2018 IEEE International Conference on Software Maintenance and Evolution

A Practical Approach to the Automatic Classification of Security-Relevant Commits

Antonino Sabetta and Michele Bezzi
SAP Security Research
{antonino.sabetta,michele.bezzi}@sap.com

***Abstract*—The lack of reliable sources of detailed information on the vulnerabilities of open-source software (OSS) components is a major obstacle to maintaining a secure software supply chain and an effective vulnerability management process. Standard sources of advisories and vulnerability data, such as the National Vulnerability Database (NVD), are known to suffer from poor coverage and inconsistent quality.**

To reduce our dependency on these sources, we propose an approach that uses machine-learning to analyze source code repositories and to automatically identify commits that are *security-relevant* (i.e., that are likely to fix a vulnerability). We treat the source code changes introduced by commits as documents written in natural language, classifying them using standard document classification methods.

By combining independent classifiers that use information from different facets of commits, our method can yield high precision (80%) while ensuring acceptable recall (43%). The use of information extracted from the *source code changes* yields a substantial improvement over the best known approach in the state of the art, while requiring a significantly smaller amount of training data and employing a simpler architecture.

I. Introduction

The software industry is facing the difficult challenge of reconciling the *opportunity* represented by the availability of mature, high-quality open-source software (OSS) components, with the *problem* of maintaining a secure OSS supply chain and an effective vulnerability management process. The lack of comprehensive, consistent, and up-to-date sources of vulnerability data represents a fundamental obstacle to tackle this challenge.

The current *de-facto* standard source of vulnerability data, the *National Vulnerability Database* (NVD)[1], is known to suffer from poor coverage: according to a recent study, as many as 25% of the OSS projects *silently fix* vulnerabilities without ever disclosing them in an official advisory [1]. The same study reports that only 10% of the OSS vulnerabilities receive a CVE (Common Vulnerabilities and Exposures) identifier and are published on the NVD. *Silent fixes* make it hard (if at all possible) for the clients of the affected OSS component to assess the urgency of upgrading to a newly released version, because upgrades that only bring new features or functional improvements are made indistinguishable from those that (also) correct security flaws.

Motivation：
1. 开源软件有静默修复行为
2. 即使有CVE也存在收录不及时的情况
3. 手动将commit链接到人工浪费大量人力

Furthermore, even the vulnerability advisories (CVEs), that *will eventually* be posted in the NVD, follow a process that has a somewhat *unpredictable timing*. In practice, the interval between the moment a vulnerability of an OSS component becomes *known to the public* (e.g., because a commit fixing it is pushed to the source code repository of the project) and the *actual availability of a new release* that includes that fix, can last a few days or even weeks (sometimes months). During such time-frame, a malicious party that would examine the code commits in an open-source project, could infer the existence of a security issue and exploit it before a fixed release is available through standard distribution channels. This risk is pushing the industry towards monitoring source code repositories in real-time and automatically, to detect new security-relevant changes in a timely manner, without the need of waiting for official advisories to be published.

Manually linking a vulnerability and the commit(s) that fix it requires considerable manual effort and expert knowledge, and is therefore expensive and error-prone. This problem motivated research on automated methods that could alleviate the cost of this activity and of vulnerability analysis in general [2], [3].

In this paper we present a novel approach to the automated identification of source code changes that are *security-relevant*. Our approach relies only on data available in code repositories, i.e., source code commits and their log messages, without the need of accessing vulnerability advisories. It allows a timely identification of security fixes, providing the necessary details to support a precise assessment of the impact of a OSS vulnerability using tools such as the one proposed by Ponta et al. [4]. To do so, inspired by the *naturalness* hypothesis [5], [6], we treat *source code* changes as *documents written in natural language* and we classify them using standard document classification methods.

我们提取commit中的源代码和log，采用自然语言分类的方法（我们将代码也视为自然语言），识别与安全相关的commit

We show that using information extracted directly from the source code, and combining independent classifiers suitably tuned, our method can yield high precision (80%) while ensuring a relatively good recall (43%). This result represents a substantial improvement over the best known approach in state of the art [3], while requiring a significantly smaller amount of training data and using a simpler architecture.

我们的准确率和找回率分别为80%和43%

The remainder of the paper is organized as follows. Sec. II positions our contribution with respect to the existing literature. Sec.III presents our approach to commit classification, whose quantitative evaluation is presented in Sec. IV. In Sec. V we elaborate on some important aspects of commit classification in practice. Finally, Sec. VI concludes the paper outlining the research directions we intend to pursue in the future.

[1] https://nvd.nist.gov/

II. Related Work

The research on applying machine learning to source code analysis has been very active in recent years. Recent surveys of the field can be found in [7] and in [5], where the emphasis is on approaches that, like ours does, rely on the *naturalness of software* [6].

Concerning the concrete problem of *commit classification*, the closest work to ours is [3], which presents an approach that uses an ensemble of six different models whose output is combined with an additional logistic regression model. Differently from us, the method of [3] classifies commits based *only on their log message*, whereas we also consider information extracted from the *source code changes*. Their architecture is more complex than ours, and uses more sophisticated word representation method (word embeddings, instead of plain *bag of words*). The training in [3] used a dataset of over 12400 commits (whereas we only used 2715). A comparison of their experimental results with ours is presented in Sec. IV. The classification of bug tracking tickets, also addressed in [3], is out of the scope of our work.

本文使用的数据库，Perl等人在commit中发现是否存在CVE字段，做了一个CVE的数据库

Vulnerability datasets like the one we used are a precious (and scarce) resource. Perl et al. [2], as part of their work on finding *Vulnerability Contributing Commits* (VCC), have produced what they claim to be the largest (public) database of mappings between CVEs affecting OSS and the commits that *introduced* the vulnerability. Their methodology, although automated, only considers CVEs that include an explicit reference to the corresponding commit(s); in our experience, these references are rather rare (and often inaccurate). The dataset of [2] includes 640 commits, corresponding to 718 CVEs, affecting 66 C/C++ projects.

Vendors such as Snyk (http://snyk.io/vuln) and SourceClear (http://sourceclear.com) provide vulnerability data through their Web pages; Snyk also publishes monthly dumps of their databases on GitHub. These resources could be used to replicate our study.

III. Commit Classification

Our approach to commit classification revolves around the following two key ideas.

之前工作只考虑了commit log，而我们将patch code也当成自然语言数据

First, we use conventional machine-learning algorithms and *natural language processing* (NLP) methods. Differently from other works (such as [3]) that concentrate on classifying commits based solely on the commit log message, we propose a novel way to treat the lines of code modified by a commit (in the following referred to as the *patch*) as a *text document* containing terms from the natural language. To do so, we extract the terms that the developer used to name entities in the source code (e.g., names of classes, functions, parameters, variables, constants etc.): the result is a document to which standard NLP methods can be applied.

Second, we are guided by the practical need (observed in our day-to-day experience with OSS vulnerability analysis at SAP [4]) to ensure *high precision*, to avoid wasting precious resources to deal with false-positives. To achieve this goal, we construct and train two different independent models, each considering a different aspect of the commit (the log message and the patch, respectively), tuning each of them for high-precision. We experimented with different models (logistic regression, random forests, support vector machines, among others), and with different combinations of parameter settings (e.g., different kernels and regularization methods, and other model-specific parameters). Due to space constraints, we omit here a discussion of the performance obtained with each combination. We observed that SVM performs consistently better than the other methods in our tests, and in particular, we found that a linear kernel and L2 regularization provided both good predictive performance and acceptable training time. We do not claim this is a general result, but we did observe that the impact of the choice of the model seems to have a lesser effect on predictive performance compared to the impact produced by different choices of feature sources and data pre-processing strategies.

我们将log和patch分开训练模型，并且尝试了逻辑回归，随机森林，支持向量机，发现SVM效果最好

When used separately, each of such models yields a relatively low recall. To compensate for this effect, we combine the two models in a *joint* model, implementing a simple *voting mechanism* that flags a commit as security-relevant as soon as *at least one of the two* models reports it as security-relevant. Because each model is trained on different facets of commits, their predictions are practically *independent*, and therefore the likelihood that the joint model produces a false negative is reduced, thus increasing recall.

我们利用投票机制，只要两个方面的模型有一个认为时安全性commit，则报出，这样可以获得更高准确而率

In the following we describe the two models (msg and patch) that are combined in the joint model.

A) Log Message Classifier (msg). Using the well-known *bag of words* approach, after some pre-filtering (stopword removal, normalization, removal of special characters, stemming) we transformed each log message into a vector of word counts and we trained a linear SVM model.

对于Log，使用词袋做特征提取，去除停用词，将其做成向量输入SVM

While this is a well-understood, basic document classification exercise, it comes with its own challenges. First, log messages can be rather short, and thus contain little information for the classifier to exploit. Second, by examining manually hundreds of security fixes, we observed that, depending on the community and on the specific project at hand, security fixes can be quite difficult to distinguish (even for a human expert) based only on the commit message.

但是，(1)log信息都很短，本身包含的信息较少（2）很多厂商会在log中回避安全相关字眼，以防止攻击者识别，展开1day攻击。

Ideally, one would expect that the log message should clearly indicate whether the commit at hand is meant to correct a security flaw. However, in practice, we observed that many projects avoid explicit log messages when pushing security fixes, possibly because that would make it easier for an attacker to infer what the vulnerability is, and to exploit it while a release embedding the fix is not ready yet.

B) Patch Classifier (patch). Along the lines of [6], this approach stems from the intuition that the choice of identifier names in source code (that is the terms that the developer uses to name classes, methods, variables) is an essential means to convey *semantics* and thus can convey information that can be used for classification.

Based on this intuition, we treat the source code changes themselves as a plain text document, and we reduce the problem of classifying code changes to the problem of classifying a document containing natural language terms. Once reduced to such a document, code changes can be represented as *bags of words* and be treated with standard text-classification methods (similar to the log message classifier, except for a different preprocessing phase (see next section).

IV. Evaluation

A. Dataset

For our studies we used a labeled dataset of known open-source vulnerabilities constructed while operating an open-source vulnerability scanner, described in [4], developed and adopted at SAP for internal use. Differently from the dataset of [2], which contains commits that *introduce* vulnerabilities, our dataset contains commits that implement *fixes* to vulnerabilities. Our data cover 220 open-source Java projects that are used in SAP software (either products or internal tools) and contains around 1100 security-relevant commits corresponding to the fixes for 540 vulnerabilities (e.g., CVEs). For our study we used a subset of 660 such commits from 152 repositories.

The data preparation phase includes several ad-hoc steps of little technical interest, but whose impact on the classification results is significant: we summarize here the main ones in the hope of encouraging studies that replicate (and possibly improve) our results. After skipping invalid commits, we removed duplicates (commits with different identifier but identical content), obtaining 456 unique commits, assigned to the *positive* (pos) class.

We then augmented the dataset with instances of the "negative" class (non-security-relevant): for each positive instance p from repository R, we took k random commits $n_1, \ldots, n_k$ from R and, under the assumption that security-relevant commits are rare compared to other types of commits, we treated these as "negative" examples. To check the validity of this assumption, and to avoid including obvious outliers (extremely large, empty, or otherwise invalid commits), we performed a manual review of these commits, with the help of ad-hoc scripts and pattern matching (similarly to [3]). We chose $k = 5$ to get approximately $(1 + 5) \times 456$ commits overall. After the manual review, we obtained a dataset of 2715 instances (456 of the positive class, and 2259 of the negative class).

This raw dataset was further processed to remove stop-words, filter non-alphanumeric characters and normalize the text data. In particular, for the patch data, we removed context lines, keeping the actual changed lines, and splitting composite tokens (e.g., camel-case identifiers).

B. Results

We used 10-fold cross-validation to evaluate the predictive performance of our models in terms of precision and recall, and we compared them to the best results in the state of the art. The outcome of our experiments is summarized in Fig. 1, for the two models *log message classifier* (msg) and *patch classifier* (patch) considered independently, as well as for their combination (joint). The x-axis represents the weight

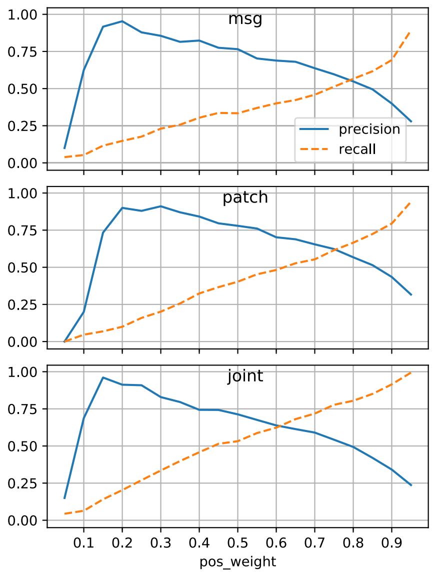

Fig. 1: Evaluation results

w_p assigned to the misclassification penalty for the positive class (the negative class gets $w_n = 1 - w_p$).

These plots illustrate clearly the trade-off between precision and recall. Depending on the *operating point* (controlled by adjusting the weights w_p, w_n), these models can be tuned to obtain the desired level of precision, at the expense of lower recall, or vice-versa.

The two left-most columns of the table below contain the different performance figures reported by [3], when comparing their approach with prior work or when describing the model they deployed in production. For comparison, in the three right-most columns we report the precision obtained with each of our models (msg, patch, joint) for each level of recall fixed in [3].

rec.	prec. [3]	prec. (msg)	prec. (patch)	prec. (joint)
0.50	0.51	0.58	0.71	0.74
0.72	0.34	0.42	0.51	0.57
0.76	0.34	0.39	0.45	0.56

As we explain in the next section, practical considerations stemming from industrial practice make us emphasize precision rather than recall. The table below shows the recall figures obtained when fixing precision at different levels equal to or higher than 0.80.

prec.	rec. (msg)	rec. (patch)	rec. (joint)
0.95	0.15	0.14	0.20
0.90	0.19	0.18	0.25
0.85	0.21	0.23	0.31
0.80	0.30	0.29	0.43

These figures show that, when tuning our models for high-precision, we still obtain useful levels of recall, which makes our approach suitable for real-world industrial use.

V. Commit Classification in Practice

We briefly discuss here two important aspects of practical commit classification: the choice of reference performance metrics, and the generalization of our approach to projects written in languages other than Java.

The Importance of Precision. In an industrial scenario, the security-relevant commits identified through automated classification need to go through the manual assessment of a human expert, who decides whether they should be added to the corporate database of vulnerabilities that is used by automated vulnerability scanners to check hundreds of projects across the company. In this context, any approach that would overwhelm the expert with too many false positives would be unsustainable; in our experience, precision figures below 0.80 are impractical, because the number of commits analyzed daily can quickly become unwieldy.

Cross-Language Application. While our approach is inherently (programming-)language independent, its predictive performance could change when our model is applied to projects written in programming languages other than the one covered by the training data.

In an initial attempt to study this aspect, we analyzed the 30 most recent vulnerabilities affecting *Ruby* projects published on snyk.io at the time of writing. For five of the 30 vulnerabilities, we could not identify a corresponding commit. On the other hand, some vulnerabilities are linked to multiple commits, and some commits contain the same changes. As a result, we obtained a dataset of 23 unique commits of the "positive" class, from 15 distinct repositories. Using the same technique as in Sec. IV-A, we derived a new validation dataset (Snyk-Ruby in the following) containing 224 additional "negative" samples (summing up to 247 commits).

We classified these commits with the joint predictor model to that we had previously trained on Java-only data, which resulted in a noticeable improvement of performance compared to the result obtained in the cross-validation using the SAP dataset. For comparison, on the Snyk-Ruby dataset we obtained a precision of 82% with a recall of 61%, whereas the best result we obtained on the SAP dataset (Java) was 80% precision and 43% recall. A similar result was reported in [3], with a increase in performance when using a model trained on Java to classify projects coded in Ruby or Go.

Based on our experience and some anecdotal evidence, we believe that these differences in performance might depend on the practices followed in different communities and even in different projects: the information included in log messages and the style of fixes (e.g., the amount and content of comments included with security fixes) could make it substantially easier (or harder) to classify them correctly.

While, of course, *no generality* can be claimed for the results of this simple exercise, they suggest that the effect of community-specific (and project-specific) practices on classification performance cannot be neglected. A systematic study of this problem is in our plans for future research.

VI. Conclusion

Motivated by the need to support open-source vulnerability management in an enterprise environment, we tackled the problem of identifying security-relevant commits in source code repositories using machine learning.

By translating the problem of classifying *code changes* to the problem of classifying a *document containing natural language terms*, we could apply classification models that require no sophisticated code analysis and are agnostic of the programming language. By combining relatively simple models optimized for high precision, we obtained classification capabilities that are suitable for industrial use. Our approach outperforms state-of-the-art approaches [3], while employing a computationally less expensive model and requiring a considerably smaller training dataset.

We do believe that additional improvements are possible, and that further research in model selection and optimization is necessary. We expect more *modern* approaches (e.g., word embeddings, recurrent neural networks) to improve performance even further and we plan to investigate their applicability. A key conclusion of this work is that, as a consequence of the *naturalness* of source code [5], even using basic NLP methods and without recurring to sophisticated code-analysis techniques, effective and relatively inexpensive commit classification can be achieved. A promising direction we are exploring at the time of writing is the classification of a given commit together with other resources (bug tracking tickets, advisories, mailing list discussions) that *link* to it [8]. We will report on our progress in this direction in a future paper.

Acknowledgements. We would like to acknowledge the insightful comments of the anonymous reviewers. We are also grateful to Ivan Pashchenko, Sule Kharaman, and Jamarber Bakalli for their help with the validation of our approach, and to Henrik Plate and Serena E. Ponta for their comments on early drafts of this work.

References

[1] Snyk.io, "The state of open-source security," 2017, Available: https://snyk.io/stateofossecurity/pdf/The%20State%20of%20Open%20Source.pdf

[2] 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 *Proc. of the 22Nd ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, New York, NY, USA: ACM, 2015, pp. 426–437.

[3] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, New York, NY, USA: ACM, 2017, pp. 914–919.

[4] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, Sept. 2018.

[5] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv preprint arXiv:1709.06182*, 2017.

[6] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. of the 34th Int. Conf. on Software Engineering (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[7] 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, Aug. 2017.

[8] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *SIGSOFT '10/FSE-18: Proc. of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering*. Association for Computing Machinery, Inc., Nov. 2010.