Automated Identification of Security Issues from Commit Messages and Bug Reports

Yaqin Zhou
SourceClear, Inc.
yaqin@sourceclear.com

Asankhaya Sharma
SourceClear, Inc.
asankhaya@sourceclear.com

ABSTRACT

The number of vulnerabilities in open source libraries is increasing rapidly. However, the majority of them do not go through public disclosure. These unidentified vulnerabilities put developers' products at risk of being hacked since they are increasingly relying on open source libraries to assemble and build software quickly. To find unidentified vulnerabilities in open source libraries and secure modern software development, we describe an efficient automatic vulnerability identification system geared towards tracking large-scale projects in real time using natural language processing and machine learning techniques. Built upon the latent information underlying commit messages and bug reports in open source projects using GitHub, JIRA, and Bugzilla, our K-fold stacking classifier achieves promising results on vulnerability identification. Compared to the state of the art SVM-based classifier in prior work on vulnerability identification in commit messages, we improve precision by 54.55% while maintaining the same recall rate. For bug reports, we achieve a much higher precision of 0.70 and recall rate of 0.71 compared to existing work. Moreover, observations from running the trained model at SourceClear in production for over 3 months has shown 0.83 precision, 0.74 recall rate, and detected 349 hidden vulnerabilities, proving the effectiveness and generality of the proposed approach.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

vulnerability identification, machine learning, bug report, commit

ACM Reference format:
Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117771

1 INTRODUCTION

To aid in the software development, companies typically adopt issue-tracking and source control management systems, such as GitHub, JIRA, or Bugzilla. In fact, as of April 2017, GitHub reports having almost 20 million users and 57 million repositories. According to Atlassian, JIRA is used by over 75,000 companies. These tools are very popular for open source projects, and are essential to modern software development. Developers work on reported issues in these systems, then commit corresponding code changes to GitHub (or other source code hosting platforms, e.g. SVN or BitBucket). Bug fixes and new features are frequently merged into a central repository, which is then automatically built, tested, and prepared for a release to production, as part of the DevOps practices of continuous integration (CI) and continuous delivery (CD).

There is no doubt that the CI/CD pipeline helps improve developer productivity, allowing them to address bugs more quickly. However, a significant amount of security-related issues and vulnerabilities are patched silently, without public disclosure, due to the focus on fast release cycles as well as the lack of manpower and expertise. According to statistics collected at SourceClear [2], 53% of vulnerabilities in open source libraries are not disclosed publicly with CVEs. For Go and JavaScript language libraries, the percentage is as high as 78% and 91% respectively.

In today's world of agile software development, developers are increasingly relying on and extending free open source libraries to get things done quickly. Many fail to even keep track of which open source libraries they use, not to mention the hidden vulnerabilities that are silently patched in the libraries. So even if the vulnerability is fixed in source code as seen by the commit message or bug report, there may be users of the library who are not made aware of the issue and continue using the old version. Unaware of these hidden vulnerabilities, they are putting their products at risk of being hacked. Thus, it is important to locate such silent fixes as they can be used to help developers decide about component updates.

Motivated to find the unidentified vulnerabilities in open source libraries and secure modern software development, in this paper we describe an automatic vulnerability identification system that tracks a large number (up to 10s of thousands) of open source libraries in real time at low cost.

Many techniques, such as static and dynamic analysis, are employed to flag potentially dangerous code as candidate vulnerabilities, but they are not appropriate for tracking existing unknown vulnerabilities on a large scale, at low cost (due to false positives). Firstly, these tools can only support one or two specific languages and certain patterns of vulnerabilities well. For example, the static analysis tool FlawFinder [19] is limited to finding buffer overflow risks, race conditions, and potential shell meta-character usage in C/C++. Moreover, most of these approaches operate on an entire

ESEC/FSE'17, September 4-8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3117771

software project and deliver long lists of potentially unsafe code with extremely high false positive rates (i.e., 99% shown in [14]). They require a considerable amount of manual review and thus are not suitable to track projects at large scale.

For most software systems, bugs are tracked via issue trackers and code changes are merged in the form of commits to source control repositories. Therefore, it is convenient to check these basic artifacts (a new bug report or commit) of software development to detect vulnerabilities in real time. Besides, source code changes, bug reports and commits messages contain rich contextual information expressed in natural language that is often enough for a security researcher to determine if the underlying artifact relates to a vulnerability or not.

Building on these insights, we automate the above identification process via natural language processing and machine learning techniques. From GitHub, JIRA, and Bugzilla, we collected a wide range of security related commits and bug reports for more than 5000 projects, created between Jan. 2012 and Feb. 2017, to build commits and bug reports datasets with ground truth. The collected data, however, demonstrates a highly imbalanced nature, where vulnerability-related commits and bug reports are less than 10%. Hence, it is challenging to train classifiers to identify vulnerabilities with good performance. To address it, we design a probability-based K-fold stacking algorithm that ensembles multiple individual classifiers and flexibly balances between precision and recall.

We summarize our major contributions below.

- We present an automatic vulnerability identification system that flags vulnerability-related commits and bug reports using machine learning techniques. Our approach can identify various vulnerabilities regardless of programming languages at low cost and large scale. Our proposed K-fold stacking model for commits outperforms the state of the art SVM model by 54.55% in precision. The combined model for bug reports achieves an a precision of 0.70 and recall rate of 0.71, which is even better than the commits considering that the data are more imbalanced.
- We present an extensive experimental and production evaluation to validate our proposed approach. Specifically, we integrate the trained model for commits into our production system at SourceClear to track new projects and languages. 3 months of use in production has shown encouraging results, where the model identifies vulnerabilities with a precision and recall rate of 0.83 and 0.74 respectively. Moreover, during the same period, it identified 349 hidden vulnerabilities as compared to 333 CVEs.

2 RELATED WORK

We now compare prior work in static analysis, dynamic analysis, symbolic execution, and machine learning with our approach for vulnerability identification.

Static analysis is a way of analyzing source code or binary without actually executing it. A significant part of effort in static vulnerability detection has been directed towards analyzing software written in high-level languages, e.g. FlawFinder [19] and IST4 [18] for C/C++, RATS for multiple languages (C, C++, Perl, PHP and Python). However, these analyzers are language-specific and even for supported languages may have cases where they fail to find the underlying issues due to imprecision of analysis. For example, RATS does not find Cross-Site Scripting (XSS) or SQL Injection vulnerabilities. Moreover, when applied to real world projects, these tools raise massive false positives that are hard to reduce.

Dynamic analysis analyzes the source code by executing it on real inputs. Basic dynamic analysis (or testing) tools search for vulnerabilities by trying a wide range of possible inputs. There are also dynamic taint analysis tools that do taint tracking at runtime [1, 3, 13]. For example, PHP Aspis does dynamic taint analysis to identify XSS and SQL vulnerabilities [13]. ZAP [1], a popular testing tool in industry, finds certain types of vulnerabilities in web applications. It requires users to define scan policies before scanning and perform manual penetration testing after scanning.

Symbolic execution [8] is a technique that exercises various code paths through a target system. Instead of running the target system with concrete input values like dynamic analysis, a symbolic execution engine replaces the inputs with symbolic variables which initially could be anything, and then runs the target system. Cadar et al. [5] present KLEE, an open-source symbolic execution tool to analyze programs and automatically generate system input sets that achieve high levels of code coverage. However, it requires manual annotation and modification of the source code. Also, like most symbolic execution tools, runtime grows exponentially with the number of paths in the program which leads to path explosion.

Machine learning. Besides the above techniques that focus exclusively on the raw source code, machine-learning techniques provide an alternative to assist vulnerabilities detection by mining context and semantic information beyond source code. Among these works, some [11, 16, 17, 20, 20] focus on detecting vulnerabilities combined with program analysis. For instance, Shar et al. [17] focus on SQL injection and cross-site scripting while Sahoo et al. [16] investigate malicious URL detection.

The aforementioned works are related to finding new vulnerabilities in source code. Our work and others described below are related to find existing vulnerabilities that are unidentified. The work most similar to ours is from Perl et al. [14] that classifies if a commit is related to a CVE or not. They mapped 718 CVEs to GitHub commits to create a vulnerable commit database which includes 66 C/C++ GitHub projects, 640 vulnerability-contributing commits, and trained a SVM classifier to flag suspicious commits. The CVE IDs are very unique features to identify the vulnerability-contributing commits, which make the learning task much easier. Extending and building upon their work, we find hidden vulnerabilities without assigned CVE IDs among 5000+ projects and 6 languages. Our experiment results in Section 4 show that the linear SVM classifier adopted in [14] cannot handle highly imbalanced data sets without the unique CVE ID features. Thus, the best result for the state of the art linear SVM based classifier on our commit dataset has a precision rate of 0.22 with 0.72 recall. Under the same recall rate, the precision of our proposed approach is 54.55% higher.

3 APPROACH

We now present the design of a commit-message/bug-report based vulnerability identifier developed using supervised machine learning techniques. Our identifier extracts a wide range of security-related information from the commits/bug reports stream in real

Table 1: Examples of regular expression used to filter out security-unrelated issues

<table>
<tr><th>Rule name</th><th>Regular Expression</th></tr>
<tr><td>strong_vuln _patterns</td><td>(?i)(denial.of.service|\bXXE\b|remote.code.execution|\bopen.redirect|OSVDB|\bvuln|\bCVE\b |\bXSS\b|\bReDoS\b|\bNVD\b|malicious|x − frame − options|attack|cross.site|exploit|directory.traversal|\bRCE\b|\bdos\b|\bXSRF\b|clickjack|session.fixation|hijack|advisory|insecure|security |\bcross − origin\b|unauthori[z|s]ed|infinite.loop)</td></tr>
<tr><td>medium_vuln _patterns</td><td>(?i)(authenticat(e|ion)|bruteforce|bypass|constant.time|crack|credential|\bDoS\b|expos(e|ing)|hack |harden|injection|lockout|overflow|password|\bPoC\b|proof.of.concept|poison|privelage|\b(in)?secur (e|ity)|(de)?serializ|spoof|timing|traversal)</td></tr>
</table>

time, and locates the tiny portion of vulnerability-related commits/bug reports among the massive data.

3.1 Data Collection

We collected data from GitHub, JIRA, and Bugzilla to track the source code changes and reported bugs of open source projects. For each of the three sources, we tracked projects covering the following 6 programming languages: Java, Python, Ruby, JavaScript, Objective C and Go. We collected all the commits, pull requests and issues from 5002 projects in GitHub, all the issue tickets of 1310 projects in JIRA, and all the bug reports of 2224 projects in Bugzilla. As the format and provided information of GitHub pull requests, issues, JIRA tickets and Bugzilla reports are similar, we normalized them to a uniform bug report format. Considering the significant amount of information in such a large-scale tracking task, we filtered out the commits and bug reports clearly unrelated to security by matching with a set of regular expression rules. To cover all possible vulnerabilities, our regular expression rules included almost all possible expressions and key words related to security issues, such as security, vulnerability, attack, CVE, DoS, and XSS etc. (Please refer to Table 1 for part of the rule set). Consequently, we have a GitHub commits dataset, GitHub bug reports dataset (including pull requests and issues), JIRA bug reports dataset, and Bugzilla bug reports dataset.

We used the data from Jan. 2012 and Feb. 2017 for initial training. In the commit dataset, out of the 12409 collected commits during this period, 1303 are vulnerability-related. In the bug report datasets, 612 out of 10414 bug reports from GitHub, 204 out of 11145 from JIRA, and 1089 out of 2629 from Bugzilla are vulnerability-related. Thus, except Bugzilla (that has a more balanced ratio between issues related and unrelated to vulnerabilities), other data sources are highly imbalanced in nature.

Ground truth. We have a team of professional security researchers who manually investigated the collected data. The security researchers checked every single commit and bug report. The overall effort took almost one man year. To ensure the accuracy of results, for an entry (a commit or bug report) that is related to a vulnerability, our security researchers conduct at least two rounds of analysis on it. In the first round, one security researcher will first check if the vulnerability is published publicly in National Vulnerability Database (NVD) or the SourceClear Registry[2], then analyze the exploitation process, CVSS score, vulnerable code, affected versions, and document it in a vulnerability report. In the second round, another security researcher will verify the vulnerability report by examining all the details, then publish it on the SourceClear Registry. In addition, all disputed reports are set aside for team discussion before a final decision. The data can be accessed from SourceClear Registry, where the data for CVEs are free and the data for vulnerabilities without CVEs are commercially accessible.

3.2 Classifier Features

Commits. The initial features collected in this study for commits included commit messages, comments, project name, and the name of the person who committed the code. We finally choose *commit messages* as the only feature, because 1) only a few commits have comments, 2) we exclude the project name because we want to apply the trained model to more projects that are not in the current data set, 3) we can not get information about whether a person belongs to the development team of the projects or not; different persons may share the same name, and one person may change their name over time, which may cause inaccuracy.

We use the word2vec embedding method [12] to transform commit message text to numerical vectors. After tuning the parameters for the word2vec models, we decided to use a 400-dimensional vector to represent a commit message. We build the word2vec model over 3 million commit messages without being filtered from the regular expression matching. This gives us better performance over the word2vec model trained on only filtered commit messages.

Bug Reports. We select *title, description, comments, comment number, attachment number, labels, created date,* and *last edited date* as the features, as this information is generally available in most of the bug reports, and can provide potential signals for vulnerabilities. Among the selected features, title, description, comment, and labels are text features that contain the semantic information of the report. The numbers of comments and attachments are numeric features which may reflect the attention and resolution a bug report received. Similarly, the difference between the created and last edited date reflects the resolution time.

As with commits, we use word2vec to obtain the vectors for representation of text features. In this case, we find 200-dimensional vectors are enough as larger dimensional vectors do not improve the model but severely slow down the training process.

3.3 Classifier Training and Design

We compare a number of classification algorithms that are reported to perform well for imbalanced datasets and natural language processing in literature, including SVM, random forest, Gaussian naive Bayes, K-nearest neighbors, AdaBoost, and gradient boosting [7, 9]. However, the performance of a single classifier is inadequate. Thus we propose a K-fold stacking algorithm that ensembles multiple

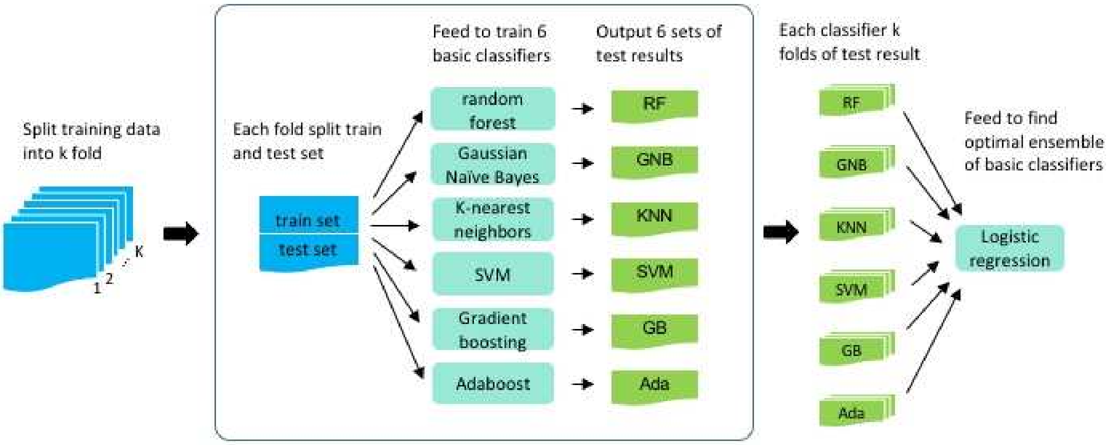
Split training data into k fold
1
2
K
Each fold split train and test set
train set
test set
Feed to train 6 basic classifiers
random forest
Gaussian Naive Bayes
K-nearest neighbors
SVM
Gradient boosting
Adaboost
Output 6 sets of test results
RF
GNB
KNN
SVM
GB
Ada
Each classifier k folds of test result
RF
GNB
KNN
SVM
GB
Ada
Feed to find optimal ensemble of basic classifiers
Logistic regression

Figure 1: K-fold Stacking Model

classifiers to address the difficulty brought by the highly imbalanced nature of the dataset.

Fig. 1 illustrates the flow of K-fold stacking model. First, we split the training set in K parts. For each fold $1 \leq k \leq K$, the kth part of data will be used as the test data, while the rest of the $K - 1$ parts serve as the training data which a set of individual classifiers is trained over. We choose random forest (RF), Gaussian Naive Bayes (GNB), K-nearest neighbors (KNN), linear SVM, gradient boosting (GB), and AdaBoost (Ada) as the basic classifiers in the set. For every basic classifier, the trained model is tested on the kth part of data. After the K folds training, each classifier has K sets of test results where the union is the whole training set. We feed the test results of all the basic classifiers to a logistic regression, to find the best ensemble of the set of classifiers.

4 EVALUATION

We evaluate the effectiveness of our automatic vulnerability identification system in several aspects. First, we shuffle and split the dataset into training and test sets to evaluate the predictiveness of the stacking algorithm. We compare it with several individual classifiers, including the state-of-the-art linear SVM described in [14, 15]. Table 2 summarizes the distribution of the commits and bug reports. We define the *imbalanced ratio* as the ratio between the number of positive samples and negative samples. During this step, we conducted over a hundred different experiments for tuning the parameters on embedding word models (from bag-of-words to word2vec), sampling techniques to balance the data (e.g., SMOTE [4, 6], BalanceCascade [10]), a number of individual classifiers, and ensemble methods. We present the results of the tuned parameters based on the best prediction results. To validate the generality of our system, we deployed the automatic vulnerability identification system in production for real-time vulnerability tracking on 2000+ new projects that had no records in training data, and compared the predicted results with the ground truth.

Evaluation metrics. To measure vulnerability prediction results, we use two metrics: *Precision* and *Recall rate* (or true positive rate). Here is a brief definition:

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

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

The reasons that we target these two metrics are:

(1) Precision reflects the ratio of true and false positives, in unbalanced scenarios like here it helps us focus on true vulnerabilities. The overall ratio of vulnerability-related items in our datasets is less than 8% percent. That is, if manual effort is devoted to checking the data, 92% of that time would be spent on false-positive items. Therefore, a high precision would save a lot of manual work from false positives. This also explains that why we don't use false positive rate directly as it can not intuitively reflect the percentages of false positives among predicted positives.

(2) Recall rate indicates the coverage of existing vulnerabilities. We aim to cover all the vulnerability-related commits/bug reports, where a higher recall rates means only a smaller percentage of vulnerabilities are missed (i.e., commits/bug reports predicted as vulnerability-unrelated even though they are actually related to a vulnerability).

Probability-based Classification. It is challenging to achieve both high precision and recall rate as the two are in conflict with each other. However, we can select a reasonable tradeoff between the two metrics. To achieve this, instead of directly outputting a binary prediction result to classify if an entry is related to vulnerability or not, the stacking model calculates the probability of vulnerability-relatedness. Thus, it is flexible to set the probability threshold to balance the precision and recall rate.

Table 2: Distribution of commits and bug reports

Data	Imbalance ratio	Train (positive)	Train (negative)	Test (positive)	Test (negative)
Commits	0.1050	997	8309	306	2797
BR_Github	0.0588	449	7361	163	2441
BR_JIRA	0.0183	150	8208	54	2733
BR_Bugzilla	0.4142	802	1169	287	371
BR_Combined	0.0788	1414	16727	491	5556

Note: BR is abbreviation of bug report

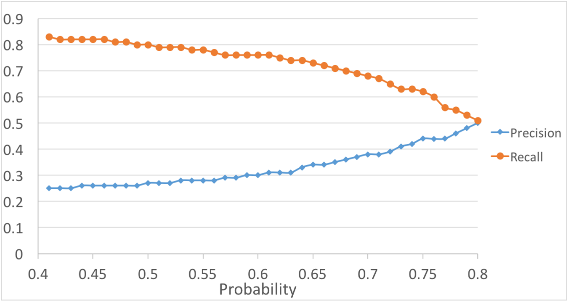

Figure 2: Identification performance of our stacking approach under commits

Table 3: Comparison with basic classifiers under the same recall rate in commits

Classifier	Recall rate	Precision (compared classifier vs.stacking)
Linear SVM	0.72	0.22 vs. 0.34
Logistic Regression	0.76	0.22 vs. 0.31
Random Forest	0.76	0.19 vs. 0.31
Gaussian Naive Bayes	0.77	0.14 vs. 0.28

4.1 Commits

Fig. 2 illustrates the tradeoff between the precision and recall rate for the commits dataset based on our algorithm. As the probability threshold increases, the precision increases while the recall rate decreases. Under the probability of 0.75, the precision and recall rates are almost the same, respectively 0.50 and 0.51. Table 3 compares the best results of linear SVM, logistic regression, random forest, and Gaussian Naive Bayes with our model under the same recall rate. It shows that our model improves precision by at least 0.12.

4.2 Bug Reports

Bug reports have 3 data sources. We train a model for each source, and an additional combined model with a source feature, indicating whether the bug report is from GitHub, JIRA, or Bugzilla.

Fig.3 shows the precision and recall rate for GitHub, JIRA, and Bugzilla bug reports, and the combined bug report model. From Fig.3, the trained model for Bugzilla has best performance due to a well-balanced dataset, while the trained model for JIRA has worst performance due to an extremely imbalanced ratio of 0.0183. Overall, the combined model achieves encouraging performance in both precision and recall rate (respectively 0.70 and 0.71), which is much better than the commit dataset even though it has a more imbalanced structure (imbalance ratio is 0.1050 for commits vs. 0.0788 for bug reports).

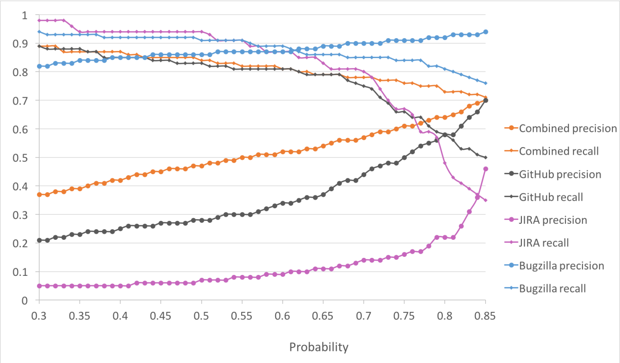

Figure 3: Identification performance of our stacking approach under bug reports

4.3 Production Observations

We deployed our trained commit model on SourceClear production system and evaluated the model for initial 3 months (March 2017-May 2017). We choose the 12-fold stacking model with probability threshold 0.75 (test precision 0.44 and recall 0.62) to deploy, with the number of projects increased from the initial 2070 to 5002.

Table 4 shows the validation results at the production system, where the precision and recall rate are 0.83 and 0.74 respectively. This is 88.63% and 19.35% higher than the test results.

To validate the value on finding hidden vulnerabilities, we only count the number of hidden vulnerabilities without CVEs found by our automatic vulnerability detection system though it flags vulnerabilities with CVEs as well, and compare it with the total number of published vulnerabilities with final and reserved CVEs from NVD and other official sources during the same period. The number of hidden vulnerabilities released on SourceClear platform are 66, 69, and 214 in March, April, and May, while the number of public vulnerabilities with CVEs are respectively 123, 80, and 130. In May we have a surge in the hidden vulnerabilities because we added a new language Go to the platform. Consequently, during the same period, the total number of found hidden vulnerabilities is 349, even larger than the total number of CVEs 333.

Table 4: Validation of performance on production

Projects	Commits (Total)	Commits (Positive)	Commits (Negative)	True positive	False positive
5002	2268	215	2053	160	32

5 DISCUSSION

Our experiments demonstrate that our approach can automatically spot unidentified vulnerabilities from commits and bug reports with high precision and recall rate. Compared to manual checking and other vulnerability identification techniques, it provides an efficient but low-cost way to track and detect vulnerabilities in real time, securing the development processes of various projects that use open source libraries.

Especially notable is the fact that we applied the trained model for commits in production, where we added Ruby and Go languages that were not covered in the training data, and increased the number of projects from 2070 in the model training process to 5002. The 0.83 precision and 0.74 recall rate, which is even better that the test result, has shown the generality and scalability of our approach. The reason for that is that we use the features that are not language-specific, and we exclude the project name. Consequently, we believe that applying our results to other new projects and languages will not threaten validity.

In evaluation on test data, we see that the model trained for bug reports achieves a much better result than the model trained for commits. There are several reasons for this: 1) Bug reports have richer text information than commits. Commit messages summarize the code changes that have fixed an issue, thus they usually tend to be short and concise, while bug reports are submitted to describe an issue that waits for a fix, thus usually they have a detailed description. 2) We have collected more useful features to train the model for bug reports. For commits we only obtain commit messages that are useful. Short commit messages may be not unique enough to reflect the distinction between vulnerability-relatedness and vulnerability-unrelatedness. In our future work, we will fetch and analyze more features to improve the model.

6 CONCLUSION AND FUTURE WORK

We have designed and implemented an automated vulnerability identification system based on commits and bug reports collected from thousands of open source projects. It can identify a wide range of vulnerabilities, including undisclosed vulnerabilities without CVE IDs, and significantly reduce false positives by more than 90% when compared to manual effort. The experimental results on test data and validation data from production show that it is promising to utilize machine-learning techniques to track and detect hidden vulnerabilities in open source projects.

Our work demonstrates an encouraging application of machine learning techniques to a realistic industrial problem of large-scale vulnerability identification in an economic way. We are still working to improve both the precision and recall rate of our system. In future work, we will explore more features (e.g., code changes, project properties), advanced machine learning techniques such as deep learning to learn hidden representations from the data, and neutral networks to train our models.

REFERENCES

[1] 2017. OWASP Zed Attack Proxy Project. (2017). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[2] 2017. SourceClear. (2017). https://www.sourceclear.com/

[3] Sruthi Bandhakavi, Prithvi Bisht, P Madhusudan, and VN Venkatakrishnan. 2007. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 12–24.

[4] Gustavo EAPA Batista, Ana LC Bazzan, and Maria Carolina Monard. 2003. Balancing Training Data for Automated Annotation of Keywords: a Case Study.. In *WOB*. 10–18.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[7] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2012. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 4 (2012), 463–484.

[8] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[9] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5, 4 (2016), 221–232.

[10] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. 2009. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39, 2 (2009), 539–550.

[11] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*. ACM, 63–74.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[13] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *Proceedings of the 2Nd USENIX Conference on Web Application Development (WebApps'11)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2002168.2002170

[14] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 426–437.

[15] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. 2015. Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-world Exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 1041–1056. http://dl.acm.org/citation.cfm?id=2831143.2831209

[16] Doyen Sahoo, Chenghao Liu, and Steven C. H. Hoi. 2017. Malicious URL Detection using Machine Learning: A Survey. *CoRR* abs/1701.07179 (2017). http://arxiv.org/abs/1701.07179

[17] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. 2013. Mining SQL Injection and Cross Site Scripting Vulnerabilities Using Hybrid Program Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 642–651. http://dl.acm.org/citation.cfm?id=2486788.2486873

[18] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. 2000. ITS4: a static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*. 257–267.

[19] David A. Wheeler. 2017. Flawfinder. (2017). https://www.dwheeler.com/flawfinder/

[20] Dumidu Wijayasekara, Milos Manic, Jason L. Wright, and Miles McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *Proceedings of the 2012 5th International Conference on Human System Interactions (HSI '12)*. IEEE Computer Society, Washington, DC, USA, 89–96. https://doi.org/10.1109/HSI.2012.22