Discovering Vulnerabilities and Patches for Open Source Security

Tamara Gunkel and Thomas Hupperich
Department of Information Systems, University of Münster, Germany

Keywords: Web Security, Data Set Generation, Commit Classification.

Abstract: Open source software is used in numerous systems and security vulnerabilities in such software often affect many targets at once. Hence, it is crucial to find security vulnerabilities as soon as possible. A convenient method to check software for vulnerabilities is executing a static code analysis tool before deployment. However, for verifying the reliability of such tools, real-world data including labeled non-vulnerable and vulnerable code is required. This paper introduces an approach to automatically create and enhance a labeled data set of open source projects. The ground truth of vulnerabilities is extracted from up-to-date CVEs. We identify repositories related to known vulnerabilities, select vulnerable versions and take patch commits into account. In this context, we utilize Gradient Boosting based on regression trees as a meta classifier for associating patch commits to CWE categories. With a high precision of this matching, we give insights about the impact of certain vulnerabilities and a general overview of open source code security. Our findings may be used for future studies, such as the impact of certain code design criteria, e.g. clean code, on the prevalence of vulnerabilities.

1 INTRODUCTION

Open source software is popular and widely used in many contexts. Therefore, it is crucial to identify vulnerabilities in open source code as soon as possible. Using automatic tools such as static code analysis tools in continuous integration pipelines can prevent vulnerable code from being deployed. However, testing static code analysis tools and applying machine learning is challenging as only a few data sets with real-world vulnerabilities and source code are available. As this need was already identified, previous work contributed by creating manually maintained data sets which were often limited to specific projects or programming languages. Thus, to foster code analysis of open source software projects and ultimately contribute to securing them, we seek a solution to automatically create data sets of vulnerable and patched software based on the ground truth of Common Vulnerabilities and Exposures (CVE).

Not only is identifying vulnerabilities in software a key to preventing security incidents but also keeping track of when a vulnerability is mitigated. This allows security experts an assessment of impact and provides information about the software being secure to use in the future. Therefore, we determine for vulnerabilities classified by our approach whether or not these have been fixed.

Our contribution to open source software security is:

- We compose a database to represent current vulnerabilities as a ground truth. As vulnerability data feed, the popular National Vulnerability Database (NVD) is used that lists vulnerabilities following the CVE specification (of Standards and Technology, 2021; Corporation, 2020b).
- We search open source repositories for known vulnerabilities and patches. In total, we inspect 545 repositories of mainly web-related projects written in JavaScript, Java, and PHP.
- We create a classifier to determine the vulnerability types fixed by specific commits. In this process, we examine commit messages, file extensions, and names to be most relevant for vulnerability classification.
- We build a data set containing commits fixing vulnerabilities described by CVEs. The data set contains 1000 CVEs and will be published with this paper.

This work allows developers to create their own data sets of vulnerable and patched code, which can be used to conduct further research. By using such data sets, static code analysis tools may be validated regularly and analysts may react in a timely manner to new vulnerabilities that are not yet detected.

Gunkel, T. and Hupperich, T.
Discovering Vulnerabilities and Patches for Open Source Security.
DOI: 10.5220/0011299400003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 641-648
ISBN: 978-989-758-588-3; ISSN: 2184-2833

2 RELATED WORK

The need for a data set consisting of vulnerabilities and patch commits was already identified by multiple researchers. Ponta et al. created a data set consisting of 624 vulnerabilities in 205 Java projects and 1282 commits fixing them (Ponta et al., 2019). Another data set with 153 projects was created by Gkortzis et al. (Gkortzis et al., 2018). While promising in their approaches, both projects are not maintained anymore and lack actuality. The most recent work, published in December 2020, is the OpenSSF CVE Benchmark project. It contains around 200 JavaScript and TypeScript CVEs (working group, 2021). However, our approach is not limited to specific programming languages and can identify multiple patch commits instead of only the last one.

Furthermore, research was conducted on identifying security-relevant commits. Sabetta and Bezzi as well as Sawadogo et al. trained machine learning classifiers for commit messages and the changed code lines to flag security-relevant commits (Sabetta and Bezzi, 2018; Sawadogo et al., 2020). As part of their methodology, regular expressions are used, e.g., to filter out obvious non-security patches (Sawadogo et al., 2020). Another approach by Barish et al. uses only the source code as input vector (Greg Barish and Minton, 2017). In our research, we combine the approaches and use source code, commit messages, and file types to classify commits into vulnerability types.

The research closest to this paper is the VulData7 framework from Jimenez et al. (Jimenez et al., 2018). For a given project, VulData7 can retrieve related CVEs and commits. Although it is partly automated, the approach is still limited in terms of usability. The user has to define regular expressions to match CVEs to a project. This requires extensive knowledge of the project itself and CVEs. Moreover, unknown projects cannot be included in the data set. Our approach overcomes these limitations and provides a tool that is independent of specific software projects and can be used without further configuration.

3 APPROACH

Chasing vulnerabilities to measure open source projects' security needs to be based on reliable data about relevant vulnerabilities. Therefore, we utilize CVEs, officially released by The MITRE Corporation (Corporation, 2020b). Two limitations apply. First, we focus on GitHub because most repositories are hosted there. Second, we do not process CVEs with multiple Common Platform Enumeration (CPE) configurations referring to different products as the identification of vulnerabilities cross-repository may be ambiguous. In the following, we describe the process of matching CVEs to repositories and their corresponding patch commits.

3.1 Locating Source Code Repositories

After retrieving a CVE, the affected repository has to be found by utilizing five different heuristics that include the following criteria:

The **Similarity Criterion** is used to check if a repository is similarly named as specified in the CPE. The normalized Levenshtein similarity is calculated between (1) repository and CPE product name, and (2) repository owner and CPE vendor name. The average of both similarities must exceed 0.7 to consider the repository as a potential match. The Levenshtein measure is well suited as we allow only minor differences between the names. The threshold 0.7 was chosen because it worked well during manual testing.

The **Language Criterion** identifies the programming language(s) of repositories belonging to a CVE by utilizing the target software configuration from the CPE as well as the file endings of files and programming languages mentioned in the CVE description. If more than one programming language was identified, the potential repository must include at least one of these languages.

The heuristics for finding potential repositories are based on two information sources: CVE/CPE references and GitHub Search API. GitHub repository references are considered as a potential match if

1. **CVE References** are tagged as *Patch* or *Issue Tracking*.
2. **CPE References** are tagged as as *Product*, *Version*, *Project*, or *Change Log*.
3. **Remaining CVE or CPE References** fulfill the similarity criterion. We include references from CVEs sharing the CPE as other CVEs of the same product might be more descriptive.

Additionally, up to five repositories from the following GitHub search queries are considered as potential matches.

4. **CPE Product Name** as keyword.
5. **CPE Product Name** as keyword and **CPE Vendor** as user. The results of this query and the previous one must fulfill the similarity criterion.
6. **CPE Product Name** as keyword and **Programming Language** specified. The product name must be mentioned in the repository description, or the similarity between the CPE product and

repository name must exceed 0.7. The similarity criterion was weakened because repositories with high name similarities were already identified by previous heuristics.

The last heuristic uses the **GitHub links included in referenced websites**. Repositories must include the product name in the description or readme and fulfill the language criterion. Additionally, only the repository with the highest name similarity is considered as a potential match. If no repository is found because e.g. it is not open source, the CVE cannot be further processed. After collecting potential repositories, they are rated by summing up the scores of the applicable heuristics. The scores are derived during the evaluation in section 4.1.

3.2 Probing for Vulnerable and Fixed Versions

Next, the first and last vulnerable and first fixed versions are identified to isolate the relevant range of commits. The lowest and highest affected versions are extracted from the CVE applicability statements. These versions have to be found in the repository where versions are represented by Git tags. The lowest tag between the lower and upper boundary versions is considered as the first vulnerable version and the highest tag as the last vulnerable version. For the comparison, we extract the version from the tag name.

Furthermore, any version of the repository higher than the last vulnerable version is assumed to be fixed because a patch must have been implemented if the version is not affected anymore. The first fixed version is defined as the lowest release version among all repository versions, which are higher than the last vulnerable version.

3.3 Identifying Patch Commits

After identifying the repository and vulnerable versions, we seek to find patch commits to gain information about possible mitigations. A vulnerability may be fixed with one or multiple commits.

The first heuristic instruments the CVE references. If references tagged as *Patch* are commit or pull request links, they are considered as the patch commits. For pull requests, the commits are extracted. If the pull request not merged, the CVE receives the status `No fix` because the fix is not included in the repository yet. The second heuristic checks if any *Issue Tracking* reference links to a GitHub issue. Open issues are considered as not fixed. If it is closed, associated commits are marked as the patches. Then, commit hashes are extracted from the CVE description. If these heuristics do not yield useful information, the individual commits are investigated. The identified boundary vulnerable versions are used to limit the commit range. The obvious way to identify patch commits is to search for the CVE Identifier (CVE ID) in the commit message. If one or multiple commits contain the CVE ID, they are considered as patches. If not, a commit scoring system is used to rate the commits. We utilize the following heuristics:

(1) The tool *git-vuln-finder* is used to determine if a commit is security-related by utilizing regular expressions (Dulaunoy, 2021).

(2) The similarity of the CVE description and commit message is calculated. We tokenize both inputs and calculate the the Inverse Document Frequency (IDF) score of every token to respect the different importance of tokens. As vocabulary, all commit messages and the description are used. For an individual commit, the similarity is then calculated by summing up the IDF values of the commit message and description tokens. The similarity is normalized by the sum of the IDF values of all description tokens to allow the comparison of the scores across CVEs.

(3) We check if a URL contained in the commit message appears in the CVE references.

(4) If file names were extracted from the description, all commits modifying the files receive points.

(5) The Common Weakness Enumeration (CWE) classifier categorizes commits into CWE-79 (XSS), CWE-89 (SQLi), CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), and CWE-20 (Path Traversal). These CWEs were selected because they are part of the top 25 CWEs and performed best during the experiments (Corporation, 2020a). The training data for the classifier was collected automatically. For each CWE, all CVEs with a *Patch* reference to a GitHub commit or pull requests were retrieved. From the commits, the following five features are engineered and used as input vectors.

a) Tokens of the **commit message**

b) **File extensions** of the modified files.

c) **Class names** occurring in all modified files.

d) **Method names** occurring in all modified files.

e) **Names** (variables, parameters, methods called) occurring in the modified lines only as they are especially relevant for the vulnerability. Names consisting of multiple words e.g. using camel-case notation are split into single words.

The last three features utilize the modified source code of a commit and might be well suited to distinguish between CWE categories. For example, names

such as `database` are more likely to occur in commits regarding SQL Injection than buffer overflows. All features are transformed into a numerical representation by applying a count vectorizer, meaning that the frequency of the tokens is used. After that, TF-IDF transformation is applied so that the importance of each token is reflected. The classifier is only used as scoring criteria for commits if the CVE is categorized as one of the supported CWEs.

4 EVALUATION

For evaluating how well the introduced approach works, we created a data set with commits that fix a vulnerability described by a CVE. The data set contains 1000 CVEs. The data set was created by manual selection of the CVEs, crawling the Snyk vulnerability database (Ltd., 2021) and extracting CVEs from the Vulas data set (Ponta et al., 2019)and the OpenSSF CVE Benchmark (working group, 2021). Most CVEs are related to injection and resource handling vulnerabilities because the majority of projects are web-related. Another negative data set with 142 CVEs, where the repository is not publicly available or not hosted on GitHub, was created to evaluate how many false positives are found.

4.1 Finding the Source Code Repository

As different heuristics (also called origins) are used, a scoring system was developed to select the best match. First, we grouped the CVEs by the target repository and for the CVEs without a repository by the CPE vendor and product. Some heuristics such as the GitHub search always return the same results for shared CPE. Thus, the success rate rather depends on the repository than on the CVE. The groups are called *repository groups*. There are 572 repository groups for the normal and 73 groups for the negative data set.

The scoring system is dynamically calculated and evaluated by using a 10-fold cross validation. First, it is determined how many points an origin should receive. For each CVE, every origin referring to the correct repository receives one point. The points are added up within a repository group. After that, the relative success rate of each origin is calculated by dividing the points by the total number of found repositories within the repository group. We average the success rate across all repository groups. The final score for an origin is determined by rounding the success rate to either the next 10% or 5%. We map the score to numbers between 0 to 10 and 0 to 20, respectively.

After calculating the origin scores, the best repository needs to be selected. Two different approaches were evaluated. The *hard threshold* approach accepts the highest-rated repository if it has a score above or equal to a certain threshold. The *soft threshold* approach also accepts a repository if only the language criterion and the version criterion are fulfilled. The version criterion requires that at least one of the affected versions must appear as a tag in the repository. Thus, it is possible that the top-ranked repository is not selected if it has a score below the threshold and does not fulfill the additional checks.

In total, four approach combinations were evaluated. The CVEs were labeled with the result of the scoring system, allowing the following combinations. For CVEs from the negative data set, either no repository (NN) or an incorrect repository (NI) can be found. For CVEs from the general data set, no repository (RN), an incorrect (RI), or the correct repository (RC) might be selected.

The metrics used to evaluate the performance of the approaches should be suitable for the following two goals: **G1:** Select as few wrong repositories as possible, **G2:** select as many correct repositories as possible. The first goal is considered more important because wrong repositories reduce the quality of the final data set. However, not detecting some repositories does not affect the data quality but only the quantity. We select the three following metrics for evaluation:

1. **Precision** is used to evaluate goal G1.
$$\frac{RC}{RC+RI+NI} \tag{1}$$
2. **Recall** reflects goal G2.
$$\frac{RC}{RC+RI+RN} \tag{2}$$
3. The **F0.5-score** is used to combine the previous metrics into a single evaluation score. We use the F0.5-score to put more emphasis on precision than on recall to reflect the different importance of the two goals (Chinchor, 1992).

The evaluation results in Table 1 demonstrate no significant difference between the 10% and 5% rounding. We select the 10% threshold because the score

Table 1: Scoring Performance Metrics.

	10 % Hard	10 % Soft	5 % Hard	5 % Soft
Precision	.9405	.9608	.9497	.9608
Recall	.8974	.8942	.8645	.8942
F0.5-Score	.9314	.9465	.9310	.9465

All numbers are averages on test fold

range is smaller and easier to interpret. Furthermore, the soft threshold performs better showing that the additional checks are sufficient to select the correct repository.

To create the final scoring system, the approach is applied to the entire data set. The threshold of 19 has the best F0.5-score of 0.9493 with a precision of 96.43% and a recall of 89.37%. The following origin scores show that a repository must be supported by at least two origins to be directly selected as the match without the additional checks.

The origins *Patch* and *Issue Tracking* perform best and receive 10 points. This supports the assumption that these references are reliable because they are individually added to each CVE. The *CPE References* also have a high success rate with 9 points because they are specific to the project but not to the CVE. If they fail, the reason is generally the imprecise labeling of the CVEs. The origin *Third Party Websites* receives 9 points as well demonstrating the relevance of CVE references. The scores of the GitHub search (*Owner and Repository Search*, and *Language Search*) are comparably low with 6 and 3 points because weaker similarity criteria are applied. This is intended because these origins rather have the purpose of supporting other origins than to be used as individual indicators.

4.2 Finding the Vulnerable and Fixed Versions

To evaluate if the version/tag matching works, we collected statistics about the characteristics. In contrast to the repository search evaluation, this and the following sections are based on CVE and not repository level because CVEs of the same repository group have different versions and patches.

Of 1,000 CVEs in total, only 26 do not define vulnerable versions. For the other CVEs, manual inspection shows that the tags are correctly identified in most cases. Multiple reasons were identified why the version matching might fail: no maintaining of tags and releases, insufficient support for parsing some version formats and missing tags due to maintaining releases on external platforms. This shows that well maintained repositories following common coding and naming guidelines allow a more precise security assessment of possible vulnerabilities.

4.3 Finding the Patch Commits

An essential part of our research is evaluating how well the heuristics can identify patch commits and creating a scoring system for selecting patch commits. As this step is built upon the version matching, it is assumed that the correct repository was identified to evaluate the performance independently of the repository search.

During the evaluation, four categories are used to group the commit results. If the found commits exactly correspond to the true commits, the CVE is flagged as `success`. If the found commits are a subset of the true commits, the category `not_complete` is used, and if it is the other way around, the tag `found_more` is used. The last category `failure` is used if wrong commits were found and not all of the true commits are included. On the commit level, commits are evaluated like binary classification. *Positive* commits are patch commits and *negative* commits are non-patch commits.

First, the three approaches extraction of commits from the *Patch* references, the *Issue Tracking* references and the CVE description are evaluated because they execute very fast in contrast to the scoring system and are more precise. The approaches demonstrate the following success rates: 97.46% for *Patch* references, 92.68% for *Issue Tracking* references and 91.67% for the *CVE Description* heuristic. To calculate the success rate not only the `success` CVEs were considered as successful but also the `found_more` CVEs. The latter CVEs were manually inspected, and the additional commits are either commits with the same content but on different branches or contain patch related information such as changelog updates. These CVEs are considered as successful because the tool does not intend to narrow down the fix to specific code lines.

If none of these approaches finds a patch commit, we utilize a high-effort method by further investigating the commits within the valid version range. This process required between one minute and eight hours for one CVE depending on the number of commits as the commits' messages are parsed through the Natural Language Processing (NLP) pipelines. Though this seems to be a high time effort, this part of the procedure needs to be run only once on creation of the data set. However, CVEs from ten repositories were excluded from the following evaluation because the process took over eight hours on a standard computer.

When the individual commits are inspected, CVE IDs are extracted from the commit messages. This method could be applied to 38 CVEs and was successful for 94.74%. One CVE failed because someone suggested including the CVE ID after the patch was already partially implemented. For another CVE only the changelog commit was found because the inspected version range which was incompletely listed in the CVE specification.

Overall, 597 CVEs remain where the scoring system must be applied. For 179 CVEs, the true patch commits were not included in the retrieved commits. The main reason was that the version range was wrongly specified in the CVE. These CVEs are skipped because the preceding steps are obligatory to evaluate the scoring system. The evaluation of the scoring system is conducted on commit level. In total, 195,507 commits are included and out of these, 549 patches must be identified showing that finding patch commits is a highly imbalanced problem.

For the scoring system, appropriate scores for the five heuristics must be determined. Except for the *CWE Classifier* heuristic, each heuristic score is represented by its F-measure value because it reflects the heuristic's precision and reliability. By creating three different scoring systems using the F0.25-, F0.5-, or F1-Score, the users can decide how much they value precision and recall.

For the *Description Fit* heuristic, a commit receives the points if the fit is above or equal to a certain threshold. We evaluated different absolute and relative thresholds and found that the absolute threshold 0.2 with a precision of 86.29% and recall of 12.54% should be used if precision is highly important. Otherwise, both the F0.5- and F1-score are best at threshold 0.1 with a precision of 58.41% and recall of 32.35%. This shows that the heuristic works well in a few cases but is limited in its applicability because commit messages are short and often use abbreviations for terms occurring in the CVE description.

For the heuristics *git-vuln-finder*, *URLs*, and *Files*, no thresholds exist. The heuristics either succeed for a commit or not. Thus precision, recall, and F-measure are calculated from the results of the entire data set. The performance values are listed in Table 2.

Table 2: Performance of the Heuristics URLs, Files and git-vuln-finder.

	Num. CVEs	Precision in %	Recall in %
URLs	7	100.00	100.00
Files	50	49.68	93.75
git-vuln-finder	295	22.70	41.67

The last heuristic is the *CWE classifier* which was evaluated in further experiments. Three common classification algorithms suitable for labeled data with a small data set were evaluated using 10-fold cross validation. The Gradient Boosting classifier outperformed the Random Forest and the Support Vector Machine (SVM). The average precision of the cross validation is 0.85 and the recall is 0.73. Furthermore, undersampling techniques applied to the majority class were evaluated because the class CWE-79 included approx. 700 CVEs while the other classes included between 150 to 300 CVEs. The Tomek Links method performed best and increased the precision by 2% and the recall by 1%. Thus, the final classifier uses Gradient Boosting with Tomek Links undersampling for CWE-79.

Since the CWE classifier only categorizes vulnerability types, it is used for discarding commits unrelated to the vulnerability type. As a metric, only performance values from the CWE classifier evaluation can be used because there are too few CVEs in the test data set with corresponding CWEs to reevaluate the classifier. We use recall as score because it reflects how reliable the classifier discards negative commits. The following recall values apply: 0.97 (CWE-79), 0.90 (CWE-119), 0.61 (CWE-22), 0.46 (CWE-89). The CVE classifier can be applied to 22% of CVEs included in the commit scoring system evaluation.

After creating the scores for all heuristics, the final threshold, below which commits are discarded, needs to be determined by applying the heuristics to all commits. For each CVE, precision and recall are calculated so that all CVEs are equally weighted. Finally, the F-measures are calculated to select the best thresholds. To evaluate how well this approach performs on unseen data, 10-fold cross validation is conducted. The heuristic scores and the thresholds are selected based on the training set and the performance is evaluated on the test set. The F.025 strategy shows a precision of 80.17% and a recall of 22.70% whereas the F1 strategy has a precision of 47.55% and 47.41%. The metrics of F0.5 lie between these values. Considering that the data set is highly imbalanced with 549 patches and 195,507 non-patches, a precision of 80% is quite good. Moreover, it is reasonable to offer different precision levels as there are remarkable differences in the metrics.

The approach was applied to the entire data set to create the final scoring system. Figure 1 shows the precision-recall curves of the three precision levels. For the highest precision level with the F0.25-score as well as the F0.5-score precision level, the final threshold is 0.3 and for the F1-score 0.2. The plot illustrates that a threshold above zero results in a recall decrease of 40%, meaning that there are many patch commits where no heuristic succeeds.

In summary, the results show that it is not trivial to identify the patch commits. The biggest challenge is that patches are not always described appropriately in the commit message and might even be included in a bigger merge commit. Additionally, it is hard to identify the correct patch if multiple vulnerabilities of the same type exist in the repository.

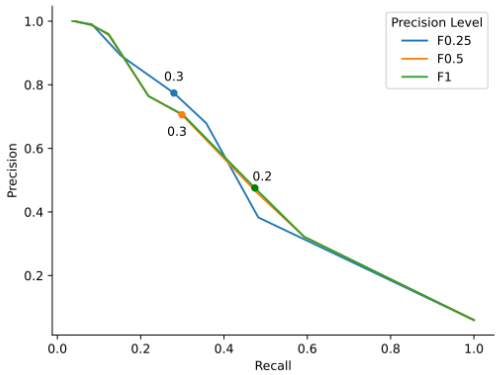

Figure 1: Precision-recall Curve of the Proposed Commit Scoring System.

4.4 Total Performance

After evaluating the individual steps, this section gives an overview of the entire approach's performance. Figure 2 shows the previously presented metrics and the overall performance. Step 2 presents the aggregated performance of the *Patch*, *Issue Tracking*, *CVE Description* and *CVE ID* heuristics. The blue numbers presents the number of CVEs that are processed or omitted in the paths. As for step 4 and the overall performance, the metrics of the F0.25 precision level are illustrated because the scoring system outperforms the other levels in the relevant area around the threshold of 0.3 (see Figure 1). Overall, a precision of 74.65% and recall of 48.42% is achieved.

Since there are not any similar studies, these metrics cannot be directly compared to previous work. The performance of the simple heuristics can be classified as really good, while the scoring system may still be improved. However, the evaluation shows that the approach is independent of any programming language or specific repositories and does not require any user interaction. A shortcoming of the prototype is its slowness if the version range cannot be limited and many commits need to be investigated. As a mitigation, the processing of CVEs could be parallelized.
In summary, the following four key takeaways can be derived from the conducted evaluation:

1. Matching from a CVE to a repository is simple because the CVE references and applicability statements usually describe the repository well.
2. The CVE description, references, and CVE IDs mentioned in commits are a reliable source for identifying patch commits.
3. The NVD information regarding the vulnerable versions does sometimes not match vulnerable tags in repositories.
4. Finding the patch commits using the scoring system is challenging as documentation of the vulnerabilities in issues or explicit commit descriptions is often missing.

5 DISCUSSION

Despite the high performance of our approach, there are underlying assumptions and certain limitations. These lead to possible enhancements for future work to strengthen the overall robustness.

CVE Correctness. The evaluation showed that the performance depends on the availability and correctness of the CVE applicability statements. We found in 1.8% that the applicability statements are wrong or imprecise. As a mitigation, the software name could be extracted from the CVE description and used for finding the repository.

Source Diversity. For further enhancement, additional information sources such as the GitHub security advisories could be incorporated. The API endpoint for the GitHub security advisories did not work reliably at the time of our experiments but might be a valuable information source in the future.

Scoring Precision. The developed approach and first four heuristics to find patch commits are found to be highly precise. Nevertheless, there is room for improvement on the commit scoring system.

As for the *Description Fit* heuristic, a more sophisticated approach could be developed that goes beyond textual similarity e.g. by incorporate code semantics. The same applies for the *git-vuln-finder* heuristic. A machine learning model might improve the precision as it was developed by Sabetta and Bezzi (Sabetta and Bezzi, 2018). Their model yields a precision of 80% and a recall of 43%, but their tool is not published. The *CWE classifier* might be used as a stronger indicator if more CWEs are recognized. Currently, we focus on four vulnerability types to ensure high performance. When collecting more independent training data, additional vulnerability types can be included allowing to apply the classifier heuristic to more CVEs. Additionally, the CWE classifier could be improved by adding further data sources such as changelogs or comments in the source code.

While the existing heuristics already demonstrate good performance, additional heuristics based on new information sources could enhance it even further. Such information sources might be mailing lists or external issue trackers that contain information about the affected files or patches. The challenge is that this information is unstructured and the relevant data needs to be identified.

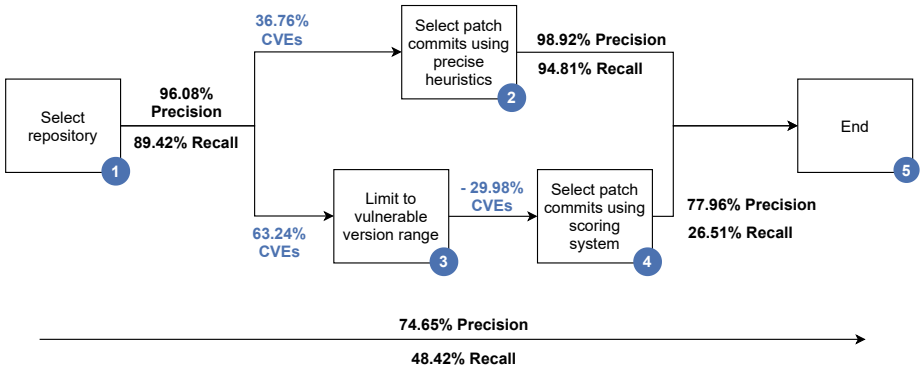

Figure 2: Evaluation of the Entire Approach.

6 CONCLUSION

This work presented a novel approach to find vulnerable open source repositories and patch commits matching given CVEs. We retrieved relevant CVE information directly from the NVD, identified repositories with a vulnerable code version, and finally checked for a patch commit to determine whether or not the vulnerability has been mitigated. To select the correct repositories and commits, we developed sophisticated scoring systems, instrumenting multiple heuristics, and conducted an evaluation with a data set containing 1000 CVEs, validating our approach.

We have found that the success of identifying the correct patch commits greatly depends on the repository. Well-maintained repositories provide more qualitative information to find connections to CVEs. Furthermore, especially patches to CVEs from the CWE-119 category are successfully identified. Good success rates are also demonstrated in the categories CWE-706 and CWE-74.

Future studies may be built upon the generated data set and our methodology to assess the effects of coding practices, guidelines, and clear documentation on code security.

REFERENCES

Chinchor, N. (1992). MUC-4 Evaluation Metrics. In *Proceedings of the 4th Conference on Message Understanding*, MUC4 '92, page 22–29, USA. Association for Computational Linguistics.

Corporation, T. M. (2020a). 2020 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. (Accessed on 10/07/2020).

Corporation, T. M. (2020b). About CVE. https://cve.mitre.org/about/index.html. (Accessed on 02/14/2021).

Dulaunoy, A. (2021). git-vuln-finder. https://github.com/cve-search/git-vuln-finder. (Accessed on 01/01/2021).

Gkortzis, A., Mitropoulos, D., and Spinellis, D. (2018). Vulinoss: A dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 18–21, New York, NY, USA. Association for Computing Machinery.

Greg Barish, M. M. and Minton, S. (2017). Mining commit log messages to identify risky code. In *Proceedings of the 2017 International Conference on Artificial Intelligence*, pages 345–349. CSREA Press.

Jimenez, M., Le Traon, Y., and Papadakis, M. (2018). [engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 56–61.

Ltd., S. (2021). Vulnerability DB — Snyk. https://snyk.io/vuln/. (Accessed on 02/09/2021).

of Standards, N. I. and Technology (2021). NVD - General. https://nvd.nist.gov/general. (Accessed on 01/14/2021).

Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. (2019). A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 383–387. IEEE Press.

Sabetta, A. and Bezzi, M. (2018). A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 579–582.

Sawadogo, A. D., Bissyandé, T. F., Moha, N., Allix, K., Klein, J., Li, L., and Traon, Y. L. (2020). Learning to Catch Security Patches.

working group, O. S. T. (2021). OpenSSF CVE Benchmark. https://github.com/ossf-cve-benchmark/ossf-cve-benchmark. (Accessed on 01/07/2021).