerabilities. Perl et al. [6] classify commits as related to a CVE or not using machine learning. Unfortunately, this requires prior identification using CVE ids. Zhou and Sharma [7] explore the identification of vulnerabilities from commit messages and issues or bug reports using machine learning for natural languages. Compared to Perl et al. [6], 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 [7]. Their machine learning approach is a supervised one where true positives and negatives are identified by security researchers. The approach creates separate models for issue and bug reports data, and for commit messages data. Sabetta and Bezzi present a machine learning approach to security-related commits [8]. Differently to Zhao and Sharma, in addition to using commit messages, they also use 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 is usable 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 [7], where we similarly use the publicly-available open-source bug and issue reports as well as commit data. However, although commit messages have been considered by Zhou and Sharma [7], other parts of the commit data including committer id, changed file path, and the patch itself have not been considered. Zhou and Sharma [7] only consider commit messages, however, Sabetta and Bezzi [8] claim that considering commit patches can be useful in practice. In this article, we explore the inclusion of more commit data in the following two directions.

1) *We consider more components of the commit data*. Zhou and Sharma only consider commit messages of the entire commit data [7]. In this article, in addition we consider the committer's id, changed file paths, and commit patch, which is further divided into lines added and lines deleted.
2) *We consider the addition of of commit data into issues, bug reports, or pull requests vulnerability identification*. Zhou and Sharma treat commits and issues (including issues, bug reports, and pull requests) independently [7]. In this article, we explore the possibility of models

generated using the combination of an issue or a pull requests with commit data, in order to improve the performance of the identification of issues or pull requests as vulnerability-related.

We consider 20,000 commits from GitHub, and we demonstrate that adding extra commit data to commit message, in particular committer's id and file paths, is more effective in identifying vulnerability-related commits, resulting in higher precision with the same recall, when compared to both Zhou and Sharma [7] and Sabetta and Bezzi [8]. However, although adding patch information does improve the precision and recall when the recall is low, the improvement is insignificant with higher recall. In combining the commit data into the related GitHub issues or pull requests in order to identify the issues and pull requests to be security related or not, we use close to 3,000 data items. This is much less than the commit data we used before, since multiple commits may correspond to only a single GitHub issue or pull request. Even so, here also we discover a marked improvement to Zhao and Sharma [7] in precision given the same recall, signifying the effectiveness of our approach. Our new approach on average improves the precision for vulnerability-related commits by 78% and for vulnerability-related issues by 50% for the same recall. ¡ We start 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 first provide a summary of K-fold stacking model, which is used by Zhao and Sharma [7]. The article explores the identification of vulnerabilities from bug or issue reports, and commit messages using machine learning approach for natural languages. The machine learning approach is a supervised one, where models are created using existing data triaged by security researchers. Similar to [7], our machine learning approach is a also supervised one, where labeled data are used to train the machine learning models. The labeled data were manually triaged and labeled by security researchers.

K-fold stacking model is known to be effective for imbalanced data. We illustrate the model using Figure 1. Here, the dataset is split into K parts. For each such split, 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 experiments, we discover that using 10-fold 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 set of six classifiers as Zhou and Sharma [7]. The set includes random forest (RF), Gaussian Naive Bayes, K-nearest neighbors (K-NN), linear support vector machine (SVM), gradient boosting (GB), and AdaBoost (Ada). Our model generates a decision on whether a commit, issue or pull request corresponds to a vulnerability in terms of confidence value from 0 to 1, called the *probability of vulnerability relatedness*. We further abbreviate this term as *PVR*. This allows for an adjustment of a threshold to distinguish vulnerable and non-vulnerable items.

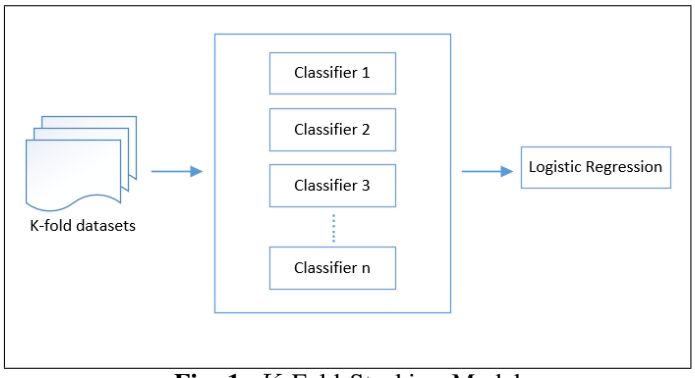

Fig. 1: K-Fold Stacking Model

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) Paths of the updated files.
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, encompassing commit messages, committer's ids and paths of the updated files.
2) Using all natural language features and the commit patch.

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 code artifact and the natural language artifacts[1]. In our approach for the commit patch, we treat the patch as natural language and use word2vec to extract feature vector from word2vec model. The word2vec model for commit patch is built separately with natural language features. We keep most of the content in patch except comment symbols and brackets. 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

[1]In GitHub's natural language semantic code search, their correlation needs to be modeled using machine learning [9].

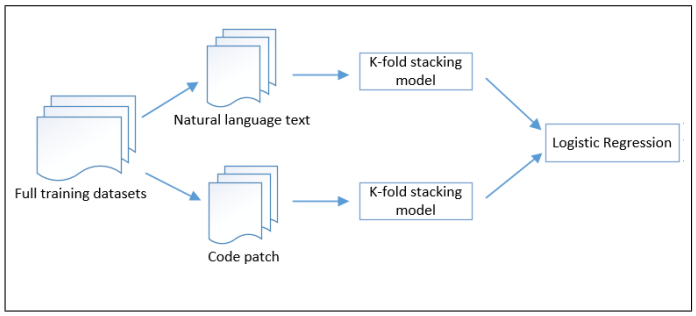

Fig. 2: Overall Model

results of the two models using logistic regression. The overall approach with the combining of the results is illustrated in Figure 2.

C. Enhancing Identification of Vulnerability-Related Issues and Pull Requests Using Commit Information

In our second approach, we consider the commit information that are related to the issues. While Zhou and Sharma [7] consider datasets form Jira and Bugzilla deployments and GitHub, here we only consider GitHub data since we use commit data from GitHub only. This is because 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, and we consider a dataset with about 3,000 records (see Table I). For about 2,000 of the records, we iterate over all the commits that we have to track the issues or pull requests if they were mentioned in the commits. For the rest of about 1,000 records, we iterate over all of our issues or pull requests data and track the related commits from them if any, as many issues or pull requests do not have related commits.

Similar to our approach with identifying vulnerability-related commits, here we also combine the results of two models using logistic regression (Figure 2). The difference here is that the natural language features now include issue and pull request texts. Here, an issue or pull requests may be related to multiple commits. Therefore, given a single issue or pull request, we append all the related commits into contiguous texts by commits message, committer's id and changed file path. We then use word2vec to compute the feature vector from the text. For the patch itself, we append all the lines added and deleted across all of the commits that correspond to an issue or a pull request into two contiguous texts, and then also use word2vec to compute a feature vector from the text. So for each issue or pull request, the patch part has one lines added feature and one lines deleted feature.

III. Evaluation

A. Setup

The dataset sizes that we use are shown in Table I. The second column shows the numbers of positive and negative data for our commits dataset, whereas the third column shows the numbers of positive and negative data for our issue or pull request and commit pairs. As mentioned, our labeled dataset

	Commit	(Issue/PR,Commit)
Positive	3,989	1,396
Negative	16,011	1,548

TABLE I
Sizes of Our Datasets

for the commits is an unbalanced one, having positive data of only 20% of the total number of data, hence we use K-fold stacking model. For generating the models we employ algorithms provided by the scikit-learn [10] library. We use logistic regression to combine the results across all the models. The models we generate map a source data item (issues/pull requests or commits) to a PVR value from 0 to 1.

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. 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 [7], 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:

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \quad (1)$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (2)$$

The reasons that we target these two metrics are:

1) Precision reflects the ratio of true and false positives. In unbalanced scenario like ours, it helps us focus on the true vulnerabilities. The overall ratio of vulnerability-related items in our datasets is less than 24%. That is, if manual effort is devoted to checking the data, 76% 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. [11]), 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 our approach is to cover all of the vulnerability-related commits and bug reports. The higher the recall value, the more the actual such commits and bug reports are reported by our approach.

C. Results in Identifying Vulnerability-Related Commits

We generate models using commit data only and answer the following research questions:

- *RQ1. Can we effectively identify vulnerable software using the commit data only?* Here we use 20,000 commit

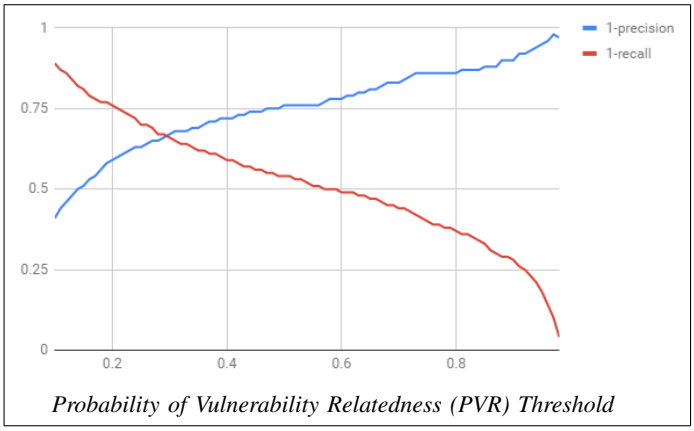

Fig. 3: Precision and Recall for the Identification of Vulnerability-Related Commits without Considering Patch

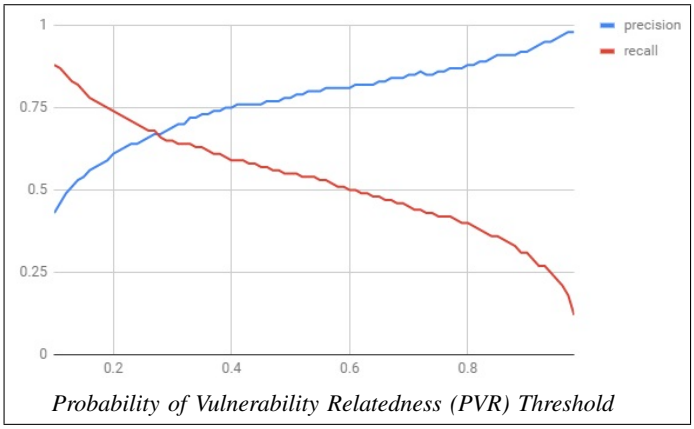

Fig. 4: Precision and Recall for the Identification of Vulnerability-Related Commits with Considering Patch

data manually triaged by security researchers. The same question has been answered by Zhao and Sharma, and by Sabetta and Bezzi [7], [8], 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 improve the identification quality of vulnerability-related commits?* Sabetta and Bezzi [8] 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.

We have two trainings on our commit datasets. One is to train our model using the natural language features only (commit message, committer's id, changed file paths). We show the precision and recall results for this training, 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 is shown in Figure 4.

In Table II, we show the precision of our models given fixed recall. The recall values are the values selected by Sabetta and Bezzi in presenting their results in [8]. In the same table we also show the results of Zhao and Sharma (second column) and Sabetta and Bezzi (third column) [7], [8]. As can be seen from the table, our results outperform both Zhao and Sharma's and Sabetta and Bezzi's results. The fourth and fifth columns show our precision results with and without the usage of commit patch, respectively. This shows that our approach is practical, and we can answer RQ1 in the affirmative. Table II also provides an answer to RQ2. The commit patch does improve the identification quality of the vulnerability-related commits when the recall is low, but improvement is insignificant when recall increases. As can also be observed from Figures 3 and 4, the precision and recall have similar curves for the cases without and with commit patch.

Recall	Precision			
	[7]	[8]	Ours w/o Patch	Ours w/ Patch
0.50	0.50	0.74	0.76	0.81
0.72	0.34	0.57	0.63	0.63
0.76	0.31	0.56	0.59	0.58

TABLE II
PRECISON WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED COMMITS

Recall	Precision		
	[7]	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 III
PRECISON WRT. RECALL FOR THE IDENTIFICATION OF VULNERABILITY-RELATED ISSUES/PULL REQUESTS

D. Results in Enhancing Identification of Vulnerability-Related Issues and Pull Requests Using Commit Data

Here we answer the following two research questions:

- *RQ3. Does the addition of commit information improve the quality of the vulnerability identification for the related issues or pull requests?* The identification of issues as vulnerability related using machine learning has been attempted by Zhou and Sharma [7]. 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 issues or the pull requests, are easier to identify.
- *RQ4. 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 in furthermore in addition use lines added and lines deleted from the patch. We compare the models generated in the two experiments.

Figure 5 shows the comparison of precision and recall of the identification of vulnerability-related issues/pull requests with using the commit data but without using commit patch.

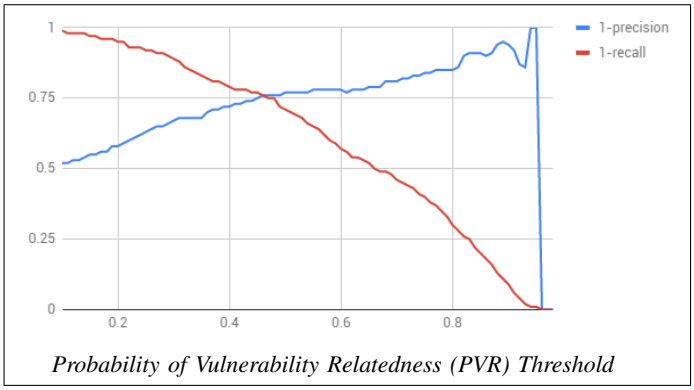

Fig. 5: Precision and Recall for Identification of Vulnerability-Related Issues/Pull Requests without Considering Patch

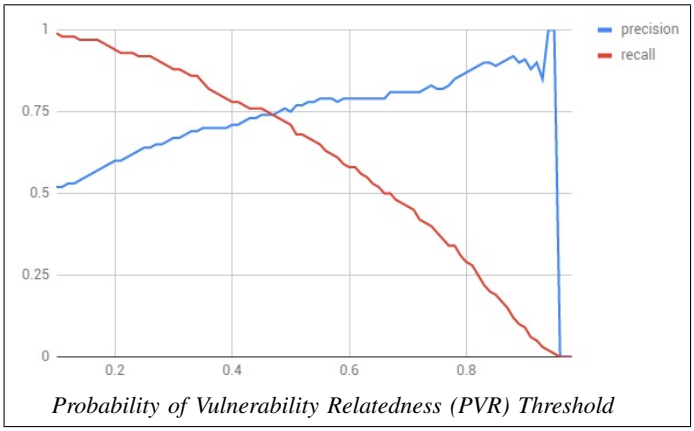

Fig. 6: Precision and Recall for Identification of Vulnerability-Related Issues/Pull Requests with Considering Patch

Figure 6 shows the comparison of precision and recall of the identification of vulnerability-related issues/pull requests with using the commit data and with using 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 no item's PVR is higher than the threshold, so the precision is 0.

Table III shows the precision, given a set of recall, where we include the results of [7] in the second column. From Table III, 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 5 and 6, 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 help the identification of vulnerability-related issues or pull requests, and we can therefore answer RQ3 in the affirmative. Similar to the result for RQ2, for RQ4 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.

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, committer's id may contribute significantly to the identification of vulnerability-related commits, and since different persons may be involved in different projects, this indicates that the results should also depend on the projects that we have selected in our dataset.
- Similar to Zhou and Sharma [7], in building our datasets, we filtered out commits and issues or bug 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. Results without filtering or with different filtering may be different.

B. Committer's Id as Feature

It is identified by the authors of [7] that the results for bug reports are better than those for commit messages. This is because bug reports have richer text information than commits, and more useful features have been used to train the model for bug 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 [12]. We postulate that the more advanced developers are more likely to be assigned the task to correct a vulnerability. We have shown experimentally that considering committer's name 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, 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 [3]–[5]. The use of code-specific features therefore is a possible future direction to be attempted in improving our approach.

V. Related Work

Our work is related to various program analysis techniques in the literature for finding vulnerabilities, particularly machine learning techniques. In this section we compare our approach

with 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 [13], IST4 [14] for C/C++, RATS [15] for multiple languages (C, C++, Perl, PHP, and Python), and Clang static analyzer for C and C++ [16]. 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 [17] analyzes Java bytecode and source code to discover bug patterns and Parfait [11] 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 [18], and Parfait has a low false positive rate [11], their precisions are unknown.

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 [19]–[21]. PHP Aspis does dynamic taint analysis to identify XSS and SQL vulnerabilities [21]. ZAP [19] 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 [22] 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 [23] used in ZAP and Peach fuzzer [24]. Grey-box fuzzers execute the PUT with a limited knowledge about the PUT. Well-known grey-box fuzzers are AFL [25] and libFuzzer [26]. 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 [27] to achieve high path coverage. However, they suffer from scalability issues. Well-known white-box fuzzers include DART [28], CUTE [29] and jCUTE [30].

Symbolic execution [27] 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 [31]. 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 [32] and STP [33].

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. [34] detects false positives in static program analysis bug report using machine learning. Shar et al. [35] focus on SQL injection and cross-site scripting (XSS). Others perform bug detection using non-program-code artifacts. Zhou and Sharma [7] employ commit messages and bug reports to identify security issues in software. Wijayasekara et al. [36] propose an approach to mine bug databases for unidentified software vulnerabilities. Sahoo et al. [37] provide a survey on malicious URL detection approaches using machine learning. Sabetta and Bezzi present a machine learning approach to identify commits that are related to security [8] using commit message and patch. Their model is shown to have 80% precision and 43% recall. We propose a different approach to theirs, where other than considering commit message and patch, we also consider committer's id, and changed file paths. In addition, we explore the problem of identifying vulnerable issues or pull requests with added commit data. Ghaffarian and Shahriari provide a survey on software vulnerability detection using machine learning [2].

Different to most of the work mentioned above, our work is on finding vulnerabilities that are unidentified. In this area, Perl et al. [6] classifies commits as related to a CVE or not. Our work is most closely related to the work of Zhou and Sharma [7], 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. [6], 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. Zhou and Sharma's dataset was obtained from publicly-available Jira [38] issue tracking systems, Bugzilla [39] bug trackers, and GitHub [40] issues and pull requests. For each source,

(Jira, Bugzilla, or GitHub), they generate separate models using an ensemble of classifiers. The dataset of Zhou and Sharma is imbalanced, where the number of vulnerability-related commits and bug reports are small (10%) compared to the whole training data. For training, therefore, the approach uses K-fold stacking model, known to be effective for imbalanced data. The machine learning approach is a supervised one where true positives and negatives are manually identified. Decision on whether a commit or an issue or bug report is vulnerability related or not is in terms of a confidence from 0 to 1. This allows for an adjustment of a threshold to decide vulnerability-related and non-vulnerability-related items. For commit messages, the precision and recall are respectively 0.50 and 0.51 under a threshold of 0.75, and when deployed in production, with a threshold of 0.75, the precision and recall are respectively 0.83 and 0.74. Compared to Zhou and Sharma, in identifying vulnerability-related commits, we include committer'S id, changed file paths, and patches. In identifying vulnerability-related issue reports (including pull requests), we pair the issue reports with their commit data, considering only issue reports which have commit data, and extract features from both the issues or pull requests and commit data. We show improvements in these two directions.

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 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 by extracting features from more parts of the commit data, including committer's id and updated file paths. We also consider pairing commits and their corresponding issues or pull requests to improve the identification effectiveness of vulnerability-related issues or pull requests. We experimentally demonstrated that our new approach improved the performance of the previous approaches.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," *CoRR*, vol. abs/1807.02458, 2018. [Online]. Available: http://arxiv.org/abs/1807.02458

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

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

[11] 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.

[12] 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.

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

[14] 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.

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

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

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

[25] M. Zalewski, "American fuzzy lop (2.52b)," http://lcamtuf.coredump.cx/afl/.

[26] "libFuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html.

[27] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *14th TACAS*, vol. 4963, 2008, pp. 337–340.

[33] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *19th CAV*, vol. 4590, 2007, pp. 519–531.

[34] 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.

[35] 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.
[36] 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.
[37] D. Sahoo, C. Liu, and S. C. H. Hoi, "Malicious URL detection using machine learning: A survey," *CoRR*, vol. abs/1701.07179, 2017.
[38] "Jira – issue & project tracking software – Atlassian," https://www.atlassian.com/software/jira.
[39] "Home :: Bugzilla :: bugzilla.org," https://www.bugzilla.org/.
[40] "The world's leading software development platform – GitHub," https://github.com/.