Detecting Security Fixes in Open-Source Repositories using Static Code Analyzers

Therese Fehrer∗
t.fehrer@uvt.nl
JADS, Tilburg University
's-Hertogenbosch, The Netherlands

Rocío Cabrera Lozoya
rocio.cabrera.lozoya@sap.com
SAP Security Research
France

Antonino Sabetta
antonino.sabetta@sap.com
SAP Security Research
France

Dario Di Nucci
ddinucci@unisa.it
University of Salerno
Italy

Damian A. Tamburri
da.tamburri@tue.nl
JADS, Eindhoven University of Technology
's-Hertogenbosch, The Netherlands

ABSTRACT
The sources of reliable, code-level information about vulnerabilities that affect open-source software (OSS) are scarce, which hinders a broad adoption of advanced tools that provide code-level detection and assessment of vulnerable OSS dependencies.

In this paper, we report our findings from using features extracted from four (PMD, Checkstyle, CK, Progex) off-the-shelf static code analyzers relying on pattern matching, software metrics or program analysis in a machine-learning pipeline to identify source code commits that contain vulnerability fixes. We show that successful machine learning models based on base classifiers and ensemble techniques can be trained on the combination of the features.

CCS CONCEPTS
• Software and its engineering → Maintaining software
• Computing methodologies → Supervised learning

KEYWORDS
source code representation, commit representation, machine learning, software vulnerability analysis

1 INTRODUCTION
The adoption of open-source software (OSS) components in commercial products has dramatically increased over the past two decades. Snyk [14] reports that the large majority of the applications they analyzed contained at least one open-source component; as much as 50% to the entire code-base of those applications is open-source. By building upon these free, community-developed building blocks, vendors can focus their efforts on differentiating features to bring them to the market faster. However, they become responsible for assessing and mitigating the impact that a vulnerability in those open-source components might have on their products.

Unfortunately, this is not an easy task. The data about security vulnerabilities is scattered across heterogeneous sources, often not machine readable, and do not provide the necessary level of detail, especially when it comes to code-level details. This difficulty in obtaining accurate (code-level) vulnerability data hinders further development of new tools that could push the state of the art in vulnerability detection and mitigation. Also, it makes it harder for the research community to learn from real-world vulnerability and their fixes. In this perspective, the role of automated tools to find vulnerability fixes in code repositories, possibly using machine learning, becomes increasingly important [2, 13].

In this paper we present an approach to analyze repositories considering the changes introduced by individual commits. We investigate the use of off-the-shelf tools for code quality analysis as sources of features to represent code changes captured in commits. Based on representations build from these features, we train models to predict whether a commit is likely to fix a vulnerability or not.

This research originated from an industrial context through cooperation with SAP. In this context, vulnerabilities are implicitly

∗Therese Fehrer contributed to this research during her internship at SAP Security Research where she worked on her MSc thesis.
defined through their associated fixes. Commits that fix vulnerabilities are categorised as security-relevant commits.

Static analyzers read a program and construct its abstract representation without executing it [9] to analyze its quality. They have been combined with machine learning techniques for a range of other software engineering-related prediction tasks, such as software fault prediction [8, 11], defect prediction [1, 12, 15, 17], code smell prediction [10] and for the prediction and prioritization of technical debt [3]. We identified four static analyzers as feature extractors for the commits in our dataset, two (PMD\(^1\) and Checkstyle\(^2\)) rely on pattern matching, one (CK\(^3\)) relies on software metrics and another (Progex\(^4\)) relies on program analysis.

Our contributions are relevant from both an academic and an industrial perspective. In particular, we provide:

- new insights concerning the usage of off-the-shelf static code analyzer as sources of features that can be used to predict security-relevant commits;
- an extensible pipeline to pre-process and combine the outputs of various static code analyzers to train machine learning models on commits;

The remainder of this paper is structured as follows. Section 2 provides the methodology used in this study, including data preparation, feature extraction and machine learning pipeline. The results are shown in Section 3. The threats to the validity of the empirical study are reported in Section 4, while Section 5 discusses the related literature. Finally, Section 6 concludes the paper.

## 2 METHODS

This study aims to understand to what extent machine learning classifiers can leverage static code analyzers to identify security-relevant commits. These tools are widely used in the software development process and can help create numerical representations of source code, which can in turn be fed to ML algorithms.

For this purpose, we used a dataset of security-relevant commits [2] which captures a representative sample of open-source projects of practical industrial relevance.

We applied several data pre-processing steps to the dataset [2], ensuring that only reachable commits were included and added information concerning the vulnerability types. Furthermore, repositories included in the data set only through unreachable commits were dropped entirely. We aimed to keep the percentage of positive and negative instances as close as possible to the original data set. In the end, the final data set contained 1,821 commits.

### 2.1 Feature extraction

The feature extraction process for all commits is shown in Figure 1. We analyzed the files contained in the commits of our dataset and their predecessors using the static code analyzers previously introduced. We computed one feature vector per tool and per commit. The three main phases of the pipeline are

- **Relevant file extraction**: identifies the files changed by the commit
- **File feature extraction**: Code features are extracted for the previous and current versions of each relevant file. PMD and Checkstyle return a list of bugs and lines where they occur. We compute the occurrence of each bug per file. For CK we use the class.csv file capturing the software metrics at the file level. Finally, we use Progex to extract source code abstract syntax trees and control flow graphs and measure their characteristics using Networkx\(^5\).  
- **File-to-commit representations**: For each of the extracted features, we add an additional two flags indicating a positive or negative feature changes at the file-level embedding: the first to aggregate positive file-level changes and the second to aggregate negative file-level changes. This step allows us to avoid information loss: a simple sum over all the feature values could have balanced positive and negative values.

### 2.2 Machine Learning Pipeline

The feature vectors resulting from static code analyzers were then used to train machine learning models to classify security-relevant commits. Figure 2 depicts an overview of the adopted machine learning pipeline. First, we train models separately for each feature vector resulting from each of the four code analysis tools. We applied feature scaling, performed feature selection by removing

---

1. https://pmd.github.io
2. https://checkstyle.org
3. https://github.com/mauricioaniche/ck
4. https://github.com/ghaffarian/progex
5. https://networkx.org/
This section describes the threats to validity of our study. First, Table 1 presents the performance metrics for the best performing vulnerabilities and commits that are likely to fix such vulnerabilities.

We tried seven different machine learning algorithms belonging to different classification families (Decision Trees, Random Forests, AdaBoost, Gradient Boosting, Support Vector Machines, Logistic Regression, and Gaussian Naïve Bayes) and tuned the hyperparameters through 200 iterations of a randomized search and optimizing for precision.

We also explored the use of ensemble techniques to combine different classifiers to make a final prediction. We select the best performing models and use voting and stacking techniques to combine them. Voting [7] averages the predicted probabilities of the base estimators to make a final prediction and stacking [16] stacks the predictions of the individual estimators together and uses them as input to a final estimator to compute the prediction. The final estimator is trained through cross-validation.

Performance was measured using precision, recall, accuracy and F1-scores averaged after a 5-fold stratified cross-validation strategy.

### 3 RESULTS

Table 1 presents the performance metrics for the best performing model trained on each of the tools’ features. The CK model performs the best regarding all four evaluation metrics. Similarly, the Checkstyle model reaches the worst performance for all four metrics. The PMD model performs better than the Progex model regarding precision and accuracy but performs worse in terms of recall and F1 score.

When combining these four best-performing models using stacking and voting, the voting classifier performs better in all four metrics and achieves an average precision of 77.5% and an average recall of 48.64%. Combining the models leads to significantly better results than using them independently, indicating that the models are complementary to each other.

### 4 THREATS TO VALIDITY

This section describes the threats to validity of our study. First, this research is inherently based on the definition of software vulnerabilities and commits that are likely to fix such vulnerabilities.

| Embedding | Precision | Recall | F1-Score | Accuracy | Model         |
|-----------|-----------|--------|----------|----------|---------------|
| CK        | 75.62 ± 5.60 | 40.07 ± 1.28 | 52.34 ± 2.11 | 63.05 ± 2.50 | Random Forest |
| Checkstyle| 64.89 ± 3.69 | 31.25 ± 3.36 | 42.05 ± 2.78 | 56.56 ± 1.44 | Gradient Boosting |
| PMD       | 75.44 ± 5.37 | 35.98 ± 4.05 | 48.05 ± 3.30 | 61.35 ± 1.58 | Random Forest |
| Progex    | 69.27 ± 4.01 | 30.25 ± 4.14 | 49.23 ± 4.22 | 60.18 ± 3.15 | Gradient Boosting |
| Stacking  | 74.79 ± 4.87 | 49.89 ± 5.02 | 56.56 ± 2.94 | 64.52 ± 1.68 | -              |
| Voting    | 77.51 ± 5.28 | 48.64 ± 1.71 | 59.68 ± 1.35 | 66.73 ± 2.13 | -              |

The classification of commits in our dataset into security-relevant and non-security-relevant commits is not based on a formal definition; vulnerabilities are only implicitly defined through their fix-commits.

Additionally, the results of our study are tied to the specific static code analyzers considered. Other static code analyzers or other preprocessing approaches may significantly change the results. We mitigated this risk through researching tools and techniques that have been applied in related software engineering prediction tasks.

Finally, the dataset we used contains repositories, which are practical relevant to SAP, the company which provided the dataset. Within these repositories, there might be a bias towards easily reachable commits. Since part of the dataset was manually curated, certain types of commits might have been harder to reach and not represented appropriately in the dataset. Furthermore, our data distribution might not reflect the actual distribution of vulnerability types.

### 5 RELATED WORK

The survey in [6] focuses on the application of machine learning and data mining approaches to analyze software vulnerabilities. Common approaches include vulnerability prediction models based on software metrics, anomaly detection approaches, and vulnerable code pattern recognition. Our study deviates from this work particularly in the unit of analysis. We are not interested in identifying if a software component is vulnerable but instead if a commit is likely to fix a vulnerability. Furthermore, the focus of most existing approaches is a method or file-level analysis. We instead use whole commit as our unit for analysis. In this respect, the closest work to ours is commit2vec [2]. It represents whole commits to identify security-relevant commits relying on abstract syntax trees and neural networks.
Static code analyzers have been used to create different feature sets for prediction tasks in software engineering: software metrics, graph-based metrics, and pattern-matching-based metrics. The work in [10] used the output of bug finders to predict code smells, namely poor implementation choices applied during software evolution that can affect source code maintainability [5]. As more attention has been dedicated to machine learning solutions for identifying code smells [4], the authors examine the role of static analysis warnings as features of machine learning models for detecting code smell types. Specifically, CHECKSTYLE, FINDBUGS, and PMD are used to detect three types of code smells.

Software metrics have also been used as a feature set for a variety of other prediction tasks. Most notably, they have been used in software fault [8, 11] and technical debt prediction [3]. Finally, some papers use dependency graphs to identify program units that are more likely to face defects [1, 12, 15, 17].

6 CONCLUSION

In recent years, the need for automated identification of security-relevant commits emerged to ensure software security. We address this need by proposing a framework to use static code analyzers combined with machine learning techniques to detect security-relevant commits. We extracted features from four off-the-shelf code analyzers (PMD, CHECKSTYLE, CK, and PROGEX).

We created machine learning models based on these features to predict the security relevance of commits. We show that successful machine learning models based on base classifiers and ensemble techniques can be trained on the combination of the features.

ACKNOWLEDGMENTS

Antonino Sabetta’s and Rocío Cabrera Lozoya’s contributions to this work are partially supported by the European Commission grant no. 101120393 (Sec4AI4Sec).

REFERENCES

[1] Pamela Bhattacharya, Marios Ilkofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In 2012 34th International Conference on Software Engineering (ICSE), pages 419–429. IEEE, 2012.
[2] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. SN Computer Science, 2(3):150, Mar 2021.
[3] Zadia Codabux and Byron J Williams. Technical debt prioritization using predictive analytics. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pages 704–706. IEEE, 2016.
[4] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner), pages 612–621. IEEE, 2018.
[5] Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
[6] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv., 50(4), August 2017.
[7] Josef Kittler, Mohamad Hatef, Robert PW Duin, and Jiri Matas. On combining classifiers. IEEE transactions on pattern analysis and machine intelligence, 20(3):226–239, 1998.
[8] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. Progress on approaches to software defect prediction. IET Software, 12(3):161–175, 2018.
[9] Panagiotis Louridas. Static code analysis. IEE Software, 23(4):58–61, 2006.
[10] Savanna Lujan, Fabiano Fecorelli, Fabio Palomba, Andrea De Lucia, and Valentina Lenarduzzi. A preliminary study on the adequacy of static analysis warnings with respect to code smell prediction. In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, pages 1–6, New York, NY, USA, 2020. Association for Computing Machinery.
[11] Ruchika Mallhotra. A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing, 27:504–518, 2015.
[12] Rahul Premraj and Kim Herzig. Network versus code metrics to predict defects: A replication study. In 2011 International Symposium on Empirical Software Engineering and Measurement, pages 215–224. IEEE, 2011.
[13] Antonino Sabetta and Michele Bezzi. A practical approach to the automatic classification of security-relevant commits. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSM), pages 579–582. IEEE, 2018.
[14] Liran Tal. The state of open source security report, February 2019.
[15] Ayşe Tosun, Burak Turhan, and Ayşe Bener. Validation of network measures as indicators of defective modules in software systems. In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE ’09, New York, NY, USA, 2009. Association for Computing Machinery.
[16] David H Wolpert. Stacked generalization. Neural networks, 5(2):241–259, 1992.
[17] Thomas Zimmermann and Nacihiapam Nagappan. Predicting defects using network analysis on dependency graphs. In Proceedings of the 30th International Conference on Software Engineering, ICSE ’08, pages 531–540, New York, NY, USA, 2008. Association for Computing Machinery.