Vulnerability prealerting by monitoring the online repositories of open source projects

Andrzej Westfalewicz

Vulnerability prealerting by monitoring the online repositories of open source projects

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Andrzej Westfalewicz
born in Świebodzin, Poland

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.se.ewi.tudelft.nl

SIG Research Department
Software Improvement Group BV
Fred. Roeskestraat 115-123
1076 EE Amsterdam, the Netherlands
www.softwareimprovementgroup.com

Vulnerability prealerting by monitoring the online repositories of open source projects

Author: Andrzej Westfalewicz
Student id: 5247179
Email: A.Westfalewicz@student.tudelft.nl

Abstract

Software security plays a crucial role in the modern world governed by software. And while closed source projects can enjoy a sense of confidentiality when addressing security issues, open source projects undertake them publicly even though just as many projects rely on them. In 50% of documented cases, the vulnerabilities could have been spotted almost 20 days before their disclosure leaving plenty of time for a potential attacker to exploit the weakness.

Based on the results of a basic text search, we conclude that the majority of security-related activity is in reaction to known vulnerabilities and that maintainers are not always mentioning security terms when fixing exploits. We also confirm that many security-labeled issues are not pushed to vulnerability systems, even though the maintainers realize their security aspect. Then, while commit classification models can spot security-related commits automatically, the models struggle in realistic scenarios, and no particular feature or sampling method is vastly better than the others. Nonetheless, we evaluated the state-of-the-art models which spot security-related commits with an F1 score of 0.36.

Given the findings, we conclude that security-related activity is hard to automatically distinguish from everyday development activity and that manual review is required to spot these traces. Proposed methods can make this review easier. We suggest that more attention should be given to open source security to avoid early public traces of vulnerabilities.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. S. Proksch, Faculty EEMCS, TU Delft
Company supervisor:	Dr. M. Bruntink, Software Improvement Group BV
Committee Member:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft

Preface

It was quite a wake-up call to realize how fragile and potentially dangerous the software ecosystem currently is. It's really the case that some critical vulnerabilities are being discovered while a ticket on them was public for weeks. It's really the case that companies are using libraries that were not updated in years. And it's really the case that multiple projects are maintained by one person or not maintained at all...

I, after working for a bit as a software engineer, am guilty of ignoring some security concerns or known vulnerability warnings. But it needs to change. Vulnerability alerts have to be more respected, packages need to be kept more up-to-date and libraries have to be reviewed for their quality and sustainability before including them in solutions. To achieve this common effort is needed from all parties. Maintainers need to keep packages as backward compatible as possible to allow seamless updates. Managers need to set targets to keep as many vulnerabilities as possible out of the systems. Developers need to use packages more responsibly. Finally, companies need to give back to the open source community to ensure it can keep on evolving. In this thesis I commonly refer to security-relevant commits, but the truth is that all commits are important to security.

Special thanks to my supervisors: Dr. Magiel Bruntink and Dr. Sebastian Proksch who lead me through this project and kept me motivated, as well as to the entire SIG Research Team for the support along the way.

Andrzej Westfalewicz
Delft, the Netherlands
December 22, 2022

Contents

List of Figures

Chapter 1

Introduction

In the modern world defined by computer systems, software security is a critical subject. System vulnerabilities, data breaches and other types of exploits can cause serious damage to companies or governments either by financial or trust losses. Altogether, cybercrime has been estimated to cost the global economy approximately one trillion USD annually [57]. This can be partially attributed to software vulnerabilities. While developers can try to examine components, modern systems are too complex to be grasped and fully audited for security risks. Even when a company manages to fully secure its own code, there are still third-party components: frameworks, interface layers, or standalone supporting systems that a software project relies upon.

Where software is produced in an incremental fashion, many have decided to balance the risk by accepting the threat of vulnerabilities and fixing security issues as soon as possible. When the company discovers a security issue in their software (either by penetration testing, analyzing logs or from an external source) they patch the issue and release a new version of the product. As this happens, they can also choose to publish the information about a security release to encourage client projects to update to a newer version. This can happen via a security bulletin, or better yet, via a vulnerability database. Vulnerability databases are structured catalogs of known vulnerabilities with standardized data schemas, allowing automatic processing tools to determine whenever a given software stack has known vulnerabilities.

Disclosing vulnerabilities is especially useful in the case of open source packages, where potentially millions of projects rely on the package and the maintainers have no way of reaching out to them nor know who they are. Recent reports show that amongst these projects are also systems in critical industries like public, health, and financial sectors [51]. As such, vulnerabilities in open source gained a lot of popularity in research and industry in the latter years.

When a vulnerability is discovered, maintainers usually work on a patch and disclose the problem after releasing the fix. Optimally the information about the vulnerability is limited to a handful of trusted people before the fix is available, but because of the public nature of open source, work being done on fixing vulnerabilities eventually becomes public. We show that in a large number of cases vulnerabilities have public traces of security-related activity such as commits, reviews or issues long before the vulnerability is published, which is

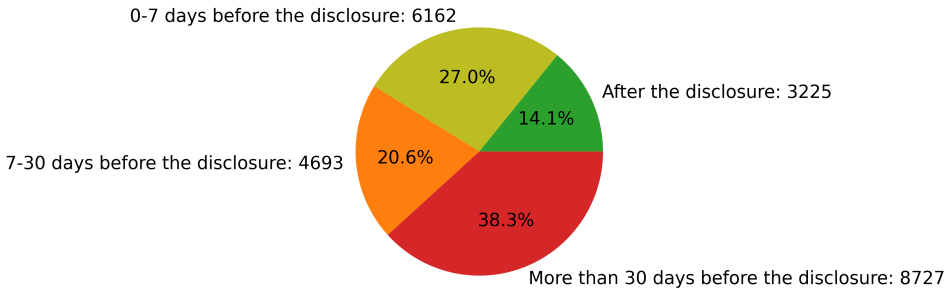


Figure 1.1: The number of vulnerabilities, from 2017 to 2022, by how much earlier their documented references were published.

confirmed by other works in the field [39]. This means that **an attacker could gain serious leverage by monitoring the online repositories of an open source project and trying out exploits that are being fixed or were recently fixed**. Because these issues weren't yet integrated or disclosed to the public, client projects don't know about the potential danger they are in [29].

Seeing this problem, the main goal of this project is to evaluate methods of finding the public traces of vulnerabilities and to characterize these traces. These methods can be implemented to provide tools that monitor open source projects the client project depends on and warn the developers if any security-related activity is published. Furthermore, by characterizing the traces and showing it's possible to spot we hope to encourage the open source community to more responsible handling of vulnerabilities.

We have performed four different experiments to determine the properties of security-related activity and to evaluate the performance of the methods listed below. Initially, we split the data according to the ecosystem or programming language of the open source project, but later explored also other classes of projects.

1. A method based around searching for security-related phrases like names of exploits or weaknesses. We also determine if maintainers are explicit when addressing security issues.

2. A method based on finding security-labeled GitHub issues. We also investigate whether maintainers always 'promote' a security issue to a vulnerability.

3. A method based around feature-based commit classification. We additionally explore the features of security-related commits.

4. A method based on deep learning (DL) commit classification. We explore different methods of sampling code to capture the security-related content, train our models, and evaluate the transfer learning potential of existing solutions.

We find that security-related commits and issues are similar in nature to regular ones. Many of them do not mention common security-related phrases. We also find that there is no particular feature for which distribution is largely different for documented security-related commits. This finding is also confirmed by the low performance of the data-driven approaches. We believe, based on manually reviewed objects, that a large contributor to these results is the fact that the same change can be considered a normal bug fix or a vulnerability fix depending on the context of the project or the file that was changed. Additionally, a valuable finding is that even when issues are marked in the repository as security related, they're rarely pushed to a vulnerability database.

Focusing on the DL commit classification. While our models achieved competitive results, due to a lack of resources we were unable to evaluate them properly and relied on the VulCurator [31] models (commit message classifier and VulFixMiner [59]) for the evaluation. These state-of-the-art tools do not perform as well in our evaluation as in prepared datasets, suggesting poor generalizability of the problem. Nonetheless, the models can assist during the review of security-related activity, especially since they work best for larger commits, potentially reducing the effort required. We also see that the performance of deep learning models is affected by the sampling methods, but the best possible performance is more likely determined by the training data rather than by the sampling method. We believe that as long as the sampling doesn't produce harmful artifacts, the transformer architecture can extract a similar amount of information from the same changes.

These findings suggest that it's difficult to automate the finding of security-related activity. While this is good news, as malicious parties cannot automate their attacks in this way, it's troublesome for smaller companies that cannot afford a security team to assess the current health of their dependencies. We believe that the goal should be to eliminate as many early references as possible and suggest some features to be added to vulnerability databases and repository hosting systems that could help with that.

Finally, to improve the results we recommend profiling the methods to specific repositories. For the deep learning models, we suggest utilizing context representation, e.g. the embedding of the file within the project and the context of the project itself. Furthermore, if a human is required to review suggestions of the model it's worth investigating the explainability of the model to provide hints on what to look for in the commit. Lastly, we recommend using a more thorough filtering process for acquiring the training data from vulnerabilities to create classifiers.

Contributions

1. We provide models and datasets for security-relevant commit classification.
2. We test out novel approaches to sample code from commits.
3. We evaluate existing state-of-the-art models in a practical scenario.
4. We highlight the pitfall of current disclosure processes and suggest improvements.

Chapter 2

Project overview

In this chapter, we present the overall motivation and the approach that we are taking in this project. First, we analyze the coordination process of fixing issues in an open source project to find the best approaches to spotting security issues before their disclosure. Then we analyze each of our proposed methods and formulate the research questions that can help us understand the security-related activity.

2.1 Approach

Many vulnerabilities have documented traces that were available a significant time before their disclosure, giving hints to attackers on possible exploits. As such, our goal is to understand the properties of these traces and devise methods for automatically spotting them. These methods should be able to spot any security-related activity in the project, especially those recent or non-disclosed. To determine the optimal approach, we look at the typical integration process code changes go through in an open source project.

Figure 2.1 shows the typical Coordinated Vulnerability Disclosure (CVD) [21] process. The process starts with the security issue being communicated to the maintainers, e.g. via GitHub issue or private message. Even though many repositories suggest contacting maintainers confidentially about security issues, it's possible that initially the security aspect of the issue was not known. After some time an issue is usually followed by some discussions before commits are pushed to the repository. When the changes are ready, usually a pull request is created so that a second maintainer can validate the code before it's merged to the main branch. After that, the change is included in a release and the vulnerability is disclosed to encourage client projects to update.

The CVD process is the recommended process by many vulnerability databases, including GitHub Advisory Database.[1] In this process disclosing the vulnerability is delayed until a fixed version is published. In the meantime, only a few trusted people should know about the exploit. For example, GitHub suggests creating a private fork for fixing the vulnerability so that both the discussion and the pull request review are hidden from the public. Nonetheless, after the change is merged into the main branch, it has to be public because of

[1] `https://github.com/advisories`

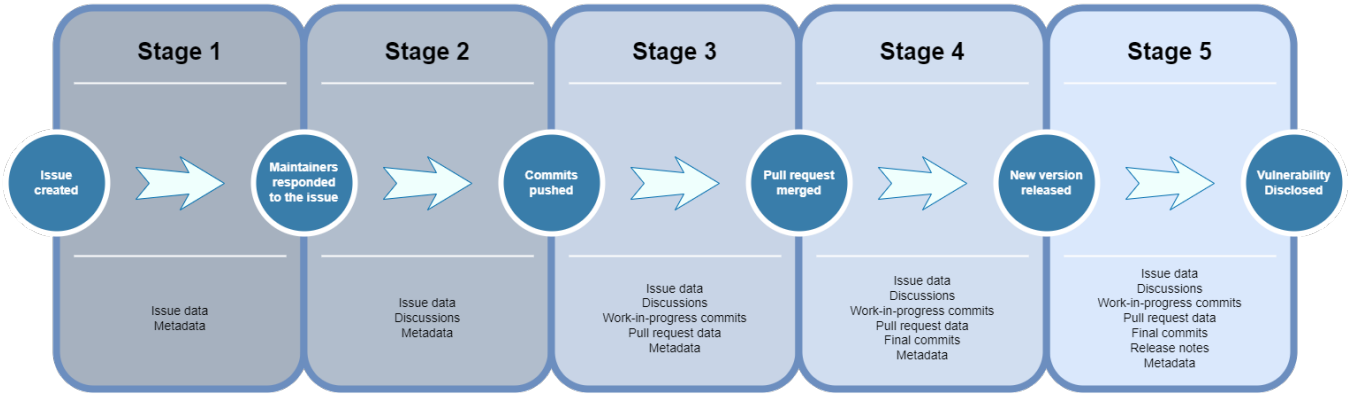


Figure 2.1: Typical coordination of vulnerability fixing in an open source project

the nature of open source. This applies only when the exploit was communicated privately to the maintainers or was discovered internally – if the user that discovered the bug was unaware of the security application and created a GitHub issue, the vulnerability is traceable publicly from the start of the fixing process. Naturally, the further in the process the more data is generated. Each stage is characterized by additional data being produced that makes classification hypothetically easier:

Stage 1 Issue created
In this stage, a user created an issue and it is awaiting to be handled by the maintainer. The only available data is the issue itself and its metadata. The issue may be invalid or just a question.

Stage 2 Maintainers responded to issue
Before work is done for a specific issue it is common for maintainers to discuss the issue in comments under the issue, assign the right developer or ask for more details. These discussions can be valuable for determining the importance and category of the issue. Maintainers can also assign labels to categorize them or close them if they're invalid.

Stage 3 Review stage
The review stage is marked by code changes being pushed to the online repository. Obviously, from this moment onward, code analysis can be used. Commonly the commits are first reviewed by other maintainers before they're merged into the main branch. Until the commits are merged in, they can still be modified.

Stage 4 Preparation for release
After the changes are integrated into the main branch the change is prepared to be released. This usually means testing the package on a specific frozen version to detect potential bugs, changing version numbers in manifest files or preparing release notes. Commits are now frozen on a public branch and metadata about the pull request (e.g. how fast the changes were approved) can be used.

Stage 5 Preparations for disclosure
After the new version was released the maintainers can wait with the disclosure to allow client projects to update before the vulnerability is communicated. This is augmented by the fact that attackers may get encouraged by the publication of the vulnerability in the first hours of it being out.

Depending on the data an approach relies upon, it can catch security-related activity in different stages of this process. Relying on the initial issue body or the discussions enables us to find the activity as soon as possible, but fixes for which these steps were made in secret will remain invisible for approaches relying on these objects. On the other hand, relying on, e.g. release notes, pushes the discovery date of security-related activities further away from any initial traces that were exposed to the public. First, we evaluate simpler, supporting methods in earlier stages to explore the security field: one based on issues and discussions, and one based on the labels assigned by the maintainers. Then we focus on our main method which will be relying on the commits pushed to the repository. We choose commit classification because:

1. While other data can be easily obscured by handling the process confidentially, the code changes have to be eventually submitted into the repository, as it's the nature of open source. Furthermore, while some issues are not created intentionally to reduce the traces, some issues might have never been considered, for example when the fix was performed 'accidentally', entangled with other changes or as part of a refactor.

2. As mentioned, GitHub is currently promoting fixing security issues on a private fork of the repository, meaning that issues and discussions won't be public. Since GitHub is our primary source of repository data, our approach should align with the recommendation and focus on the moment when these changes become public, i.e. after a pull request is created and quickly merged.

3. Using commit classification, one can catch security-related commits that were never discussed, possibly spotting silent fixes or vulnerability-inducing commits.

4. Code changes are a natural validator of whenever an issue was valid and required work to be fixed or implemented. When relying on the very first data, like the issues and the discussions, there is much noise created by invalid issues or simple questions to the maintainers. By placing our approach later in the process this noise is filtered out.

5. Based on the related work in the field, state-of-the-art deep learning models reach impressive results with performance metrics (like precision, recall, f1-score or area under the precision-recall curve) above the 0.70 mark.

As deep learning models tend to obscure the problem and provide only final results, we also test out a feature-based approach with statistical models (logistic regression, random forest, and support vector classifier). The advantage of testing out this approach as well is that in the process we can investigate the features of security-related commits and extract

feature importance from the models to understand the nature of the commits better. It's also common practice to apply these simpler models before moving on to more advanced deep learning methods, although these models tend to perform worse than a deep learning approach [58].

The disadvantage of relying on commits is that security issues can't be caught in the first and second stages of the process, as they have no commits linked to them yet. As such we propose the mentioned two supporting methods that do not rely on commits – one relying on simple text search and the other on GitHub labels. Theoretically, these should be able to find the vulnerabilities sooner than commit classification, which will be validated on historical data from vulnerability systems.

Additionally, investigating these methods will show more information on how maintainers handle security issues. First, by looking into the result of the text search we will gain insight into how explicit the maintainers are when addressing security issues. Are they mentioning the exploits and weaknesses or are they avoiding mentioning these phrases and only later discussing the vulnerability? Second, assuming that some open source projects are maintained by hobbyists, they might not bother pushing the security issues to vulnerability databases. As such we evaluate the security-related labels in selected repositories and determine for how many of them we can find vulnerabilities.

Note that both the software engineering and the security fields are very dynamic, with new practices, technologies, and solutions being introduced each year. As such in our project, we have decided to focus mainly on the last 5 years, from 2017 to 2022. Furthermore, we will analyze the dataset as a whole, but also zoom into the specific results for the npm, PyPI, and Maven ecosystems (which correlate to JavaScript, Python, and Java programming languages). These are the most popular and currently the most growing technologies in the industry. We hope to validate whether the ecosystem-specific results are different from the overall trends.

Given our motivation, we have selected commit classification as the main focus, but other approaches are also correct and also could have been applied. As security is a very active field of research, other approaches have been proposed relying on different data, for example: issue classification [61, 53, 23], release notes analysis [22], social media analysis [47, 20], or project quality metrics [49].

2.2 Research Questions

Now, let us discuss how these methods correlate to research questions to know what precisely we need to evaluate in the experiments.

Experiment 1 In the first experiment, we will scan the GitHub activity for the usage of security-related phrases. We want to know if security issues are mentioned openly or are being hidden.

Research Questions

1.1) Are the maintainers explicit when solving security-related issues?
1.2) With what performance can a phrase search spot security-related activity before disclosure?

In the first research question of this experiment, we want to know in how many cases the maintainers and the community of open source projects use typical security-related phrases when discussing issues. For this purpose, the most important evaluation will be verifying how many vulnerabilities have traces that use these keywords. Conveniently, this is simply the recall of the method based on using key-phrases to spot fixing vulnerabilities. This experiment will also validate the approach as means to filter entities to be processed in machine learning approaches explored later.

Keyword search is a very basic technique utilized by many other researchers as the first filter to acquire data. It was used as the first step of filtering to select the security-related issues for topic analysis [56]. Similarly, a text search was previously used to split GitHub discussions into security-related and background discussions [36]. Furthermore, phrase search can be used in combination with some statistical approaches, where the number of different phrases and their hit count is analysed [27].

Depending on the details of the approach (the used set of keywords or applied thresholds) the percentage of security-related entities in the results of the search varies from 0.11%, when the threshold is set very harshly, to 10% when more general phrases are used. As such selecting the phrases will be a crucial part of this experiment.

To our knowledge, no recent research has been done into correlating the security-related phrases to vulnerabilities or phrases used in the entities referenced by vulnerability reports. The majority of approaches recently, tend to use data extraction and more advanced natural language (NL) processing techniques than simple phrase search. In the second part of the experiment, we will determine the performance of the method based on text search.

Experiment 2 In the second experiment, we narrow down the search to just the labels field and validate if the data we are looking for isn't already labeled.

Research Questions

2.1) How many GitHub issues are security-labeled, but are not present in any vulnerability database?
2.2) With what performance can a lookup for security labels spot security-related activity before disclosure?

Here, we want to know if maintainers of some projects use security-related labels to categorize work in their repository, but ignore vulnerability databases. If so one could consider this as an alerting method – labeling an issue as security-related in the repository of your dependencies could be handled with the same weight as the dependency disclosing a new vulnerability.

The intuition is that maintainers will use security labels to assign priority to the issues. It's been shown that labels can be separated into 4 distinct families: Priority, Version, Work-

flow and Architecture [4]. Labels in the priority family indicate how important the issue is (e.g. prime); version labels specify the version that has the bug or that is targeted in the pull request (e.g. TF 2.5, TF 2.6.0-rc0, subtype: ubuntu/linux); workflow labels relate the status of the issue (e.g. ready to pull, awaiting review); and architecture labels annotate the components related to the issue (e.g. comp:runtime, comp:tensorboard). The provided examples were selected from the Tensorflow repository,[2] one of the most popular machine learning libraries, for which we can also see another class of labels about the type of the issue (e.g. type:bug, type:feature). Seeing this background we are confident that there will be some results to be found.

Sadly, these issues are quite rare – it was found in a study on 3,493 issues from GitHub under the 'security' label, that these issues accounted for 1.40% of scanned issues. However, the amount of them was rising [3]. Although it's estimated that a third of issues is miss-labelled [17] these issues are still a considerable amount of data. Issues under this label also had different resolution times, trends and relations to developers working on them.

To answer the first sub-question we will focus on filtering and manual review of security-labeled issues fetched for some repositories. The correlation between security-related labels and vulnerabilities wasn't described in much detail in reviewed papers. After checking how many of the issues are used on issues where maintainers have to fix exploits, we then map the issues to vulnerabilities disclosed in that project.

In the second sub-question, we look at the results of the experiment and evaluate looking for these labels as a method of spotting vulnerabilities. While the results of the review above decide how precise the method is, we also need to evaluate other performance metrics and investigate what filtering is needed.

Experiment 3 In the third experiment, we investigate the commits with metrics and begin our classification methods to spot fixes in code.

Research Questions

3.1) Can the security-related commits be separated from the background using commit metrics?
3.2) With what performance can feature-based commit classification spot security-related activity before disclosure?

In the third experiment, we want to analyze the security-related commits with metrics to see whether there are general tendencies of these commits and whenever they're different from the background. This might additionally help to design the final approach as we might design our sampling methods differently for large and small commits. First, we investigate them just by plotting them and investigating whenever they are visually different, and second, we design a feature-based commit classification approach that combines patterns across many metrics.

Then, we analyze feature-based commit classification as a method itself. Commit classification is a known topic in research, with common tasks being commit type prediction

[2] https://github.com/tensorflow/tensorflow/labels

(corrective, perfective, adaptive) [28], bug introducing [32] or security-relevant [43, 58]. Furthermore, we can identify three dominant approaches: feature-based approaches where the code is converted to metrics that are used to train the model; bag-of-words approaches which capitalize on techniques like IT-IDF or TF-IGM [8] to extract the most important phrases and tokens that are then processed by models; and deep learning approaches that aim to understand code as input. In this method, we limit ourselves to the feature-based approach to validate if a simple approach is good enough and to create a baseline for the deep learning model.

Experiment 4 In the fourth experiment, we train and evaluate our own deep learning commit classification model. We also experiment with different code sampling methods to determine the optimal way to preserve the context and security-related nature of the commit.

Research Questions

4.1) What impact on the model performance have different sampling methods?
4.2) With what performance can a bilingual deep learning model spot security-related activity before disclosure?

Different strategies and models have been proposed over the last 2 years in this field. Our idea is a combination of approaches suggested in Commit2vec by Cabrera et al. [5] and VulFixMiner by Zhou et al. [59]. First, VulFixMiner used CodeBERT [11] to generate the representation of the added code in each file in the commit which was then aggregated to the commit level. This approach has the limitation of cropping the data and not representing if one file had more changes than the others. Additionally, the next iteration of the transformer was released – GraphCodeBERT [14] which to our knowledge wasn't yet tested in the security-related classification task.

Second, in Commit2vec multiple samples were extracted from one file. As the samples were chosen randomly, the files with the most changes were represented the most. Applying this sampling method together with a transformer-based code embedding tool may yield much better results. Notably, both papers focused on the code representation and disregarded the commit message, which, being the description of the change, can provide more information. This gives yet another advantage to utilizing the transformer architecture which has been pre-trained on programming and natural language. The selection of these samples is the topic of the first research question when we test out 4 different ways of extracting them from commits and compare the results.

Finally, the embeddings have to be combined to the commit level, as individual lines or samples rarely are security related on their own. There were different aggregators proposed over the years to deal with the variable length input of the commit representation. In this project, we test 3 different approaches: the long-short term memory (LSTM) architecture, the convolution architecture and an averaging approach.

The model is then evaluated with the precision, recall and F1-score metrics. If the training of the model is too time-consuming we can fallback to utilizing existing models provided by VulCurator [31] which were trained on the SAP-KB dataset [37]. Evaluation

of this model will also shine a light on the transfer learning potential and the generalizability of the problem.

Investigating the recall Finally, in the last technical section, we combine the results of all the methods and evaluate them on different subsets of security-related data. In particular, we focus on how the recall of the methods changes when applied to vulnerabilities from specific years. Next, we validate if the recall is impacted by the severity of the vulnerability. Furthermore, do the properties of the reference itself, like commit size or the star count of the related repository, correlate to how easy it is to spot with suggested methods?

To solve these problems we use the same predictions as we did in the experiments, but group them not by the ecosystem or the model used to get the prediction, but by the specified properties. With this investigation, we hope to spot possible shortcomings of any of the methods or spot patterns in how their respective performance changes.

2.3 Performance Metrics

Above we refer frequently to the performance of the models, without quantifying what we mean by that. Especially for comparison purposes, the methods have to be evaluated in the same manner. For that we will use four criteria:

Precision The precision of the method quantifies the number of false positives the method yields. A low precision of a method would make it not viable to be applied as an alerting system as unnecessary alerts would make developers start to ignore them. Moreover, the lower the precision, the smaller the advantage of using the tool as a filter since you have to investigate unnecessary items.

Recall The recall of the method quantifies how many vulnerabilities (percentage vise) it can find. Especially required for filtering methods where the filtering should not reject positive samples. Based on how much before the vulnerability the traces were produced, we examine the recall at two moments in time, first at the day of disclosure and second a week before it. Additionally, the higher the recall the more one can trust the tool. Given the tool is extraordinarily high, one can start using the tool to skip releases if it was decided to have no security-related activity

Applicability The applicability of the methods represents how many scenarios the method can be applied to and whether it could be easily used in the industry. For example, if the method relies on specific labels being defined in a project or can be applied to one ecosystem, the applicability will be low. Obviously, low applicability is correlated with a low recall as if a method cannot be applied too often, it will not catch all vulnerabilities. As this metric is hard to quantify, we only classify the method whenever its applicability is low, medium or high.

Median Time Gain The median time gain is a measure of how much sooner the vulnerabilities are generally discovered. For this, we analyze the median of the delays between

the first spotted trace of the vulnerability and the disclosure date. Together with recall at a week before disclosure these two provide a more time-based evaluation.

To validate the precision, a manual review process will be usually required. In the process, we mainly focus on validating whether historically some changes/fixes were introduced in code and whenever these changes could be connected to some security aspects. These may include for example: validation, sanitization, memory handling or access control. While confident in our judgment we are not security specialists nor the maintainers of projects that we are reviewing. As such we consider commits/issues as background only if we are absolutely sure that the changes were not fixing a security problem, for example, if the changes were in CSS or documentation files or when the changes were purely cosmetic in nature. As a result, we might consider more objects to be positive suggesting our estimates on precision are an upper bound. Other metrics will be analyzed by applying the methods to the ground truth historical data acquired from the vulnerability systems described in the following chapter.

Chapter 3

Data extraction

In this chapter, we describe our process of gathering and preprocessing the data. We utilize the historical security data that was accumulated in the vulnerability databases as the ground truth. It's the validation data of the performance of our methods and the training data of the machine learning approaches. Furthermore, we describe the dataset of repositories used in case studies in the first and second experiments.

3.1 Security-related issues and commits

We extract the security-related objects based on the vulnerability reports submitted to vulnerability databases. Vulnerability databases agglomerate vulnerabilities for different products and ecosystems and store them in a standardized way. This way software development tools can automatically cross this information with products used in the environment and discover known vulnerabilities. In this research we consider 3 databases/feeds from the industry that we believe have the most influence on the environment.

National Vulnerability Database The National Vulnerability Database (NVD) and the Common Vulnerability and Exposure program (CVE) are the two oldest and biggest (volume vise) vulnerability database systems. Both of them originated in 1999. Currently, they are maintained by the National Institute for Standards and Technology (NIST) and MITRE respectively. While being separate projects they are constantly synchronized and as such, they hold the same data.[1] The CVE program is responsible for feeding the data into the NVD – this is done via a community effort of various partners CVE Numbering Authorities (CNAs) who can publish vulnerabilities into the system. The NVD is then responsible for storing and serving the vulnerabilities (while CVE and CNAs also have their own data stores).

Until September 2022 the database accumulated over 190k vulnerability reports, 101k of them being in the last 5 years. This data is well respected both in the industry, as it is used by various tools, especially maven dependency checker, and in the research community as it has been a backbone of a multitude of papers since its creation.

[1] `https://www.cve.org/About/RelatedEfforts`

As such including these two programs in our research is quite an obvious choice as it's the biggest dataset available and allows us to compare our results to previous work in the field.

GitHub Advisory Database GitHub Advisory Database or GitHub Security Advisories (GHSA) is a newer project launched in 2019 by GitHub which marked a big step for the platform in moving towards secure software.[2] It quickly grew in popularity as it was much easier for maintainers to manage repositories and vulnerabilities from the same website. Integration of the database in the same system as repository hosting and issue tracking improved the practicality of the vulnerability management process or feeding data in CI/CD tools like Dependabot.[3] The vulnerability database is split into two subsets: GitHub-reviewed which are manually reviewed security vulnerabilities or malware that have been mapped to packages in supported ecosystems; and un-reviewed which are vulnerabilities imported directly from the NVD feed. We decided to include GHSA in our research as we utilize GitHub references to extract code and issue tracking data. The GHSA is also the official source of vulnerabilities for the Node package manager[4] and one of the main sources for ecosystems like Erlang or Elixir.[5]

Open Source Vulnerabilities (OSV.dev) A relatively recent addition to the security ecosystem is the OSV.dev. Launched in 2021 and maintained by the Open Source Security Foundation led by Google.[6] The OpenSSF together with partners and open source community input, has developed a schema that eases looking up vulnerabilities in the case of open source packages. The OSV provides a shared feed and API of vulnerabilities from its partners who have adopted this format e.g. GHSA, Python Packaging Advisory Database (PyPA) or Global Security Database (GSD).
We decided to include this feed into our datasets as it agglomerates many databases from the open source world. Furthermore, as OSV is supported by some of the most influential companies and foundations in the software world, we expect it to grow in importance in the upcoming years.

Note that these are not all available data sources regarding current software security systems. First of all, many CNAs in the CVE program maintain their own lists of vulnerabilities for which not all are chosen to have a CVE identifier assigned. Examples of such CNAs are **snyk.io** or **OSSFuzz**. These were discarded from research as we believe that in the majority of cases vulnerabilities that are serious enough or are in popular packages will be pushed to the CVE system. It is also problematic to extract structured data as many of these CNAs internally use different data formats.

Furthermore, one can also extract exploit data from **bug bounty systems**. These are programs where (within some regulation) users are invited to hack and break the systems

[2] https://github.blog/2019-09-18-securing-software-together/
[3] https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/
[4] https://github.blog/2021-10-07-github-advisory-database-now-powers-npm-audit/
[5] https://github.blog/2022-06-27-github-advisory-database-now-supports-erlang
[6] https://security.googleblog.com/2021/02/launching-osv-better-vulnerability.html

of program participants and report the issues found. Users are then rewarded monetary bounties with the value depending on the severity of the bug. Some of the highest bounties reach up to 200,000$[7] for serious vulnerabilities. We discarded this data for similar reasons as the previous class of vulnerabilities, but additionally, many of the systems within these programs are closed source and out of access for this project.

It's also worth mentioning here the **Open Source Vulnerability Database**. It was a vulnerability database operating from 2002 until 2016. The Database was a community effort joining together different companies and individuals to look for and better analyze open source vulnerabilities. The project was shut down after many companies started utilizing their data for commercial uses without buying a license, as well as after the project had disputes with the CVE program. Luckily many vulnerabilities from the OSVDB were mapped to NVD before its closure. As we primarily focus on the last 5 years of software development (since 2017) this data is not of such high importance to us.

To sum up, we proceeded with data from NVD, OSV and GHSA.[8] The datasets were downloaded from official sites of each project on 2022.08.18.

Processing the data Before proceeding with the references we need to handle the duplicates and related vulnerabilities. The OSV schema supplies an alias field to indicate that the vulnerability is the same as other ones. We consider vulnerabilities A and B to be related if either of them contains the other in their aliases field (e.g. if B is present in the aliases field of vulnerability A). This mechanic of related vulnerabilities will have the most impact on the recall calculation as we consider the related vulnerabilities to share references, which makes it possible to spot a vulnerability with an object from another report. Additionally, we will consider the highest severity and the earliest disclosure date out of all related vulnerabilities. Note, that this logic shouldn't change the results if all properties in the disclosures are the same.

We construct the sets of related vulnerabilities to include more data for each of the analyzed vulnerabilities. But, we decided not to consider the vulnerability relation transitive, e.g. if A is related to B and B is related to C then A is not necessarily related to C. This decision was caused by some cases where multiple vulnerabilities from one dataset are related to one vulnerability in the other (for example PYSEC-2021-88). This also could introduce some significant inconsistencies in the case of vulnerabilities from Debian Security Advisories, where the aliases field is used to represent aggregates – for example, the vulnerability in Chromium identified as DSA-4824-1 is a shared vulnerability disclosure for 139 different vulnerabilities in the NVD. All of these vulnerabilities are related to the security update in Debian, however, they are individual vulnerabilities from the perspective of Chromium and putting all of them in one bag would hide some detailed data i.e. the actual number of vulnerabilities.

Since 2000 in total we analyzed 193,404, 8,696, 30,387 vulnerabilities from NVD, GHSA and OSV.dev summing to 232,487 vulnerability reports. By applying our alias logic

[7] https://redmondmag.com/articles/2022/08/15/microsoft-awards-13-7-million-in-bug-bounty-program.aspx

[8] NVD:https://nvd.nist.gov/vuln/data-feeds, OSV:gs://osv-vulnerabilities, GHSA:https://github.com/github/advisory-database/commit/18385a478b

we can join vulnerabilities together that have the same sets of related reports giving us the final dataset of 214,736 unique sets of related vulnerabilities. Figure 3.1 shows the distribution of vulnerabilities over time.

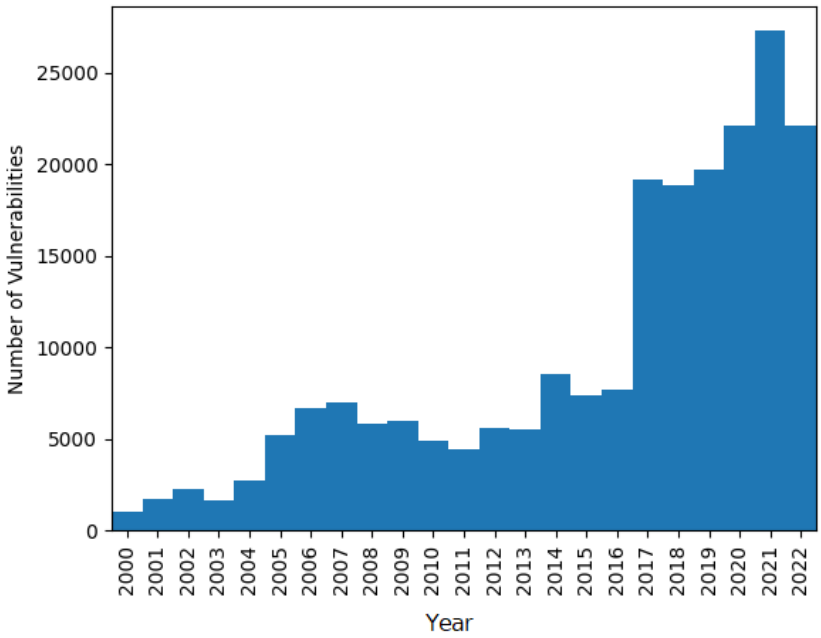


Figure 3.1: Number of disclosed vulnerabilities in each year since 2000. The sharp increase in 2017 can be explained by increased coverage coming from OSV and GHSA databases.

In the final step of preprocessing we extract all links in references pointing to GitHub, which is determined by investigating the host field in the URL. Then based on the segments of the URL we recognize the repository of the reference, the type of reference and the id of the referenced object. In the dataset we recognize 30,362 URLs to GitHub, among these, were 9,259 to commits, 9,345 to issues/pulls, 5,935 to blobs/trees and 342 to diffs/compares. By validating the uniqueness of each reference and resolving the blobs and tree to commits as described below we find 9,362 issues, 9,271 commits and 337 compares.

Extracting commits While for issue-based methods the retrieved issues can be used on their own, for commit-based approaches of RQ3 and RQ4, we need to resolve all references to commits. Directly linked commits are just validated whenever they still can be reached. For tree and blob references we extract the segment of the URL where we expect the hash of the commit or the name of the branch to be. Since the top of the branch might have had more commits pushed onto it, the link might not point to the same file version as initially when the vulnerability was published. As such, we proceed only with the commit hashes which we process in the same way as commits linked directly in the vulnerability. Compares and diffs

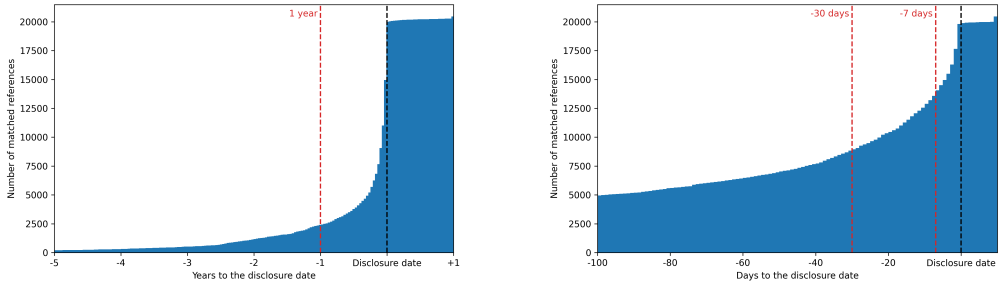
are also simple to resolve as for these we make an API request which returns the commits contained in the diff. We proceed with those as if they were linked to the vulnerability.

Much more complex logic is required for issues and pull requests which can represent more complex development processes than just code changes. Apart from the issue itself, we download the timeline, comments and events of the issue. These are used in cases when the developers were discussing the issue in the comments or when commits were referring to the issue. Furthermore, an issue can also represent a pull request (with which it shares the same number in issue tracking). Thus, we check whenever the issue doubles as a pull request and if so download the commits, pull request comments and the pull requests specific data from the GitHub API. Finally, users of issue tracking systems cross-reference issues and pull requests, for example: a very common mechanic is to create a new pull request with a different issue tracking number and cross-reference it in the original with a comment similar to *"Fixed in #123"*. To find these cases we use the 'cross-referenced' event in the timeline and download the commits and comments under related issues. To limit the generation of unrelated data we use only directly referenced, i.e. we do not follow cross-references to transitively linked. Having all this data, we extract all commits attached that are in the repository of the initial issue.

We noticed that some vulnerabilities link entire releases in the published reports, which might mean including normal, background commits as security fixes. To combat this we reject commits from references that resolved to more than 10 commits. We found that 10 commits were a good middle ground to reject the large pull requests, but still keep the majority of data. An alternative approach was to include all commits but assign less weight to them if they were resolved with other commits. This approach was rejected due to added complexity. In total we find 15,706 commits.

Investigating the delays We find that the median difference between vulnerability disclosure and GitHub objects referenced in the vulnerability is 17.4 days (15.8, 16.8, 42.3 for npm, PyPI, and Maven respectively). To get the ecosystem data we analyze the 'configurations' fields in the NVD reports containing CPEs or the 'affected' field in OSV reports. As seen in Figure 3.2 the references slowly accumulate over time as objects from years before a vulnerability was disclosed are sometimes referenced in it. This hints we are dealing with a long-tail distribution. For this reason, we use the median time as our estimator instead of the average, because it works better with a large range and with outliers. As expected, the plot flattens out at 0 days as vulnerabilities cannot reference not yet created objects and the majority of objects past this point are probably due to post-disclosure edits or related vulnerabilities. In total, for every ecosystem more than 96% of traces were created before the vulnerability. Additionally, 68.2% were created an entire week before the vulnerability (61.8%, 75.7% and 76.0% for npm, PyPI, and Maven respectively).

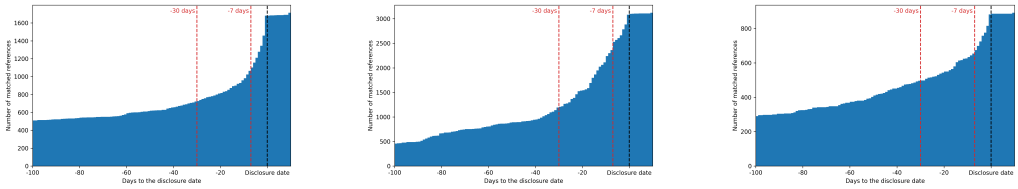
When zooming into specific ecosystems in Figure 3.3 some more specific behavior can be seen. First, reflecting on the total number of references and vulnerability pairs, the PyPI ecosystem leads with about 3,000 references, followed by npm with 1,600 and 900 for Maven. Because of this lower number of references, some bumps can be seen for example around 60(npm), 80(PyPI), and 15(Maven) days before the disclosure, which might be a result of different development processes or disclosure policies. Reflecting on the shape



(a) 5 year before and a year after disclosure, bin=12 days

(b) 100 days before and 10 days after disclosure, bin=1 day

Figure 3.2: Accumulated histograms of GitHub traces created n days before the disclosure. The x-axis shows the time difference between the traces and the disclosure date; the y-axis shows the number of traces with a time difference larger or equal to the x-axis.



(a) npm

(b) PyPI

(c) Maven

Figure 3.3: Accumulated histograms of GitHub traces created n days before the disclosure for specific ecosystems. The x-axis shows the time difference between the traces and the disclosure date; the y-axis shows the number of traces with a time difference larger or equal to the x-axis.

of the histogram itself, the plots for npm and Maven follow the overall distribution for all vulnerabilities, while for the PyPI ecosystem, the final month before the disclosure the plot looks much more linear. We did an in-depth analysis on this to find that for PyPI, the references are a bit shifted and the largest amount of references is made 6 days before the disclosure date of the connected vulnerability. We then validated that the mean values are indeed much higher than the medians. Looking for concrete results, shown in Table 3.1. We also see that the issues on average are preceding the vulnerabilities more than commits.

Finally, in Table 3.2 we show the most important finding for this project. If the traces only go back a day or two before the disclosure date, one could argue that no hackers will be able to prepare any attack before the vulnerability fix is published and the clients are alerted about the vulnerability. We see that overall 10% of vulnerabilities could have been spotted a week before based on the references found in them. Note, that this value for the PyPI ecosystem is above 50%. Furthermore, this value is calculated based on all vulnerabilities, including closed-source products and those without GitHub references. Additionally, when

Table 3.1: Count of found vulnerabilities and median delay in days for different kinds of traces.

Reference type	Whole dataset		npm subset		PyPI subset		Maven subset	
All entities	20,167	21.0	1,692	15.7	3,097	18.0	886	42.3
Issues only	9,991	27.6	666	32.8	871	40.2	518	38.6
First commit	14,794	16.7	1,512	12.1	2,841	16.1	636	49.1

Table 3.2: The percentage of vulnerabilities that had online traces n days before the disclosure date.

Dataset	0 days prior	7 days prior
Whole dataset	14.3%	9.8%
npm subset	37.7%	23.8%
PyPI subset	69.1%	52.9%
Maven subset	24.5%	18.2%

we analyzed the references we found that at least 60% of the referenced traces within the vulnerabilities are more than a week older than the report.

These results validate findings in prior work where similar values are reported. It also further motivates the goal of this project - if these vulnerabilities could have been spotted, why haven't they been disclosed to the general public? We also see that the gained time could be on average around three weeks, which is a very worrying result as for some security-critical projects, staying three weeks in potential danger is non-negotiable. We also see that (apart from the Maven ecosystem) issues preceded the vulnerabilities more than the commits which is in line with the expectation based on the process presented in Figure 2.1.

3.2 Repositories selection

For the first and second experiments, we need a set of selected repositories for which we will be downloading GitHub activity. These repositories should be a representative sample of open source projects, but at the same time, they should include the most important packages in ecosystems. We considered sampling these repositories from the vulnerability systems but decided that the fact that the repository is mentioned in a vulnerability disclosure already determines something about the practices implemented in the project. This is especially the case for the second experiment where the practice and the popularity of security-related labels will be investigated. Sadly, this abstraction cannot be applied to data-driven approaches where training and validation data has to be acquired from the vulnerability system.

As mentioned, we decided to focus on 3 major ecosystems: npm (Node Package Manager), PyPI (The Python Package Index) and Maven. Additionally, we decided to include another set of top-starred GitHub repositories that represent not only packages but general open source projects. These lists of packages are available in the reproduction package [52].

The most popular packages were downloaded using the portal libraries.io.[9] Using the search API we iterated over libraries on the platform with the highest page-rank score [33] in the dependencies graph. As there were some duplicates in the dataset, we kept only the most popular package for each unique GitHub repository. We decided to use page rank rather than downloads count as it should more closely reflect the importance of a package. Our goal was to acquire 1,000 packages from each ecosystem, but due to duplicated or invalid repositories, we fetched 999, 997 and 965 for npm, PyPI, and Maven respectively.

The top-starred dataset was fetched directly from GitHub using the Search API.[10] After initial investigation we realized that many of the top starred repositories function as social hubs or knowledge bases rather than being actual software projects – in the top 10 most starred repositories only two projects are code repositories (vuejs/vue and facebook/react). To balance out this issue, we decided to increase tenfold the size of this dataset compared to the packages sets to get a sufficient amount of repositories with. The final selected dataset had 9,906 distinct repositories.

[9] `https://libraries.io/`

[10] `https://docs.github.com/en/rest/search`

Chapter 4

Is a text search enough?

In this chapter, we perform our first experiments and evaluate the method based on a simple phrase search. We aim to answer whether maintainers mention the security issues in their discussions when working on a fix and whether the text search can be used to filter GitHub activity for spotting security-related issues and commits.

Research Questions

1.1) Are the maintainers explicit when solving security-related issues?
1.2) With what performance can a phrase search spot security-related activity before disclosure?

4.1 Experimental setup

To evaluate this method we need both the phrases we search for and the data we will scan. For our search we recognized the following objects containing natural language fields:

Issues titles and descriptions – GitHub issues are one of the main ways users communicate with the maintainers of the project and how work is organized in online repositories. Although many projects discourage people from making issues for security topics, users may ignore them or not realize the potential threat of the bug they found.

Pull requests titles and descriptions – The title and the body of pull requests should describe the changes it contains, possibly revealing that it's security related.

Issue comments – Issue comments are the place where the maintainers and users can discuss the issue and where someone could reveal how the issue is a security threat. Based on prior work, we expect the discussions happening under issues to generate a significant amount of data [36].

Review comments – Review comments are comments from a reviewer under a pull request with suggestions or opinions on the code change in the pull request. Whenever a reviewer realizes that the pull request is introducing a vulnerability, they will hopefully correct the author of the change with an explanation that we can pick out.

Commit messages – Commit messages of commits pushed to the repository can sometimes contain large descriptions of the change made including references to security concerns.

Commit comments – Finally GitHub users can comment on the code creating a commit comment. While these aren't usually used in the development process they can be used by users to mark a security-relevant line in the repository.

Fetching all these fields from the GitHub API for all selected repositories in our dataset would require several months of data scraping, which is outside of our resources. Luckily, the project GHArchive[1] is monitoring and saving the data feeds of public repositories creating a large, publicly available dataset. We used the dataset published on Google Cloud BigQuery as it allowed us to perform the query remotely within Google Cloud services and download only the results of the query. In the query, we used case-insensitive regular expression matching with each phrase sandwiched between a word boundary symbol(\b) to perform the text search. The query, raw results and the supporting data required to execute the query are provided in the reproduction package [52].

Phrases On the other side of the coin, we need phrases to search for in the selected dataset. The phrases have to cover as many security-related topics as possible and at the same time don't forfeit precision. In this experiment, we aim to prioritize the recall as the method is more likely to be used as a filter rather than an alerting tool. We construct our set of phrases based on three sets;

- The **base dataset** is the set of phrases used by Le et al. [27] to query for vulnerability-related discussions on developer Q&A websites. It contains 643 phrases written in different forms including spelling differences or acronyms.
- Phrases extracted from Common Weakness Enumeration (CWE) names.[2]
- Phrases extracted from security-relevant commits.

The first step was to filter through the base dataset to remove phrases that were potentially too general for our use. Then, to extract the phrases from CWE names we manually review the list of CWE titles. As long strings of words are unlikely to be matched exactly in a text search, we extracted the most important words from the CWE names, shortening them, at the same time keeping at least two words to reduce false positives. We also reject phrases that are likely to be normal, non-security-related bugs and reject technology-specific CWE e.g. related to 'Struts' or 'J2EE Bad Practices' CWEs. In this step, we also add several phrases that are referring to the same problem as the CWE but use different words which might be used by developers in discussions. Usually, stemming is applied to construct a more general representation of the keyword. However, as some of our phrases were very short keywords (e.g. dos, zap), we decided to look for different versions of the phrases as an entire word to exclude cases where the keyword was a substring within a word.

[1] `https://www.gharchive.org/`

[2] Version 4.8 was used available at `https://cwe.mitre.org/data/`

To extract code keywords we mine patch data of security-relevant commits extracted from vulnerabilities as described in Chapter 3. We apply NFKD normalization [24] to each added line in the commits. We then break apart multipart tokens, where a small letter is followed by a capital letter to account for casing used in programming languages. We reject tokens that are shorter than 4 characters and longer than 20 to include only meaningful and non-shortened keywords. We then repeat the process for background commits mined from the same repositories. Mined tokens are then sorted using a TF-IDF [38] inspired approach. While the purpose of TF-IDF is to extract the most important terms from documents, we aim to extract the tokens that are the best for discriminating the positive (security-related) commits from the background. We define the term frequency of a token as the total number of times the token appears in all commits divided by the number of commits to account for imbalanced data. The document frequency is then the percentage of commits the token appeared in. We use these metrics to calculate ratios which we designed to be the highest for the most discriminative tokens.

$$ratio_1(t) = \frac{DF_{positive}(t)}{DF_{positive}(t) + DF_{background}(t) + \varepsilon}$$

$$ratio_2(t) = \frac{TF_{positive}(t) * DF_{positive}(t)}{TF_{background}(t) * DF_{background}(t) + \varepsilon}$$

$$ratio_3(t) = \frac{TF_{positive}(t)}{DF_{background}(t) + \varepsilon}$$

Ratio 1 is the ratio between document frequencies which expresses the precision given by the token. We use this ratio to remove all tokens that would yield a precision lower than 0.95 in a balanced scenario. Because we chose to mine the data with more background commits the threshold is lowered by multiplying it by the ratio of positive to background commits. For simplicity, we added a zero guard to the denominators (ε) to avoid division by zero errors. We then use the other ratios to create two sorted lists from each extracting around 100 tokens that could be considered security-relevant and are not too general for our search.

After all these steps and once more deduplicating the set of phrases for those that would be detected by another phrase, we arrived at 770 phrases.

Manual review Finally, the results of the query are manually reviewed. We first look at what keywords were found the most often and which did not occur in the scanned repositories. Then, we look whenever there are anomalies in specific repositories that have much more found phrases than their size suggests. Next, we manually review the most found and the longest phrases and reason the precision of the method.

4.2 Results

The considered timespan for the query was from 2017 until August 2022 (inclusive). In the GHArchive query, a total of 41.9M GitHub objects were scanned, from which 2.2M

contained security-related phrases. 1,532 repositories out of 11,857 (12.9%) had no activity with security-related phrases in the inspected period. Out of these, 62 repositories had no activity record on GHArchive. Moreover, 152 out of 773 (19.7%) phrases were not found - not surprisingly most of them are more complex phrases that are less likely to be written exactly in the way we search them for or those extracted from code.

Precision Sadly, as seen in related work, the major shortcoming of this method is its low precision. In our case, this can already be seen by a large number of returned entities – 2.2 million potentially security-relevant issues and commits for around 13 thousand repositories. Considering that in this period around 150 thousand vulnerabilities were disclosed in general (not only to the repositories considered in the search), we sadly have to expect that the majority of hits are false positives.

In the manual review, we first have a look at the most frequently found phrases in the dataset, as these have the most influence on the precision of the method. Not surprisingly these were simple phrases. Some of them can be also associated with usual development processes suggesting a bad choice of key phrases. The results of this analysis are shown in table 4.1. The first finding immediately explains the large number of false positives found with this simple method. Many of the vulnerability-specific phrases like ”vulnerability”, ”vulnerabilities” or ”cve” return a lot of issues created by bots that are suggesting an update to packages in the system due to a known vulnerability. While they are not false alerts as these commits are security-related, they are not useful when trying to spot security issues before the vulnerability is published – since these are reactions to already existing vulnerabilities that are also in another repository.

Second, we notice a high percentage of valid issues for phrases like ”crash”, ”crashing” or ”leak”. Indeed, issues with these phrases very often were fulfilling the requirement described in the setup – they were linked to valid code changes, however, they might have been regular bugs for which we cannot account for without a deeper knowledge of the project.

Third, the phrase ”overflow” shows the issue described more in the applicability analysis, where in one context (e.g. C++ code) it is a valid threat to the application while in another (e.g. CSS) it is a normal language statement.

Finally, phrases like ’sandbox’ or ’secrets’ were often connected to continuous testing practices, where bots were commenting issues with links to development environments, i.e. sandboxes, where the code change can be tested. This lowers the hit count of actual security-related issues.

The next group of results we investigated were the results of the longest phrases. As these phrases are more specific we expected more precise results as these phrases should not come up in everyday software development discussions. Table 4.2 shows 29 phrases that were found in our data and are longer than 28 characters. If there were more than 50 hits for a phrase, we randomly sampled 50 of them. The first class of results found with these long phrases were vulnerability references. Some of our phrases were used within the titles of vulnerabilities and thus were included in the majority of alerts triggered by the vulnerability in automatic pull requests. For example, the phrase ”improper privilege management” was always connected to vulnerabilities, one of them being CVE-2022-0144.

Table 4.1: The fraction of objects that were found with the most popular keywords and were linked to potentially security-related code changes or referencing a known vulnerability.

Phrase	#	With code changes	CVE responses
crash	50	0.72	0.08
crashing	50	0.66	0.00
cve	50	0.16	0.78
leak	50	0.74	0.12
overflow	50	0.30	0.06
sandbox	50	0.18	0.02
secrets	50	0.30	0.02
trusted	50	0.26	0.02
vulnerabilities	50	0.02	0.86
vulnerability	50	0.12	0.64

The second group of results were phrases that are used by static analysis bots. As we found, many open source projects set up CI pipelines that respond to issues and pull requests in comments. Some of these responses were found by looking for the phrases, for example, one of the alerts commented by the LGTM bot[3] is 'Use of a broken or risky cryptographic algorithm' which was detected by our phrase 'risky cryptographic algorithm'. This comment was detected for example in pull 7404 in Apache Geode.[4] While our initial goal was to spot security-related discussions in comments under the issues, this gives the method additional opportunities of utilizing outputs of in-depth static analysis without the need for rerunning it or purchasing a license for the analyzer.

Third, even when the content of the issue or the discussion under it is mentioning a very specific weakness, it is still possible that the issue is invalid or relates to documentation which is the reason sometimes the issues were neither valid code-changing issues nor a reaction to a disclosed vulnerability.

In another investigation, we looked at the entities that contained the most distinct security phrases and entities which had the most comments with security phrases. We found that in many cases these entities were serving as discussion threads to which other pull requests and issues were linking to. Furthermore, some of these discussion threads were preparations for a release or pull request integrating a large number of changes. These entities naturally agglomerated a lot of content from which some were mentioning security-related phrases. While in our evaluation these are potentially security-related, from the perspective of developers it makes no sense to review them due to their size and the developers are more interested in the individual issues and changes.

Note that some objects, even though they have plenty of mentions of security-related phrases, have very few linked changes. For example, this was the case for discussions about the controversial node-ipc vulnerability (CVE-2022-23812), which attracted a lot of atten-

[3] `https://lgtm.com/`

[4] `https://github.com/apache/geode/pull/7404`

Table 4.2: The fraction of objects that were found with phrases longer than 28 characters and were linked to potentially security-related code changes or referencing a known vulnerability.

Phrase	#	With code changes	CVE responses
exposure of private information	2	1.00	0.00
exposure of private personal information	15	0.67	0.93
expression language injection	9	0.11	0.22
external control of file name	10	0.70	0.10
external control of system or configuration setting	1	0.00	0.00
externally-controlled format string	4	0.25	0.00
failure to restrict url access	1	0.00	1.00
improper certificate validation	50	0.04	0.82
improper privilege management	15	0.00	1.00
improperly controlled modification	14	0.00	1.00
improperly implemented locking	1	0.00	0.00
incorrect default permissions	6	0.00	0.67
incorrect permission assignment	5	0.00	0.80
inefficient regular expression complexity	50	0.06	0.92
insecure direct object reference	11	0.09	0.27
insufficient granularity of access control	1	1.00	0.00
insufficient session expiration	6	0.67	0.00
insufficiently protected credentials	9	0.00	0.67
mismatched memory management routines	1	1.00	0.00
missing function level access control	6	0.33	0.00
modification of assumed immutable data	1	1.00	0.00
non-serializable object stored	2	1.00	0.00
open web application security project	14	0.14	0.07
password hash with insufficient computational effort	8	1.00	0.00
permissive regular expression	6	0.67	0.00
restrictive regular expression	2	0.00	0.00
risky cryptographic algorithm	50	0.36	0.52
uncontrolled resource consumption	50	0.14	0.74
uncontrolled search path element	8	0.25	0.63

tion even though the initial issue and code change were small and the issue is now deleted.[5] Note that this highlights one of the benefits of implementing this tool as a monitoring tool where the events feed is constantly scanned, as it will catch also the issues that are quickly removed. Similarly, issue 192 in guzzle/psr7[6] was redacted to not show any information on

[5] https://github.com/RIAEvangelist/node-ipc/issues/233

[6] https://github.com/guzzle/psr7/issues/192

the exploit. However by inspecting the events of the issue, one can discover that the previous title of the issue was "CWE-73: External Control of File Name or Path". Finally, precision is also discrepant in cases where the nature of the project itself is security-related. The most common project of this kind we encountered in the manual review was the CodeQL[7] project which scans projects for vulnerabilities. This naturally means that it will reference weaknesses and other security-related phrases more often.

To sum up, only about 35% of the objects retrieved with this method contained code changes. The actual percentage of issues that are security-related is likely to be much lower, but as we aren't the maintainers of the projects nor security experts we cannot deny their security-related nature. The major contributor to low precision is the bot-generated content by bots alerting for vulnerabilities in other projects.

Recall In Table 4.3, we see the results of applying the phrase search to the mined references of vulnerabilities. In general, the percentage of vulnerabilities with references containing security phrases is only about 53%. The value falls to about 37% when one counts the issues created a week before the disclosure. Note that the values are highest for the Maven ecosystem suggesting that the phrases list was matching this ecosystem the most. Nonetheless, this value is much lower than we expected. It may suggest that many of the vulnerabilities initially are fixed as normal bugs and are later 'promoted' to a vulnerability. Alternatively, it may mean that the security aspect of vulnerabilities is being actively hidden until the disclosure which is a practice that should be encouraged.

For recall analysis, we also investigated which phrases were the most common in the vulnerability-referenced objects. As one could expect the most popular phrases are the ones like 'nvd', 'cve', or 'vulnerability'. These phrases might have been mentioned to create the vulnerability or to link it to the record in the database. On the other hand, 'overflow' and 'xss' phrases are more specific to concrete examples of exploits and errors connected to the vulnerabilities. Their relatively high number can be explained by a high number of vulnerabilities from projects related to Linux, where integer overflow is a major concern, and in the PHP ecosystem, where web vulnerabilities like cross-site scripting (XSS) are a critical issue. This relation however was not firmly explored. We would recommend using these frequently hit phrases if their context matches the project in question. The results of this analysis are shown in Table 4.4.

Table 4.3: The percentage of vulnerabilities with issues containing security phrases public n days before the disclosure among all vulnerabilities with documented issue traces

Dataset	0 days prior	7 days prior
Whole dataset	53.3%	36.6%
npm subset	51.2%	33.6%
PyPI subset	47.0%	33.4%
Maven subset	57.1%	41.4%

[7] https://github.com/github/codeql

Table 4.4: The most found phrases in vulnerabilities together with the number of unique entities containing them.

Phrase	# vulnerabilities
cve	5205
vulnerability	4410
overflow	2450
xss	2232
attacker	1650
crash	1412
attack	1364
malicious	1267
vulnerabilities	1121
exploit	957

Median time gained As shown in Table 4.5 the median delay is about 20 days with values from specific ecosystems varying from 17 to 34 days. Notably, these values are a bit lower than the total value found for all data in Tables 3.1 in chapter 3, possibly suggesting that the first content generated in the issue doesn't contain the phrases and those are added later in e.g. discussions. Still, the median value is generally higher than the median delay from the first commit which is understandable based on the integration process described in the project overview.

Applicability While the method has no theoretical limitations as text search can be performed in any repository, there are some practical limitations. First, the phrase list should be profiled to the ecosystem or even the project it is used on. As seen with the evaluation of the 'overflow' keyword while in some cases it was referring to a potentially dangerous bug in the backend logic, sometimes it was referring to an inconvenience in the UI of the project. Furthermore, some repositories use bots to automatically create a comment response to new issues with checklists or extra information. In these cases, one has to obviously remove all phrases that are present in this automatic response. The same applies when the repository defines a template for issues or pull requests.

Second, the method is very reliant on the curated phrase list. If a reference doesn't mention any of the keywords or the maintainers make a typo when describing the exploit a simple text search may fail to spot an obvious vulnerability. Such a flaw makes the method not valid in some business applications. All things considered, we would say that the applicability is medium.

4.3 Summary

We see that only 55% of the objects referred by vulnerabilities contain the selected security-related phrases. This means that maintainers may be explicit when solving security-related

Table 4.5: Median delay and the number of found vulnerabilities using security phrase search.

Dataset	Found vulnerabilities	Median Delay [days]
Whole dataset	12,494	22.3
npm subset	887	16.9
PyPI subset	1,474	21.1
Maven subset	510	32.8

issues, however, it doesn't happen as often as one would expect.

The major flaw of this method are objects created by bots that generate a lot of noise. Furthermore, many big changes naturally agglomerate a lot of security-related phrases which are again impractical to review. The methods' performance metrics are:

- Based on the review of the most commonly found phrases we conclude that the precision of the method is equal to 35%. It's worth noting that this is an upper bound, as some of the objects considered by our relaxed conditions as security-related might not be security-related. Furthermore, the precision highly differs from keyword to keyword.
- By repeating the search on objects in vulnerabilities we find that the recall a week before the disclosure is at least 36.6% and recall at the disclosure date is equal to 53.3%.
- The method has a medium applicability due to its dependence on the phrase selection and that its phrase list has to be adjusted between ecosystems or project types.
- Method finds vulnerabilities with a 22 day median time difference between the issue creation and the vulnerability disclosure date.

Chapter 5

Are security-labeled issues vulnerabilities?

In this chapter, we look into the issues labeled on GitHub as security-related and determine if they represent the same problems as vulnerabilities. If so could they be considered equal to vulnerabilities? To get these answers we evaluate a prealerting method based on security-related labels.

Research Questions

2.1) How many GitHub issues are security-labeled, but are not present in any vulnerability database?
2.2) With what performance can a lookup for security labels spot security-related activity before disclosure?

5.1 Experimental setup

The first step is to decide which labels we consider as security-relevant. In the work done by Buhlman et al. [3] only issues with the 'security' label were downloaded. While their results were satisfactory, we believe that these are too restrictive and labels such as 'vulnerability' should also be included. To validate this belief we have counted the most popular labels on issues referenced in vulnerability reports. The 10 most popular labels are shown in Table 5.1. We need to exclude general labels like 'bug' or 'enhancement' that usually will not be assigned to a security-relevant issue. Quite common are also labels related to a software testing technique called fuzzing based around providing randomly modified inputs to programs. While their high count is likely caused by the project OSS-Fuzz,[1] bugs resolving from these random inputs may not be security issues and thus are discarded. Moreover, since our goal is to operate in the timespan before the vulnerability is disclosed the 'cve' label is of limited value for our task.

[1] `https://github.com/google/oss-fuzz`

Based on this analysis we decided to include all labels that contain the keywords 'vulnerab', 'secur' or 'exploit'. These keywords will cover the majority of labels used in the projects and are related exclusively to security-related data.

Table 5.1: Top ten most popular labels in vulnerability reports.

Label	# of vulnerabilities	Covered
bug	1751	No
security	585	Yes
kind/bug	163	No
cve	147	Yes
enhancement	126	No
invalid	124	No
fuzzing	113	No
area/security	112	Yes
the bug slayer	92	No
module	76	No
vulnerability	74	Yes

Downloading and processing the issues Each repository in the selected dataset is processed in four steps:

1. Using the GitHub Search API we find labels that match selected keywords.

2. For each label we download the closed issues.

3. For each issue we download related issues and commits like discussed in subsection 3.1

4. Issues not connected to any commits are rejected.

In the first step, we look for any labels in the selected repositories that contain any of the keywords. Then, in the second step, we download closed issues under found labels. We fetch only closed issues to be able to determine if the issue is valid by checking whenever any code changes are linked to the issue. Due to the constraints of the GitHub Search API, we are limited to only 1,000 results returned for a search which wasn't always enough in a few cases. In these cases, we used the available 1,000 most recent issues.

The third and fourth steps are required to avoid the pitfall of the previous research question of bot-generated content. After an initial investigation, we realized that the majority of the content is generated by bots updating dependencies, like Dependabot,[2] Renovate,[3] or

[2] https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/
[3] https://docs.renovatebot.com/

Snyk-bot.[4] To remove these, in the third step we download details of the linked commits and in the fourth step we reject all issues which had no commits with code changes linked.

After the automatic processing, the issues are manually reviewed. We once again verify the connected changes and reject those for which we are certain that they aren't security related. Furthermore, we attempt to link the issues to vulnerabilities. While some issues explicitly specify in their content the CVE identifier of the vulnerability created for the issue (sometimes retrospectively), in most cases this data has to be downloaded and manually matched. We extract vulnerabilities by comparing whether the repository name in the issue URL matches up with any reference in the vulnerability. We additionally used the repository name to search for CPEs in the NVD API and then added vulnerabilities under potentially matching CPEs as well.

5.2 Results

Applicability We evaluate the applicability of the method based on the percentage of repositories defining labels with our keywords. As expected by prior results [3], applicability is not particularly high as at most 12.2% repositories define the labels as shown in Table 5.2. Additionally, we see that the percentage drops further when we filter for issues matching our criteria. The highest percentage is observed for the Maven and most starred repositories datasets for which the percentage is still only 4%, which sets the applicability as low.

Table 5.2: The number of repositories per ecosystem with security-related labels, issues under these labels and issues that were full-filling criteria of RQ2.

Repositories...	Most starred repos	npm repos	PyPI repos	Maven repos
All repositories	9906	999	999	965
With defined labels	830 (8.4%)	109 (10.9%)	79 (7.9%)	118 (12.2%)
With any issues	567 (5.8%)	65 (6.5%)	39 (3.9%)	67 (6.9%)
With potential issues	369 (3.8%)	18 (1.8%)	21 (2.1%)	39 (4.0%)

Precision As presented in Table 5.3, results of the manual review show that the precision of the method is between 56% and 77%. This confirms that the applied filtering increased the percentage of valid issues in the dataset. The Maven ecosystem has the most security-labeled issues, more than double the counts for other ecosystems, which correlates with the fact that it has the most repositories defining the labels. This result is slightly unexpected, as in Chapter 3 in the Maven ecosystem we found the lowest number of vulnerabilities with GitHub references. Notably, the automatic filtering for linked changes is not perfect as issues introducing only configuration changes were frequently referenced in other pull requests or contained additional changes (e.g. automatic code styling).

[4] https://docs.snyk.io/integrations/git-repository-scm-integrations/github-integration

Table 5.3: The number of issues that we downloaded, manually review and are possibly security related

Issues...	Most starred repos	npm repos	PyPI repos	Maven repos
All issues	8862	629	300	785
Potential issues	4174	142	145	330
Manually reviewed	116 / 150 (77.3%)	80 / 142 (56.3%)	104 / 145 (71.7%)	99 / 145 (68.3%)

Recall Finally, the results of the recall analysis are shown in Tables 5.4 and 5.5. First, we see that in general only 80.4% of issues are created or labeled as security-related before the disclosure of their vulnerability. We think that the major reason for this is assigning the label post factum and issues that serve as documentation. The ratio drops further for issues older than 7 days at disclosure time possibly suggesting that issues with security labels are being handled with higher priority.

Second, as expected from the low percentage of repositories defining the label, when considering all vulnerabilities the recall drops significantly to values around 8%

Table 5.4: The percentage of issues that were labeled security-related n days before the disclosure.

Dataset	0 days prior	7 days prior
Whole dataset	80.4%	49.8%
npm subset	90.9%	77.2%
PyPI subset	64.0%	37.8%
Maven subset	80.8%	46.2%

Table 5.5: The percentage of vulnerabilities with linked security labeled issues out of all vulnerabilities with linked issues.

Dataset	0 days prior	7 days prior
Whole dataset	8.1%	5.0%
npm subset	6.0%	5.1%
PyPI subset	8.2%	4.8%
Maven subset	7.7%	4.6%

Median delay The last metric we evaluated is the median time delay to disclosure. As the date when the issue was spotted by the method we have selected the date when it was assigned a security-related label. The results of the analysis can be found in Table 5.6. We see that in the majority of cases the issues are spotted much closer to the disclosure date – with a median delay of 7.1 days. From the npm ecosystem, the delay reaches over 100 days,

which might be the result of artifacts and a low number of found vulnerabilities. The PyPI and Maven ecosystems show values even lower than the general median of 0.8 and 4.7 days of delay respectively.

In total, these values are much smaller than the creation date of issues explored in Chapter 3. We believe there are two reasons for this, similar to the reasons for low recall. First, in many cases we manually reviewed we found that the issues were receiving the security labels retrospectively – it was not used to prioritize work, but rather as documentation means. Second, after the security label is assigned to an issue it means that the maintainers realize that the issue has security applications and fix it urgently. Both of these actions skew the median closer to the disclosure date.

Table 5.6: Median delay and the number of found vulnerabilities using security-related labels.

Dataset	Found vulnerabilities	Median delay [days]
Whole dataset	1,051	7.1
npm subset	44	103.2
PyPI subset	111	0.8
Maven subset	50	4.7

Mapping to vulnerabilities Additionally, for the first sub-question, we need to know how many of these issues refer to vulnerabilities. We managed to find a vulnerability for 18.8% of the correct security issues. Interestingly, when comparing the references in vulnerabilities, only the only matching repository was *tensorflow/tensorflow*, suggesting that the documentation is incomplete or it is rarely two-directional. Sadly, this result has to be taken with a grain of salt, as with manual matching some vulnerabilities might have been missed or assigned incorrectly. From these non-matched issues, one could extract a set of 12,937 commits linked to security-labeled issues, which could be an alternative source of security-related commits sourced differently than from the vulnerability database. However, while mining these commits additional filtering needs to be applied to reduce noise as for some ecosystems only 56% of issues were correct.

5.3 Summary

We see that only 18.8% of the reviewed security-labeled issues were mapped to vulnerabilities, suggesting that many potential vulnerabilities are marked on GitHub, but undisclosed to any vulnerability database.

Thanks to its relatively high precision in repositories where the labels are used to coordinate, the method has great potential as an alerting tool similar to how vulnerabilities are considered. However, a major drawback of it is that not many repositories utilize considered labels for coordinating fixing efforts.

- The precision of the method stands at 77% with values for ecosystem subsets varying from 56 to 72%. We suspect the method's performance could be improved by combining the applied code-changing commits filtering with filtering based on the author of the issue.
- The recall of the method is 8.1% with values for ecosystem subsets varying from 6 to 8%. The main reason for this low value is that the majority of vulnerabilities are being handled by more generic labels e.g. 'bug' or 'enhancement'.
- The applicability of the method is low because not many projects define security-related labels.
- The median delay time is 7.1 days with values for ecosystem subsets varying from 1 to 103 days. As this result is lower than the overall median for the issues, we believe that labeled issues are handled with increased priority or are labeled post-factum for documentation purposes.

Chapter 6

Features of security-related commits

In this chapter, we are moving to commit classification. First, we look at the commits through the lens of metrics to understand the security-related commits better. Then, we test out feature-based models to investigate how a combination of metrics can separate the set of security-related commits from the background. Furthermore, these models can be used as another method and as a baseline for deep learning approaches. In this experiment, the main points will be picking the right metrics that can be different for security-related commits and choosing the right model to combine these metrics in the best way to separate them from the background.

Research Questions

3.1) Can the security-related commits be separated from the background using commit metrics?
3.2) With what performance can feature-based commit classification spot security-related activity before disclosure?

6.1 Experimental setup

The first step in implementing this method is to collect the train and test datasets. First, the positive samples will be the security-related commits described in Chapter 3. We filter this dataset to include only repositories for which we find at least 5 security commits. We do this to reduce the space of values for our features and exclude one-off commits in arbitrary repositories. This will also reduce the mining time as not so many repositories need to be cloned. Second, we also need the background commits from which the model will learn to distinguish security commits. As mining entire repositories would require too many resources and makes us include many silent fixes, we mine background commits with an approximate ratio of 1:50. As this ratio is enforced by mining background commits with a pre-calculated probability the actual ratio might vary. Additionally, not all commits found in vulnerabilities are still present in repositories. Some commits might also contain no code changes (e.g. merge commits) or have null values for many of the metrics and are

also rejected. As such the actual ratio might be different and will be re-balanced to fit the training in further stages.

Feature Selection The feature approach heavily depends on the right selection of features extracted from the code. The best metrics are those that exhibit different values for positive samples compared to background samples. While it is very unlikely that any of our selected metrics have that property on their own, we hope that a combination of them can separate the space of commits and that our ML model will pick those patterns up. Based on intuition, prior work and initial investigation we select metrics for further evaluation. Based on the reasoning behind the metric we separate them into classes described below. Furthermore, for implementation purposes, we divide our metrics into two different levels: the commit level - where metrics are calculated for the commit as a whole, either by being based on commit-specific data or by being aggregated by an external tool; and the file level - where metrics are calculated based on properties of a file and then aggregated to commit level. On the commit level we investigate class metrics:

1. **Author & committer based**
   We used the author and the committer (and the related author and committer dates) of the commit to calculate several metrics. First, we check whenever the committer is the same as the author, which might suggest the commit was merged urgently without a pull request. Next, we realized in the manual review in previous chapters that bot-produced commits play an important role in open source repositories. As such we check whenever 'bot' is present in the fields. Since established contributors play a much more important role in security-related issues [3], we compare the committer and author emails to the list of 100 top contributors in the repository. Finally, we also calculate the difference between the author and the committer date [6]. While many repositories use merge strategies that hide this difference, setting both of them to the same value, in some cases this can be an indicator showing how long a commit was waiting on a side branch before it was merged into the main branch, which we guess will be shorter for security related commits. This can be partially seen by the vertical lines in Figure 6.1 for the strapi[1] project.

2. **Delta maintainability model** [9]
   We used values calculated by the delta maintainability model as previous research has shown that security-relevant commits may hinder maintainability [41]. Furthermore, SIG has lately conducted a survey showing that poor maintainability is coupled with a higher amount of vulnerabilities [48].

3. **History based features**
   For the same reason as the difference of committer and author-date, we also analyze the preceding and the following history around a given commit. We extract such information as whether the commit is followed by a merge, how much time passed until the next merge commit was added or how many commits were added. While

[1] `https://github.com/strapi/strapi`

these metrics are used less often for commit classification tasks it has been shown that for some applications in the security field they can perform better than typical metrics [40]. These results also match our intuition that security-relevant commits are likely to be processed with higher priority. These features are calculated based on the 50 commits before and after as returned by iterating the repository, as such may not always be correct, especially in complicated, non-linear histories.

4. **Security keywords**
   For this class of features, we search for selected security keywords in the commit message. We selected a small set of simple keywords that might hint at the security nature of a commit. We separate the first line of the commit message cropped to 72 characters as the commit title for which we perform the text search separately. While specifying the security aspect in the commit message is generally discouraged [21], based on the result from the previous research question we can mine these features out for inspection and perhaps reject them later.

5. **Changed files**
   The final metric on the commit level is the number of changed files which is based on the intuition that a security-relevant commit as a fix shouldn't modify a large portion of the code based and rather focus on small changes.

On the other hand, file-level metrics are calculated for each changed file in the commits and are later agglomerated to the commit level. When calculating these metrics we can process the changes in the code itself and focus on more fundamental features of the code.

1. **Change types**
   One basic property we extract from each file is whenever the file was added, changed, removed or renamed. We expect security-relevant commits to not modify the structure of the projects and focus on fixing the issue.

2. **Change & file sizes**
   Another basic extracted property is the size of the change and the size of the file. It is generally believed that large, complex files are likely to contain vulnerabilities [13, 46]. On the other hand, we expect the majority of our security-related commits to be security fixes that are small in nature. Apart from being a feature on its own, the feature may be used by the model for scaling other values.

3. **Lizard metrics**
   Lizard [55] is a complexity analyzer integrated into PyDriller [50] which can calculate various complexity metrics such as token count, cyclomatic complexity or changed method count. This class of metrics is commonly used to measure software and predict its properties in commit classification, moreover, the complexity metrics may contribute to the complexity issue described above.

4. **Security Keywords**
   Same as with the commit message we search for security-related phrases in the patch of the commit and the whole file.

5. **Test relations**
   Similarly to search for security-related keywords we look for the keyword 'test' in the filename and in the file path to detect test files compared to production code. Whenever the commit contains tests for the issue it is fixing depends on the practices of the community managing the repository, but based on a case study on two Apache projects [35] it has been concluded that vulnerabilities are hard to test, and as such, some pattern may show up in these metrics.

6. **History based**
   Finally, similarly to commit level metrics, we investigate the history preceding and following the commit to validate how often it was changed, if it's a new file or whenever it was deleted shortly after.

Features calculated for individual files need to be combined to the commit level. This can be done in two ways: either one joins file-level features together to the commit level prior to processing them with a model or processing each file separately and agglomerating the results. For simplicity, we decided to use the first method. Depending on the type of metric, the results from files are combined in one of two ways. The first one, which is aimed at flag values (e.g. keyword 'test' in the filename), calculates the percentage of files for which the condition is true. The second one, meant for numerical values, extracts the maximum and the mean of the sequence of features. We select the mean value to represent the general nature of the commit. However, in some cases (e.g. newly added components, release files) features for some files might be extraordinarily large compared to others and thus we also consider including the maximal value.

The full list of the metrics with the initial motivation and aggregation method is included in Appendix A. The dataset of mined metrics is included in the reproduction package [52].

After the features are mined we need to select which of them to use in the models. Utilizing all of them, including those with invalid values, would lead to easily over-fitting models. First, we will validate how many data points the mining process was able to calculate the value of each feature in each ecosystem. In particular, Lizard does not support all programming languages and as such the metrics calculated by it might be empty. Second, we will manually review these features by plotting them vs the author date of the commit to spot anomalies suggesting the metric is invalid or was improperly calculated. Third, knowing that our features are correct, we will investigate their potential in a machine learning model. We manually review the largest (absolute) values in the correlation matrix for our features and remove features based on that. Furthermore, we evaluate which features have little to no variance and decide whether to remove those as well. We reject these features as they provide the least value for the classification while possibly causing the model to overfit. We repeat this for each ecosystem and the whole dataset. The final step of feature selection is recursive feature elimination (RFE) [15], where we train a random forest model on data for each ecosystem, evaluate the feature importance based on the trained model and remove the least valuable features for as long as the performance of the models is not falling off.

Models Apart from the features, the resulting performance of the model is influenced by the architecture of the model. We select 3 machine learning models commonly used in prior research for the same (or similar) task. We selected random forest classifier (RF), used in [8, 45, 26, 60, 19]; logistic regression (LR), used in [26, 19, 32]; and support vector classifier (SVC), used in [26, 60, 44]. We test the models in a number of hyper-parameters settings tailored according to recommendations and setups in prior work. Based on the F1 score we select the best-performing model, for which we further improve the performance by fine-tuning parameters. We decided to use the F1 score because it combines both the precision and the recall, which in an imbalanced class scenario are connected with a trade-off. The implementation of these models in Scikit-learn [34] can be found in the reproduction package [52]. Note that for all experiments (feature selection, model performance) metrics are evaluated on a test dataset coming from randomly selected repositories. Having the optimal parameters, we perform a 5-fold cross prediction (by rotating the test set of repositories) to get the predictions for the final recall calculation.

6.2 Results

As shown in Table 6.1, in total, over 425 thousand commits were mined with around 8 thousand being linked to vulnerability reports. However, samples that are associated with specific languages are less represented with npm having the most commits of all subsets at 561 security-related commits and 33,336 commits in total.

Table 6.1: Count of mined commits by each ecosystem

Dataset	Security Relevant commits	All commits
Whole dataset	7,986	425,891
JavaScript (npm) subset	561	33,336
Python (PyPI) subset	483	26,799
Java (Maven) subset	268	13,175

Selecting features First, we put aside commits mined from randomly selected repositories to use commits from them as the test dataset. Next, we looked into the features that do not have a set value due to processing being impossible. As predicted, in the majority of cases these were the code complexity metrics calculated by Lizard, when the language wasn't supported by the tool. The only not supported language that we decided to include in our ecosystems is TypeScript for the npm ecosystem and as such, it has a significant number of invalid values. The final percentages of rows with NaN values for each ecosystem were: 29.1% for all data points, 16.7% for npm, 0.1% for Maven and PyPI. As these weren't such a large majority we continued with them filled with zeros while being aware it may swing the correlation or the variance evaluations to the wrong side for some of these rows.

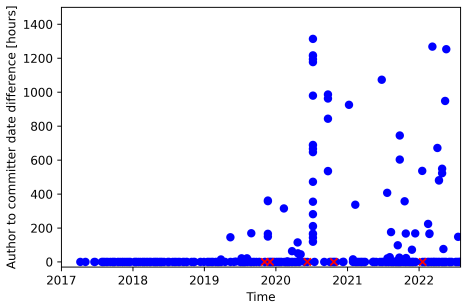
The next stage was plotting the data and manually reviewing the figures to determine if features were calculated correctly and possibly spot features that separate well the space

of metrics. For this purpose we use 2 types of plots: scatter and violin. First, the scatter plots were generated with the data points being the value of the feature and the author date of the commit. We first plotted all commits to indicate the space of possible values and their distribution and then on top of these plotted the security-relevant commits to spot what values they exhibit. In some cases, we also extract commits only for one repository and plot them separately. Some example plots for the npm ecosystem are shown in Figure 6.1. The first plot shows the difference between the author's date to the committer date - indicating how long it took for the commit to end up in its current location. Note the vertical lines are probably representing branches or releases being prepared. The second plot shows one of the history-based metrics, the average amount of times the files changed in the commit were also changed in the following 50 commits. Next, the third plot shows the average number of added lines in the commit. Note that in both these plots, the security-related commits are primarily in the low values, sadly this is also the case for the majority of background commits. The final fourth plot shows the DMM Unit Complexity metric of the commits. Even though there are more commits with the value one, it seems that security-related commits appear in both extrema of the spectrum. Note that these are only selected plots showing only the part where both security-related commits and background commits have valid values.

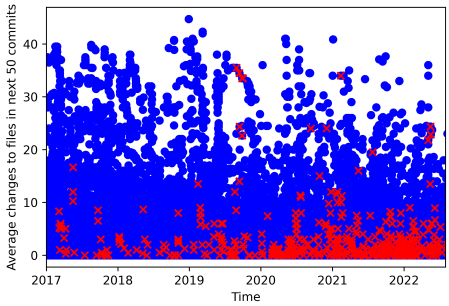
We also utilized violin plots, which were generated for the population of all commits (blue) and the population of security-related commits (red). They represent what values the different population exhibit and can be used to easily spot the features that have the potential of discriminating security commits. Example plots for the PyPI ecosystem are shown in Figure 6.2. In the first plot, we see the distribution of the feature values for the number of changed files in the commits. These distributions look very alike, with both the mean and median with similar values. Next, the second plot depicts the macro average of the number of parameters in the changed method of the file. The distributions are a bit different, with security commits being more likely to change methods with larger counts of parameters. The third plot shows the distribution of the feature representing the percentage of files that contained the 'secur' substring in the patch. While the median for both classes is 0, more security-related commits exhibit higher values for this feature. Finally, we also plot the DMM Unit Size for the PyPI ecosystem. It shows that the intermediate values are more thinned out because there are more data points in the background dataset. Using these plots we also rejected some metrics that had no variance or that were not disambiguating security commits as they were designed to.

Having validated our metrics we move to feature selection. In variance analysis, we see that a significant number of keyword-based features show little to no variance. This was expected as we designed these metrics to pick out primarily security-related commits which are in a substantial minority. As such we use the variance filtering with a very low threshold of 0.001 to remove just the features that show pretty much no variance at all. Almost all keyword-based features calculated from method names had to be removed in this step, as the selected keywords rarely appeared in the method names.

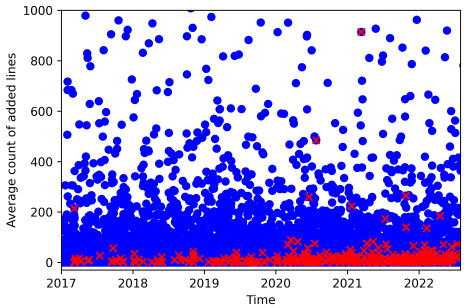
Next, we remove features based on their correlation with other features. As many of our metrics were extracted from the same data and we were aggregating file-level metrics



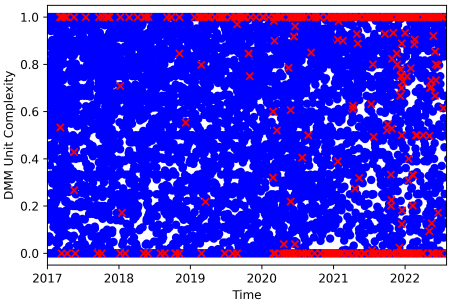
(a) Committer to Author date difference for strapi/strapi. Note the vertical line represent changes waiting to be merged in.



(b) The average number of times files changed in the commit were also changed in the following 50 commits.



(c) Average added lines count.

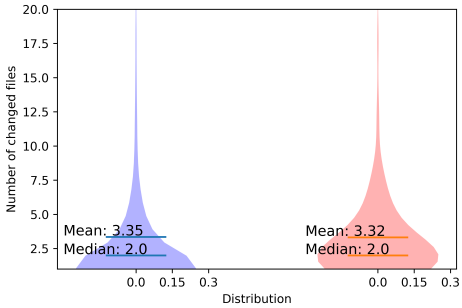


(d) The unit complexity metric from the Delta Maintainability Model.

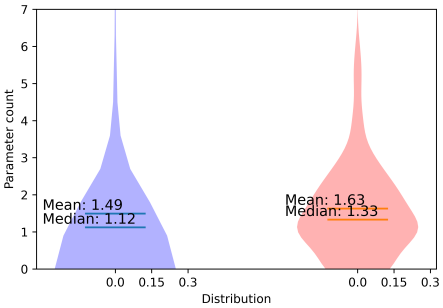
Figure 6.1: Plots of example metrics over time for commits containing JavaScript and JavaScript-like languages. Red points indicate security-related commits.

into two values at the commit level it was expected to find a lot of correlated metrics. We extracted pairs of features for which the absolute value of their correlation was above 0.75, and review them manually. In particular, features based on the title of the commit were highly correlated to the features based on the message of the commit suggesting that in the majority of cases the commit message was short (i.e. just the commit title) or contained no security-related information in the latter lines. Similarly, highly correlated were the features correlated to the total complexity of the commit files as large files usually imply high complexity and high token count. We see that including both the average and the maximum value in the aggregation from the file level was in many cases excessive as these values are frequently correlated. However, as in some cases when we had motivation to add both, we left them in for the RFE process to decide their usefulness.

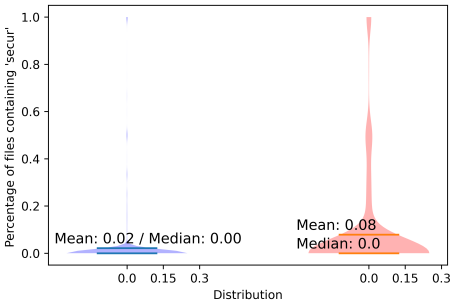
In the next step, we use RFE with cross-validation to eliminate the least important features. In the elimination, we use the random forest classifier as its training process was the fastest. We set up the process to yield a minimum of 40 features. After the feature elimination 100, 73, 51, and 80 features were left for the dataset of all data points, npm,



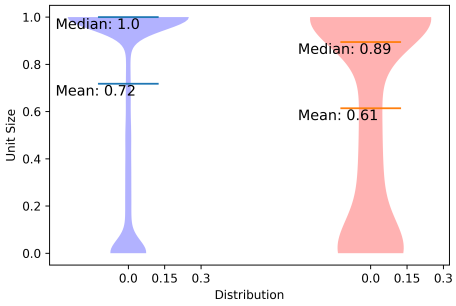
(a) Example of a feature with a similar distribution – the number of changed files in the commit.



(b) Macro average (average of averages) of parameter count in changed methods in files changed in commits.



(c) The percentage of files in each commit containing the 'secur' string in the patch.



(d) DMM Unit Size

Figure 6.2: Violin plots used to compare the distribution of different metrics for commits containing Python code. The left (blue) distribution shows the overall population while the right (red) one shows just the security-related commits.

PyPI and Maven. While we expected the entire dataset to require the most features, we were not expecting such large differences in the ecosystem datasets. This might suggest that the commits mined in PyPI ecosystems span a smaller space than commits from other ecosystems. Investigating the features, kept by the RFE process we find that features from all classes were kept. In particular, many of the history-based features were favored by the selection process. Additionally, many of the code metrics were also kept even though some represented similar properties of code. Lastly, as expected due to the long-tail nature of the data, in many cases both the mean and the maximum aggregation were among the chosen features.

At this point, we can answer the first research question of this experiment. Visually no metric was able to separate the security-relevant commits from the background. There are some patterns in the features, for example, most of these commits are small, or commits are primarily adding lines rather than removing them. However, in many cases, these patterns

match the patterns of the background. When looking at feature importance we see that only in one case the feature importance crossed the 0.05 mark – the feature *'secur' in commit message* for the npm ecosystem. All other features were around or below 0.035. On the other hand, many of the top features are repeated in different ecosystems suggesting the similarity of the patterns. While most of these were history-based, others included *'secur' in commit message*, *maximal file size in commit*, and *maximal number of lines of code in changed methods*.

Precision Models were trained with F1-score as the scoring function. As seen by the results in Table 6.2, the results are quite underwhelming. The biggest issue, limiting the performance of these models was the highly imbalanced classes. The models were trained in over-sampling and under-sampling ratios of 1 to 8, where lower values were usually preferred. This ratio is significantly different from the ratio in the test datasets which was closer to 1:50. Using such a ratio in the training process would make the model ignore the positive samples in the training process. On the other hand, oversampling causes easy overfitting or poor generalization of the models where the data points are repeated. Nonetheless, the results leave much to ask for, as the highest F1-score is at 0.38.

The performance was significantly lower for the entire dataset and the Java subset compared to JavaScript and Python subsets. We believe that the driving factor in this performance was the size of the datasets and how uniform the practices in the languages in the dataset are. For JavaScript, there was the largest dataset available for specific ecosystems and the best results were achieved. Similarly, it was followed by Python, for which less data was available, but perhaps since Python code is known for a very uniform coding style it also yielded good performances. The opposite applies to the entire dataset where without any filtering all practices were mixed making the classification much more difficult. The worst performance was achieved for the Java subset where there was little data.

Focusing just on the precision of the best-performing models we achieved 0.16 precision for the entire dataset and 0.54, 0.32, and 0.20 for JavaScript, Python, and Java respectively. Note, a precision of 0.16 means that within the set of commits classified by the model as security-related the class ratio is about 1:6.25 which represents an eight times increase in the prevalence of the security-related commits compared to the base set.

Table 6.2: Precision, recall and the F1-score of the best performing models in searching for optimal class ratios.

Model	Whole dataset	JavaScript subset	Python subset	Java subset
RF	0.13, 0.30, 0.18	0.54, 0.29, 0.38	0.50, 0.22, 0.30	0.20, 0.06, 0.09
SVC	0.15, 0.19, 0.17	0.19, 0.31, 0.24	0.33, 0.31, 0.32	0.10, 0.11, 0.11
LR	0.16, 0.08, 0.11	0.11, 0.31, 0.16	0.30, 0.28, 0.29	0.03, 0.11, 0.05

Recall To calculate the recall of the method, cross-fold predictions were utilized with 5 folds. To keep the results generalizable, the splits were done on the repository level which could have created uneven splits that hurt the final prediction accuracy. We calculate the

recall based on the number of vulnerabilities for which we had the prediction for at least one commit. As such the recall might be different from the random trials made in the previous sections. Furthermore, the result is affected by vulnerabilities being represented by multiple commits and commits representing multiple vulnerabilities. The results are shown in Table 6.3.

We see that the highest recall is achieved for the JavaScript ecosystem which lines up with the results above. We also see that the ecosystem-wise classification performed better than the overall one. The recall calculated for all vulnerabilities is 12.1% which is sadly lower than what was expected from the performance metrics in the random trials.

Table 6.3: Recall of the feature-based commit classification approach on vulnerability level.

Dataset	0 days prior	7 days prior
Whole dataset	12.1%	7.7%
npm subset	28.0%	13.5%
PyPI subset	10.7%	5.5%
Maven subset	17.6%	11.5%

Median delay The median delay is evaluated based on the same data as the recall, but instead of counting the spotted vulnerabilities with a delay of larger than 0 or 7 days, we calculate the median of the delays. The results are shown in Table 6.4. The median delay is smaller than the median delay calculated for the first commit which was calculated in Chapter 3. This suggests that the models are classifying the commits closer to the actual disclosure. This might be affected that these commits were done with fixing the vulnerability in mind, while the older references are 'accidental' fixes.

Moreover, we see that the median delay is highest for the Java ecosystem, but as there is limited data available the results might be unreliable. The delays of both JavaScript and Python were below the overall median of 10.8 days.

Table 6.4: Median time gained by spotting security-related commits with feature-based commit classification

Dataset	Found vulnerabilities	Median delay
Whole dataset	1,468	10.8
npm subset	89	4.7
PyPI subset	188	7.0
Maven subset	40	17.0

Applicability The method has high applicability as there are no significant limitations. Theoretically, the method can be run on commits mined from any repository, however, some restrictions may apply. First, the programming language of the project must be supported

by Lizard to utilize many of the Delta Maintainability Model metrics as well as many of the code metrics. Furthermore, some projects may clear the committer or author-date depending on the git workflow. Similarly, the project maintainer may hide the security aspect of the commit from the message or the comments as it's suggested by MITRE.

6.3 Summary

Based on the results of this experiment we conclude that commit metrics are not enough to pick out security-relevant commits from their background. Metrics of security-related commits follow the general trends of the commits population. When applying machine learning approaches, more complex patterns across metrics can be found, but they're too weak to result in good separation.

While the results of the method are not that astounding, one has to remember that it has been evaluated in harsh conditions and where no apparent patterns were seen in the manual review. The major obstacle for the method was the high imbalance of classes as well as inconsistent patterns, which can be seen by the overall poor performance of the models trained on the entire dataset.

- The precision of the method varies from 13% to 54% depending on the model, test run, and the ecosystem. The highest scores were achieved for the npm ecosystem for which there was the most data.
- The recall at the disclosure date of the method varies from the lowest of 10.7% for the Python ecosystem to the highest of 28% for the JavaScript ecosystem.
- The applicability of the method is high as it can be applied to any repository and the commits will always be public. Some features, however, may not be available to the model depending on the ecosystem (e.g. code analysis may not be supported) or project culture (e.g. using merge strategies that modify history).
- The median delay of the method is almost 11 days, which means it spots commits that are pushed to the repository generally 11 days before the disclosure of the vulnerability.

Chapter 7

Validating deep learning approach

Finally in this chapter, having explored the field and validated simpler approaches, we move to apply deep learning models to the problem. In particular, we are testing the performance of different sampling methods, training our model inspired by Commit2Vec [5] and by VulFixMiner [59], and lastly evaluating the performance of deep learning models on our datasets.

Research Question 4

4.1) What impact on the model performance have different sampling methods?
4.2) With what performance can a bilingual deep learning model spot security-related activity before disclosure?

Background One of the biggest challenges in deep learning approaches is sampling the code for the model. In the majority of cases, a machine learning model can only accept inputs of a set length which is usually not big enough to handle entire files or changes. As such code from these files needs to be sampled in a way that extracts as much information on the code and code structure as possible. Many approaches for sampling are inspired by techniques used in natural language (NL) processing.

Once converted to a sample, text data needs to be embedded into numerical values. Early approaches create non-distributed embedding, meaning that samples are converted into vectors where each token is mapped to one or a small group of output elements in the vector. One of the simplest approaches is creating a map of tokens assigning each unique token an index. These approaches may struggle in big vocabularies or when the same token is used in a variety of contexts.

On the contrary, distributed approaches represent samples as vectors (or matrices) in such a way that each token is distributed to many (if not all) of the output elements of the vector. These representations have proven to work especially well with deep learning algorithms which can capitalize on a more complete representation of the samples and pick up more subtle patterns. One of the most popular implementations of this approach is Code2Seq [1] and Code2Vec [2].

Bilingual models operate on both natural language and programming language (PL)

inputs and usually produce distributed embeddings. As much of the code is accompanied by some sort of a description - either an issue, commit message or just some comments, these models have the potential to utilize even more data. Recently they have gained a lot of attention due to the release of CodeBERT [11] and GraphCodeBERT [14], two pre-trained RoBERTa [30] models on a large dataset of NL and PL pairs.

Finally, in the case of commit classification, the representation has to be on the commit level and not on the sample-level. While there are other solutions for example those based around graph neural networks [62, 54], the dominant trend is to utilize encoding of samples produced for different files in the commit and aggregate them together into the commit representation. This approach was suggested and evaluated in Commit2Vec [5] reaching 0.72 F1-score in a balanced set scenario.

7.1 Experimental setup

The experiment to answer this research question consists of 4 different stages.

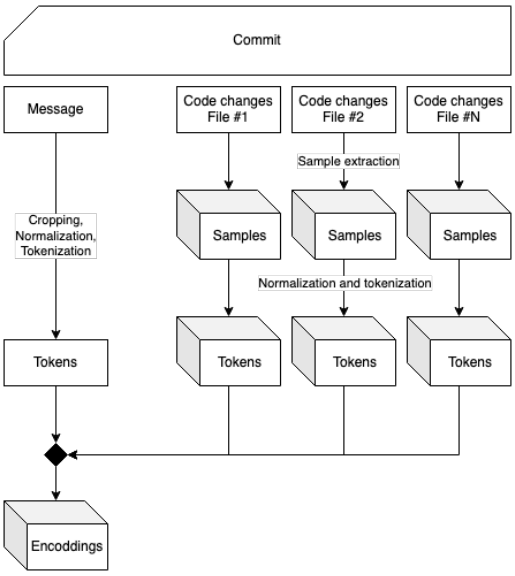


Figure 7.1: Extraction of samples from commits

Stage 1: Mining and sampling The first stage is to obtain the data for training our models. Similarly to the previous research question, we will utilize the security-related commits mined from the vulnerability reports to train the models. As shown in Figure 7.2 GraphCodeBERT model expects the input to be prepared in a specific way. The first token of the input should be the CLS token which can be followed by the natural language input for which we utilize the commit title of the commit. It is then followed by a separator token and the programming language input. If required, additional separator and padding tokens are appended. The sampling method of the programming language input from code can have

an impact on the model performance [5]. The sampling method has to represent the nature of the change, which in our case is preserving the security-related aspect of the sample within the commit. Furthermore, it should put the code change in a wider context of the file. We implement 4 different sampling methods described below inspired by practices used in other research, 2 of which to our knowledge weren't applied in security-related commit classification tasks yet.

Added code The simplest of proposed methods for code sampling is just using the added code for each file in the commit. This will yield as many samples as there were files changed in the commit. Because the transformer has a fixed input size of 512 and beside the code sample, the commit message is also included, and the code might get cropped. However, at least 438 tokens (512 minus CLS, SEP tokens and at most 72 NL tokens) from the code will be added to the input. As simple as it is, it was utilized in VulFixMiner [59] yielding promising results.

Rolling window A modified version of the added code sampling method is the rolling window method. Since added code may be cropped and contain only the beginning of the change, it might contain a lot of boilerplate code at the file start. Also, the added code method in no way represents the size of the change. We suspect that added code may provide too little context for small changes. To counteract these flaws, in the rolling window sampling method we randomly choose 100 lines added in the commit and for each, we sample the line together with up to 10 preceding and following lines. This way commits that have less than 100 changed lines will have fewer inputs and will contain more context. Files that have relatively more changes than other files will be more represented and by nature of random choosing more samples out of the 100.

AST Paths Many approaches in literature have been proposed that parse the code into the abstract syntax tree (AST). An AST represents how the expressions are contained within each other in the file structure, and as such it's suspected that extracting samples from this structure captures more context. We implement a similar sampling technique as used in Commit2Vec [5]. In the AST we randomly choose 2 nodes and find a path between them. Then we collect all lines where the nodes on the path start. This procedure is repeated 100 times. Looking for the paths between nodes can represent how related the changed lines were – a long input means a long path between the changed lines and a spread-out change in the file. We use the start lines instead of the tokens to hopefully utilize the pre-trained information on code structure contained in GraphCodeBERT. This sampling method will also include information on the relative change sizes in files as in the previous method.

Edit scripts The edit script based sampling method is a modification of the added code sampling method. The difference is that instead of relying on changed lines as described by Git, we compare the AST trees of the file before and after. Using the GumTreeDiff tool [10] and the Chawathe algorithm [7] we generate an edit script that represents the added and removed AST sub-trees between the two versions of the file. Then each added sub-tree is mapped to its lines within the file and the lines are

extracted as a sample. This way, the sampling method is not relying on just the line numbers in a text file but involves actual code understanding in extracting the samples. Furthermore, we sample each sub-tree separately (which could be compared to using each added hunk of the commit), to avoid cropping the data.

To determine the best sampling methods all of them will be evaluated for a selected programming language and the final performance will be compared. Sampling methods were implemented in Python and Java using PyDriller [50] and GumTreeDiff [10] tools.

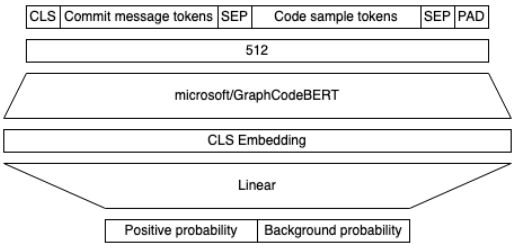
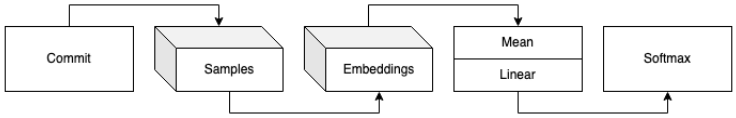


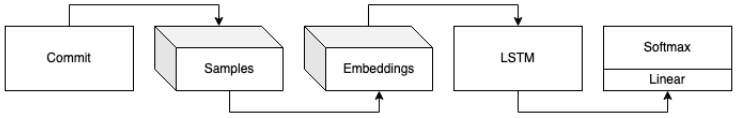
Figure 7.2: Fine tuning model

Stage 2: Fine-Tuning The second stage of the processing is the fine-tuning of the GraphCodeBERT model [14]. Before one can use the model for the downstream task (of security-related classification) it has to be fine-tuned to the vocabulary and the gradients in the data. For the fine-tuning process, we follow the approach of VulFixMiner – we connect the output of the CLS token to a linear layer which combines the output to two single values representing the probabilities if the sample is positive or if the sample is a background sample. The model is trained using cross-entropy as the loss function. In this process we are training the model to produce such embeddings for which exist two linear combinations: for positive samples, one maximizing the first probability and one minimalizing the second; and for background samples the inverse.

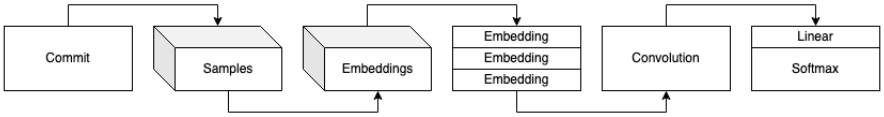
Using a pre-trained model and only fine-tuning it also means that we don't have that many hyper-parameters to optimize. In our preliminary experiments, we found 3 parameters that had the most influence on the model performance. First, the class ratio - the proportion of positive samples to background ones in the training phase impacts both precision and recall. We control it by under-sampling background samples to a specific ratio. Second, the over-sampling ratio, i.e. the number of times the positive samples are repeated, also impacted the precision on the test subset. We control it by over-sampling the data points - providing them multiple times to the model. Third, the learning rate recommended for the fine-tuning process is 2e-5, however, we found that for some sampling methods this value was too large. As such we test out models trained with a learning rate an order of magnitude lower (2e-6). To find the optimal setup of hyper-parameters we perform an exhaustive grid search on a subset of 5,000 samples randomly chosen from the appropriately prepared dataset.



(a) Mean Aggregator



(b) LSTM Aggregator



(c) Convolution Aggregator

Figure 7.3: The data flows within the proposed aggregator models.

Stage 3: Training Aggregators Up until this point, all processing is happening on the sample level which are generated for given files or lines in the commit. These on their own may not represent the whole story and to get the most out of the information we aggregate all embeddings from the sample level to the commit level. For this task, we propose 3 methods used in related work or commonly used to concatenate a series of embeddings.

Mean Aggregator The simplest of the aggregators was the one applied by VulFixMiner. For each element of the 768-long embedding vector, the mean value is calculated across all samples for the commit. Then, the means are then contracted with a linear layer to two values representing the probabilities for the commit.

LSTM Aggregator Long-short term memory (LSTM) networks are one of the most popular networks used to process sequences. Used for this task for example in Commit2Vec, they usually outperform alternatives. Based on recommended settings, in the experiments, we use an LSTM with 5 hidden layers and a dropout of 0.2. We found this configuration of hyper-parameters to offer enough parameters to capture the gradients and not overfit thanks to the dropout. The output of the LSTM is contracted to two values as in the previous case.

Convolution Aggregator Inspired by their use in image processing tasks, convolution networks have the property of mixing and compacting the input. We built a small convolution network from two sets of Convolution, MaxPolling, and ReLU layers. The output is then flattened to a single-dimensional vector. As before, their result is subjected to dropout (p=0.3) and then it's combined into 2 values by a linear layer.

These aggregators will be trained on the embedding vectors produced by the fine-tuned GraphCodeBERT transformers. The goal of the aggregator is to pick up on patterns within the embedding and focus on the samples that determine the commit as security-related or not. As such they should improve on the results of the individual samples. The final performance should be evaluated with cross-validation to reduce the impact of randomness. For the performance metrics, similar to the case of feature-based models, we have selected the F1-score as the main indicator as our dataset is heavily imbalanced and the precision and recall in case of similar F1-scores with more priority on the recall.

Note, that all performance testing is done by splitting the dataset according to the repositories to ensure no data leakage between the train and test sets. Due to the repetition of the commit message in all samples from one commit, the splits cannot be made on the sample level, as the commit message part could have been repeated both in the training and testing sets. Furthermore, the training and test split have to remain constant across all stages, i.e. the training data points of the fine-tuning phase cannot be used in the test set of the aggregator phase as it also produces unreliable results.

Stage 4: Evaluation In the final stage of this experiment, we evaluate the deep learning approach as a method of prealerting for vulnerabilities. As in the previous chapter, we use cross-fold validation to calculate the prediction of commits linked to vulnerabilities and evaluate how fast they would have been spotted. This validation can also be done with the models of VulCurator [31] which include a commit message classifier and the VulFixMiner [59] model trained on the SAP-KB dataset [37].

7.2 Results

In total 171,628 commits were mined, where 8436 were positive commits. The most data is available for the JavaScript language ecosystem, where 23,495 (744 security-related) commits were mined. Table 7.1 shows the detailed numbers of commits and samples mined for each subset. The sample columns indicate the total amount of mined data from all methods. Regarding the individual sampling methods, the least data is available in the edit script based sampling method where the programming language needed to be supported by the parser and the parsing had to succeed for both before and after versions of the file.

Table 7.1: Counts of mined data for each ecosystem.

Dataset	Total commits	Positive commits	Total samples	Positive samples
Whole dataset	171, 628	8,436	12,487,465	729,211
JavaScript subset	23,495	732	1,931,436	61,096
Python subset	13,489	594	1,713,725	69,081
Java subset	6,712	313	1,037,158	43,868

Fine-tuning and sampling methods After mining the code samples we started working on finding the optimal setting for fine-tuning the GraphCodeBERT transformer as our embedding producer. In this step, we limited the number of training samples to 5,000 and validated different dataset rebalancing methods – oversampling for increasing the number of training samples, and undersampling for controlling the prevalence of positive samples in the training. Additionally, as the model was overfitting in the initial experiments, we tested a lower learning rate. The values were calculated for the samples from the Java ecosystem as it was the only ecosystem for which we had inputs from all sampling methods in the first run.

We find that multiple configurations yielded significantly worse results than others, but for each sampling method, there was a group of hyper-parameters that yielded the best results. Then, we realized that in one case the rebalancing parameters of oversampling and class ratio yielded the best (or similar to the best) results for all sample types. As such we selected the class ratio (CR) equal to 2 and the oversampling ratio (OR) to 4. The recommended learning rate (LR) was preferred in all cases, but for the rolling window method. The results are shown in Table 7.2. Note, that the class ratio in the test dataset was similar to the class ratio in the feature-based experiment – approximately 1:50.

From these results, we can see that the sampling methods achieve comparable performance in almost all cases. Only the edit script method achieved worse performance. This came as a surprise since the method requires the most computing to sample the data out of code (as two files need to be parsed and their trees need to be compared). We believe that this lower performance compared to other methods is caused by small trees returned by the tree comparer, which generate small samples that can be repeated both in the positive and in the negative dataset. One example of such a sample is the closing bracket added together with an 'if' statement, which in this method is converted to a standalone sample.

Table 7.2: Results of fine-tuning for samples with Java code.

	Optimal parameters			Achieved scores		
Sampling method	CL	OR	LR	Pr	Rc	F1
Added code	2	4	2e-5	0.30	0.25	0.27
Rolling window	2	4	2e-6	0.22	0.34	0.27
AST based	2	4	2e-5	0.36	0.24	0.29
Edit script based	2	4	2e-5	0.29	0.17	0.21

On the other side of the spectrum, the AST paths based sampling method performed slightly better than other methods, which is in line with the findings of the Commit2Vec model. Note, that there is not that big of a difference between the scores of other methods. In our opinion, this suggests that the transformer architecture has enough parameters and can pick out the patterns regardless of the sampling method. The sampling method has proven to matter in cases of other, simpler models, but when working with more complex models results show that the extra effort in choosing samples might not be worth it.

Finally, reflecting on the actual classification results, the sample-level predictions are

quite underwhelming. While they are better than the results of the feature-based commit classification, considering the much more complex mining and used architecture one would expect these values to be better. Furthermore, these results are significantly lower than those reported in the literature for the final models. However, we suspect that in their cases the sample level results also are lower than the final commit level performance, as not all samples from the security-related commits might represent the security-related part of the commit. As such, similar samples might be present in both the positive and background dataset. Sadly, we also suspect that due to limited data filtration, the model might be picking up on non-important patterns that have nothing to do with the security aspect of the change.

To sum up the answer to the first sub-question, some sampling methods may produce input that inhibits the performance of the embedding transformer however there is no significant improvement between 'good' sampling methods.

Aggregator results Next, aggregator architectures were tested on the produced embeddings. First, we proceeded to test the models on filtered samples with only Java code. The LSTM architecture was the most consistent model regarding finding a solution where it would learn the patterns within the data without assigning all commits to one class. The LSTM aggregator was however failing when not enough data was provided. In low-data scenarios, the mean aggregator performed the best. The most probable reason for its superiority in these scenarios was a lower number of parameters that were not overfitting to the specific commits in the training set. The convolution aggregator performed the worst. This model had very few parameters, less than the mean aggregator, which count might have been too low to capture the patterns produced by the transformer. Additionally, the variable input size meant that padding to the 100 embeddings was required, which probably impacted the convolution layers. All models had the tendency to be stuck classifying all commits to one class in which case the training was restarted.

In these tests, we realized some pitfalls of our approach. First, training on the samples with only Java code restricts the data too much and the models seem to be learning project-specific patterns rather than any security-related aspect of the commit. Increasing the dropout within the models helped, but did not solve the issue entirely. Furthermore, the large computational overhead of using deep learning and the long-running task of embedding the input samples using the transformers meant our datasets were much smaller reducing the credibility of the results. Time consumption was increased even further when the models were evaluated using 5-fold cross-validation.

To combat these issues we simplified our setup. First, we moved our domain to all security-related commits without limiting ourselves to one ecosystem. This helped with overfitting to patterns within specific projects as more variety was introduced. This however meant that the training data was more general, possibly hiding some patterns and that there was more variance in the test set lowering measured performance. Second, to reduce the time needed to embed the input, we have reduced the ratio of security-related commits to background commits to only 1:5, abandoning the evaluated class ratios evaluated above. This meant that the ratio of classes on the sample level was partially determined by the amounts of samples in security-related and background commits pushing this ratio to values of 10 to 15, as positive commits were usually smaller than the background. The mean

and LSTM aggregators were trained with this class ratio of commits kept constant at 1:5, but with the same ratio the convolution models were always classifying the commits as background and we needed to drop the ratio to only 1:2.

With these modifications, we evaluated the aggregators again for the final results depicted in Table 7.3. These results have to be interpreted with the evaluation setup in mind as the boosted ratio is quite unrealistic. Similarly to the results on the Java subset, the mean and LSTM aggregators reached similar performance, while the convolution aggregator performed much worse. A surprising result is the lower performance of the aggregators on the input mined using the AST paths sampling method. Lower performance was already seen in the fine-tuning process. Seeing that the convolution model performed the best on this data, we suspect that this result is due to the aggregator models failing to converge in finding the optimal solution in the limited runs we performed. To eliminate this uncertainty, we recommend performing cross-validation or trying different splits into training and testing data.

We also see that the convolution aggregator failed to find a good solution when working on the samples generated with the added code method. This is understandable as there were usually only a few samples per commit and the inputs to the aggregator were mostly empty.

Table 7.3: Aggregator as evaluated on subsets of data.

	Added code sampling			Rolling window sampling		
Aggregator	Precision	Recall	F1 score	Precision	Recall	F1 score
Convolution	0.03	0.01	0.02	0.08	0.60	0.15
LSTM	0.52	0.56	0.54	0.51	0.51	0.51
Mean	0.60	0.43	0.50	0.55	0.44	0.49
	AST paths sampling			Edit scripts sampling		
Aggregator	Precision	Recall	F1 score	Precision	Recall	F1 score
Convolution	0.15	0.34	0.20	0.16	0.15	0.16
LSTM	0.57	0.27	0.37	0.48	0.34	0.40
Mean	0.55	0.31	0.40	0.55	0.28	0.37

The results are promising as in multiple cases both the precision and recall reach values above 0.5. However, as these results are calculated on random trials and one-off subsets, we cannot rely on them for further evaluation. Unfortunately, calculating them with cross-validation and with a better class ratio proved to be outside of our resources. As such, to make the following results more reliable, we continue our investigation using existing models from the VulCurator [31] project – a commit (code) classifier and a commit message classifier.

Evaluation results Seeing that we use a 'third-party' model the first step of the evaluation was to exclude commits that were used in the training of the model. We found that there were 41 commits in our dataset that are also in the SAP-KB dataset. We excluded these

from further analysis. We also repeated the experiments presented by the researchers to validate our setup. Having validated the models, we used them to classify 6624 security-related commits, together with randomly sampled background commits so that the positive commits amount to 10% of all commits in the dataset.

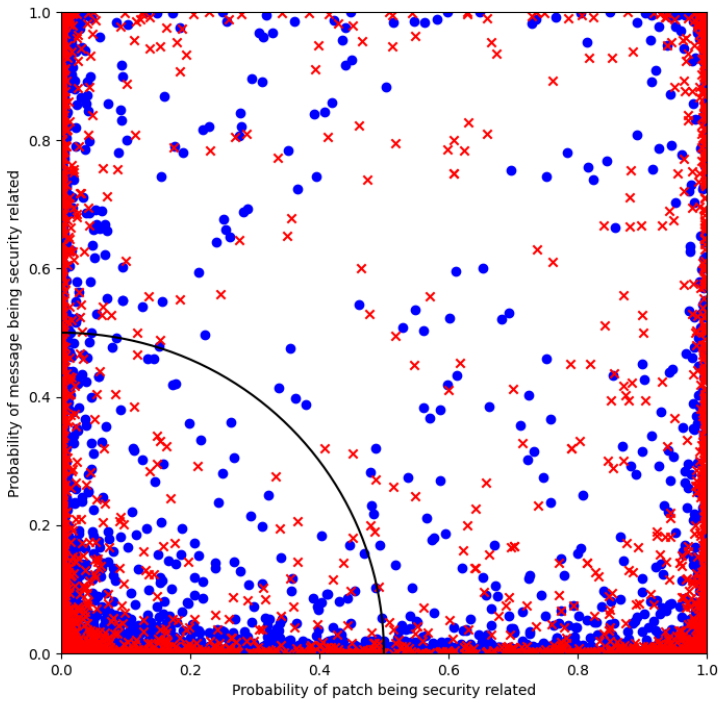


Figure 7.4: Commits classified by the VulCurator models. Red data points indicate security-related commits while the blue ones indicate the background.

As can be seen in Figure 7.4 models classify the commits usually into the extremes – the majority of commits are classified as very likely being security-related or background and the probability is rarely around 50%. As both the commit message and the patch of the commit can indicate the change is security-related we have chosen a decision boundary that compromises them both. For the commit to be security-related, it needs to be 0.5 away from the 0/0 point which is depicted in the figure with the quarter-circle. The numerical results are shown in Table 7.4. In general, the precision of the models is at 0.38 and the recall is 0.34 resulting in an F1 score of 0.36. The precision of 0.38 means that in the set of commits classified as security-relevant the ratio of true positives to false positives is around 1:2.6 which represents 3.8 times higher prevalence compared to the base set. We can also see that the results are slightly larger for the Java and Python programming languages as those are represented in the SAP-KB dataset. While not depicted in the Figure, we also found that the patch classifier performed better than the message classifier.

Sadly, the results aren't as high as the results on the SAP-KB dataset. We believe there are a few possible explanations. First, the evaluation setup in our experiment is harsher as the ratio of security-related commits was lowered from 1:5 to 1:9. Second, our dataset might be noisier as some commits linked to issues linked in vulnerabilities might not be security-related. Finally, this drop might not be about the experimental setup, but about the nature of the problem and poor generalizability and transfer learning potential. The model trained on the SAP-KB dataset might have learned the pattern of vulnerabilities within projects in the dataset and ways the community of these projects solves security issues. Once the model is evaluated on another dataset the performance drops as these patterns are no longer visible.

Table 7.4: Precision, recall, and the F1 score for the commits classified with VulCurator models assuming the decision boundary as depicted in Figure 7.4.

Dataset	Precision	Recall	F1 score
Whole dataset	0.38	0.34	0.36
JavaScript (npm) subset	0.30	0.34	0.32
Python (PyPI) subset	0.40	0.43	0.41
Java (Maven) subset	0.36	0.47	0.41

Recall Next, we move to evaluate the recall on the vulnerability level. As in our dataset, multiple commits are connected to the vulnerabilities, sometimes the model has multiple occasions to spot the same vulnerability. Furthermore, spotting a commit linked to many vulnerabilities might change the recall more than spotting one connected to only one. As seen by the results shown in Table 7.5, we see that the overall recall drops to 30% (from 34% at the commit level). This is possible in the case when the model was correctly classifying security-related commits under the same vulnerability. We also see that the recall in the Maven ecosystem is much higher compared to others. While this might be the result of the fact that VulCurator models were trained on a dataset focused on this language, there are only 143 vulnerabilities spotted with this method, increasing the uncertainty of the result. Similarly to before, there are around 10 percentage points when moving from vulnerabilities spotted just before the disclosure and a week before.

For comparison purposes, we also calculated other popular metrics used in the security-related commit classification. First, we use the area under the precision-recall curve, which is a threshold independent metric. In this case, threshold independence means that we don't select the decision boundary at which the model classifies commits as security relevant. Instead, we allow the threshold to vary and calculate the precision and recall for each value. We calculate this value at 0.33. Second, assuming the output of the models is to be reviewed by a human, we use an effort-aware metric. Similarly to other projects, we have chosen the recall at 5% of reviewed lines of code, which comes out to be 0.32. This result was expected based on the visualization of the classification results where we saw the commits being pushed to the side of the plot. Seeing both these values are similar and in line with the overall precision and recall, we conclude that there are no distinctive patterns in the results of the classification.

Table 7.5: Recall of the DL models on the vulnerability level.

Dataset	0 days prior	7 days prior
Whole dataset	30.0%	20.2%
JavaScript (npm) subset	37.0%	22.7%
Python (PyPI) subset	30.9%	18.6%
Java (Maven) subset	61.2%	52.8%

Median time gained Directly connected to the recall of the method, the median time gained describes how much sooner the vulnerabilities are spotted compared to the disclosure. What we find is that in general, the median time gained is 15.2 days, similar to the value of the first commit linked to the vulnerability. The median time gained is significantly larger for the vulnerabilities in the Maven ecosystem, however in that case the median time gained in spotting the first commit was also almost 50 days. The situation is different for npm and PyPI vulnerabilities where the median time gained is lower than the overall median time of the first commit. This suggests that for these ecosystems the model might not be spotting the first possible commit and instead finding commits later in the fixing process or those that are done with a higher priority.

Table 7.6: Median time gained for the DL-based prealerting method.

Dataset	Found vulnerabilities	Median delay [days]
Whole dataset	3,433	15.2
JavaScript (npm) subset	189	8.8
Python (PyPI) subset	345	9.7
Java (Maven) subset	143	55.2

Applicability The last factor to discuss is the applicability of the method. As already discussed the methods are obviously limited only to commits, which might appear late in the coordination process. This can be however seen as an advantage as the commits are guaranteed to appear while issues might be kept confidential. As such, we determine that the method has high applicability. However, the method performs much better if enough data was generated for the project.

7.3 Summary

In this chapter, We have validated four different sampling methods to extract code to be processed by the GraphCodeBERT transformer. We see that the worst-performing sampling method, based on the edit script, is 28% worse, F1 score-wise, than the best-performing method based on the AST path. This suggests that choosing a bad sampling method may significantly impact the model's performance. On the other hand, judging by all other sam-

pling methods resulting in a similar F1 score, we think that because of the complex architecture, the transformer model can pick out a similar amount of information regardless of the sampling method.

As such we believe that the sample level performance is primarily impacted by the curated data and whenever the used samples actually use the tokens specifying the security-related nature of the code. This in turn impacts the performance of the entire method. A major drawback to the method is long training times and dependence on a large dataset of training data. We were able to train and test our models only on subsets of data achieving an F1 score of 0.54 but to keep our results reliable the DL method was evaluated using VulCurator models. The results hint that the models may not generalize across repositories. The methods' performance metrics are:

- The precision of the method on the entire dataset is 0.38 with values for ecosystem subsets varying from 0.30 to 0.40. The model works best on a subset in which the programming language was represented in the training set.
- The recall at the disclosure date of the method is 0.30 with values for ecosystems being 0.37, 0.31, and 0.61 days for JavaScript, Python, and Java subsets respectively. The larger value for the Java subset might be explained by a relatively low number of found vulnerabilities with references to objects with this language.
- The applicability of the method is high as it relies on the commits, which are public from the nature of the open source. However, the models work best in the projects and the ecosystems that were trained on. Additionally, if a complex sampling method is used an adequate parser is required for the analyzed programming language.
- The median delay time is 15.2 days with values for ecosystems being 8.8, 9.7, and 55.2 days for JavaScript, Python, and Java subsets respectively. As for the recall, we believe the larger value for Java is an artifact due to the lower amount of data.

Chapter 8

Investigating the recall

In this chapter, we take a step back and look back at the problem of vulnerability prealerting. In particular, we want to evaluate how the proposed methods behave in specific classes of projects. First, this will give us more insights into the performance of the methods and confirm or deny their effectiveness in more specific scenarios. Second, this evaluation will provide us with more information on the nature of the security-related activity and its patterns across different projects.

8.1 Property selection

As we focus on the prealerting aspect we will be looking into the recall of the previously evaluated methods. We have chosen recall on the day of the disclosure as there was less data available for the recall a week before disclosure. Previously, in the process of designing the methods, we were splitting the data into sets according to their ecosystem or programming languages. This approach is fairly common as projects written in one ecosystem rarely depend on projects from another one. However, this split can hide insights on the recall of the methods, for example, a particularly bad case would be if a method was particularly poor at spotting critical vulnerabilities and would make up for it by spotting low-severity ones. To get more data in each of the classes we don't split the dataset into ecosystems and use the entire dataset. We've selected four aspects that we believe may influence the ability to spot the connection of the commit to the security aspect and may be interesting to investigate.

The year of the disclosure As the engineering and security fields are dynamically changing, we validate the methods' recall over time. We check how the different methods perform when spotting security-related commits in the different years since 2017. The year of the disclosure might affect the method's performance in many ways. First, over the years security practices might have changed and nowadays the maintainers might be more aware of the security ecosystem. Second, as investigated in Chapter 3 in recent years there have been more vulnerabilities published which might improve data-driven approaches. Finally, the deep learning models might suffer from concept drift on the most recent vulnerabilities.

Vulnerability severity The severity of the vulnerability can massively impact how the maintainers of the open source project handle the fixing process. Here counter-intuitively we would like for the methods to spot fewer vulnerability traces for more severe vulnerabilities, as this would mean that these exploits are handled with more care and dedication than less serious issues. We also expect a lack of data in the low-criticality class, as these vulnerabilities are rarely disclosed or are linked to vulnerabilities with a higher score.
We split the dataset according to the commonly used severity classes linked to the scores from the Common Vulnerability Scoring System (CVSS): low (0-4), medium (4-7), high (7-9) and critical (9-10).

Commit Size Next, we move to the properties of the reference itself. We want to analyze the impact of the commit size on the recall of our methods. The number of changed lines in the commit might affect the recall in multiple ways. First, based on the findings from Chapter 6, we might expect the data-driven approaches to disregard big commits as the majority of security-relevant commits are small fixes and patches. On the other hand, a bigger commit may contain more information which is easier to extract with sampling methods or with feature extraction. Finally, some big commits may also contain documentation or elaborate commit messages which address the security fix or give credit to the discoverer of the vulnerability. While this class comparison is mostly meant for commit classification, we also apply the phrases search on the commit message as in the case of the experiment.
We split the dataset into 3 sets of the bottom, middle and top thirds by the commit size.

Repository stars count Finally, the repository stars count represents how popular and appreciated the project is within the open source community. It also often correlated with older age and bigger size of the repository. We believe that methods might work differently on repositories with different star counts. It's likely that more starred repositories are more mature and handle the disclosure process more professionally, decreasing the number of traces. Larger repositories might also impact commit classification methods, where there is more data available, but also there are more patterns and more content to distinguish.
Similarly, we split the dataset into 3 sets of the bottom, middle and top thirds by the number of stars in the repository.

Before comparing the methods recall, it's important to mention that these values are calculated in a slightly different way for each method. As the methods operate on different data, their domains are different. The phrase-search method is the most general and can be applied to any vulnerability for which we have a documented reference – either a commit or an issue. The label-based method relies on labeled issues as such it is applied only to vulnerabilities with an issue reference. Finally, the commit classification (CL) methods can be applied only to vulnerabilities from which it was possible to extract and mine security-relevant commits. This means that while the counts presented below mean always the same,

i.e. the number of vulnerabilities spotted with the method, the recall is calculated based on different sets of all possible vulnerabilities to spot.

Finally, as these results represent only the recall of the method they do not show the full picture. To get more insights, the results have to be combined with the precision, applicability and time gained results from previous chapters, which we discuss below.

8.2 Results

Table 8.1: The number of vulnerabilities spotted by each method in each year.

	2017	2018	2019	2020	2021	2022
Security phrase	1703	2096	1900	1786	2818	2225
Security label	163	230	188	204	140	126
Feature-based CL	233	271	267	284	252	161
Deep learning CL	604	556	499	618	640	516

Table 8.2: Recall of different methods year by year.

	2017	2018	2019	2020	2021	2022
Security phrase	0.66	0.71	0.70	0.60	0.54	0.51
Security label	0.09	0.09	0.09	0.15	0.06	0.06
Feature-based CL	0.15	0.15	0.19	0.22	0.10	0.09
Deep learning CL	0.36	0.33	0.33	0.44	0.33	0.28

Disclosure date First, let us discuss the data available each year for each method. Notably, the order of magnitude of data available stays consistent across all years. The most data is of course available for the phrase search-based method which was able to work with around 4,000 vulnerabilities each year.

Next, the recall of the methods is shown in Table 8.2 and Figure 8.1. Two trends can be spotted: one for the security-related phrases method which peaks in years 2018 and 2019 at around 70% proceeds to fall; second for the rest of the methods for which the recall rises until 2020 and falls off after that. The latter result is unexpected as especially for the machine learning results, we expected them to work best on the most populous 2021 class. The peak might be a result of the COVID-19 pandemic which could have prompted maintainers to communicate and document their activities in the repository, but at the same time left more traces for the methods to spot.

The continued drop of explicit referring to the vulnerability is a good sign about the awareness of responsible coordination of the vulnerability fixing problem. The recall of other methods is also dropping for 2 years, proving this point further. While good news for the security field, this is not beneficial to our methods which seem to work worse on current, more important vulnerabilities than on the vulnerabilities from 4 years back. To counteract

this issue we propose to increase the relevance of the freshest traces in selecting security phrases or training machine learning models.

Table 8.3: Number of vulnerabilities spotted in each severity class

Method	LOW	MEDIUM	HIGH	CRITICAL
Security phrase	71	5195	4698	2505
Security label	17	423	399	209
Feature-based CL	24	571	597	274
Deep learning CL	25	1388	1301	712

Table 8.4: Recall of each method on vulnerabilities in given severity class

Method	LOW	MEDIUM	HIGH	CRITICAL
Security phrase	0.50	0.60	0.59	0.63
Security label	0.28	0.08	0.09	0.09
Feature-based CL	0.20	0.13	0.14	0.19
Deep learning CL	0.23	0.33	0.32	0.43

Severity score As before, let's first reflect on the amount of data available in each class. We see that in the medium and high severity class there is more data available than in the other classes. Furthermore, as mentioned before, the least represented class is the low severity class, where each method had less than 150 vulnerabilities to work with and each spotted less than 100. As mentioned, this is likely due to low-severity issues not being disclosed so frequently and if they contain an alias to a vulnerability with a higher severity, that severity was used instead. As such the uncertainty of these results is high and should be generally discarded.

Moving on to the recall depicted in Table 8.4 and the Figure 8.2, as expected the security-phrases search has the biggest recall, which is of course balanced out by the low precision. The method however is consistent across all severity classes with little difference in the recall. On the other side, the deep learning model increases its performance for the critical vulnerabilities class by 10 percentage points. This result is surprising as we expected the method to work best in the classes where the most data was provided. A similar increase, but only by 5 percentage points, can be observed by the feature-based commit classification method. Both these findings suggest that critical vulnerabilities may be handled in a more generic way than others. Lastly, the label-based solution has recall below 10% across all classes apart from low. While this might be a numerical flaw due to little amounts of data, the low-severity vulnerabilities may be disclosed from repositories where these security labels are used because of a more advanced security culture.

From the software engineering perspective, these results are somewhat worrying as they suggest that using machine learning models it's easier to spot the most severe vulnerabilities. Luckily, for methods where the precision is also high, the recall of the method is below

50%. However, security-related phrases occur with similar probabilities in critical, high, and medium vulnerability references hinting that maintainers do not pay too much attention to hiding the critical vulnerabilities before the public before the disclosure. This equal result also shows that the selected set of key phrases was universal for all severity classes.

Table 8.5: Number of vulnerabilities spotted by each method with references to commits in specific size class.

Method	Small commit	Medium commits	Large commits
Security phrase	1225	1038	1164
Feature-based CL	500	518	535
Deep learning CL	1290	1212	1350

Table 8.6: Recall of different methods on different commit size classes.

Method	Small commit	Medium commits	Large commits
Security phrase	0.28	0.31	0.32
Feature-based CL	0.07	0.10	0.13
Deep learning CL	0.21	0.23	0.29

Commit Size We found that to split the dataset of security-related commits into approximate thirds, the thresholds had to be set at 10 and 40 changed lines in the commit. As such, the bottom class are commits that change less than 10 lines, the mid class are the commits that change from 10 to 40 lines, and the top class are the larger commits that changed more than 40 lines. The top class was usually a bit underrepresented as the threshold was a bit offset and a notable amount of commits in this class were not mined. Again, the phrases search method had the most data to work with as it can be applied to all security-related commits, while the commit classification methods were limited to the sets of classified commits.

Here, an obvious realization is that the security-related phrase search is not performing that well on commit messages. The method shows a recall of only around 30% recall in all classes, much lower than the overall value of 50% evaluated in Chapter 4. This result was expected as the method was designed for primarily issues, nonetheless, it's surprising that it's not better for larger commits where we'd expected more elaborate commit messages. On the other hand, both commit classification methods seem to work better for the top third of commits. We expected they would work best in the middle class, where the size of the commit is closest to the median. However, both feature-based and deep learning based approaches increase recall with bigger commits. It suggests that the extra information in the commits is more important than the single feature of commit size. Furthermore, the biggest increase is seen for the deep learning models from mid to top class. This may be the effect of cropping the data by the models (maximum of 20 files and an input length of 512 for each file) which combined with more data available, yielded the best results. The results are shown in Table 8.6 and Figure 8.3.

This is an important finding from the perspective of a person that is reviewing pushed commits for being security-related. Simple commits are probably easier to assess than larger ones. As such, one could design a tool that always prompts for manual review for changes that changed less than 10 lines of code, and uses more profiled models on the larger commits.

Table 8.7: Number of vulnerabilities spotted by each method with references to repositories in each star count class.

Method	Bottom-stared	Medium-stared	Top-stared
Security phrase	2388	3215	4176
Security label	107	236	510
Feature-based CL	175	407	887
Deep learning CL	540	1171	1753

Table 8.8: Recall of different methods on different repository classes.

Method	Bottom-stared	Medium-stared	Top-stared
Security phrase	0.53	0.51	0.43
Security label	0.05	0.07	0.14
Feature-based CL	0.14	0.12	0.11
Deep learning CL	0.31	0.32	0.23

Repository star count Lastly, we divide the vulnerabilities according to the number of stars of the repositories they refer to. We have analyzed the number of stars of repositories for which we found security-related commits to find that to split the dataset into approximate thirds one needs to split it at 400 and 3,200 stars.

First, for the phrase search method, we find a lower recall for the more starred project, suggesting these projects handle the vulnerabilities less explicitly or with fewer public discussions. Similarly, commit classification approaches perform the worst on top starred repositories where the drop for the deep learning method is especially significant. We suspect that this drop is due to the added complexity of larger, more starred repositories which the model couldn't understand. It's also possible that the drop is due to the top-starred repositories being from different programming languages (C or JS) while the VulCurator models perform best on Java and Python. Finally, it's interesting to see that the security label-based approach is performing better on the top third of repositories than the feature-based commit classification.

The increase in the recall of the labels-based method suggests that more starred projects have better awareness about software security. Interestingly, the vulnerabilities in these top-starred repositories are harder to spot using security-related keywords than the other thirds. This suggests they either handle vulnerabilities better or the list was not matching their domain. Next, the feature-based commit classification method exhibits a slight drop in

recall but remains mostly constant across all classes suggesting the security commits show similar feature patterns across all types of projects. Finally, the deep learning approach experiences an unexpected drop in performance for most starred repositories. This may be due to the increased complexity of these repositories or due to the poor generalization of the task.

8.3 Summary

- The relative recall of the methods in general stays consistent with phrase-search finding the most vulnerability references followed by deep learning classification, feature-based classification, and label-based approach. If the phrase search is limited (e.g. to only commit messages), its recall drops significantly.
- The recall of all methods is falling in the last 2 years suggesting improving security practices and/or a need to adjust the method to prioritize the latest data.
- The recall of machine learning methods is highest in the class of critical vulnerabilities, hinting these leave the most generic traces out of the evaluated classes.
- Larger commits are easier to spot, which could be utilized to implement methods to spot only these commits while relying on manual review for the smaller ones.
- More starred repositories are easier to spot using label-based methods compared to the less starred repositories. All other methods experience the inverse relation.

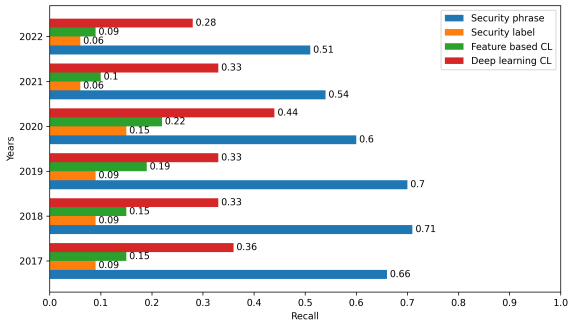


Figure 8.1: Methods' recall on vulnerabilities disclosed in different years.

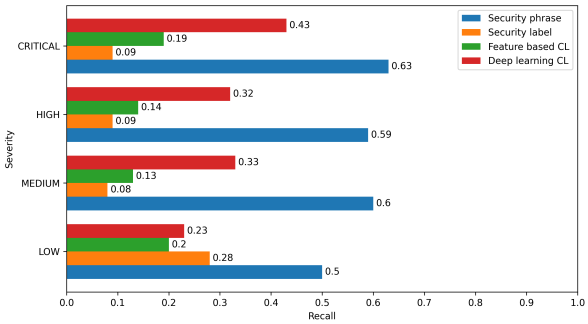


Figure 8.2: Methods' recall on vulnerabilities with different severity scores.

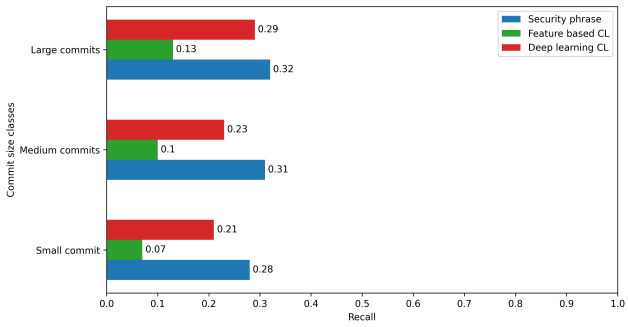


Figure 8.3: Methods' recall on vulnerabilities with references with different commit sizes.

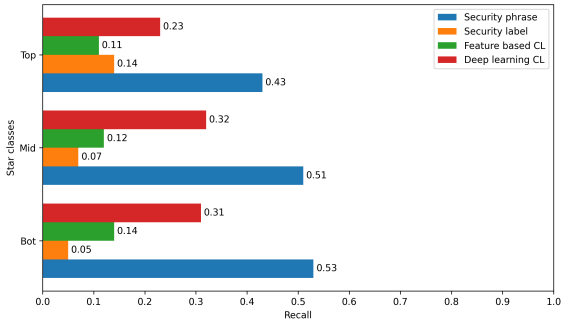


Figure 8.4: Methods' recall on vulnerabilities with references with different repository stars.

Chapter 9

Discussion

In this chapter, we analyze the threats to the validity of the project, future work together with suggestions about the security field, and the ethics of the project. Before that, however, let's elaborate on the findings from the experiments and discuss their impact on our main goals: determining what are the characteristics of the security-related activity and determining how easy it is to spot the activity in open source repositories.

Regarding the first goal, we can state that **security-related issues and commits are similar to regular ones coming from everyday development.** First, we showed that in 47% of cases vulnerability-fixing efforts only explicitly mention security-related phrases after the disclosure or do not mention them at all. This result might be impacted by our selection of phrases that we searched for, but including more phrases might further decrease the already low precision. This result is somewhat expected as it's one of the recommendations in the CVD process. In the coordination manual, it's recommended to not refer to the vulnerability while fixing the issue. Sadly, this is not ideal as these commits and issues are nonetheless public and could be utilized by malicious parties.

Second, we investigated issues with security-related labels. From the gathered activity we were forced to remove 53% of issues that were either invalid or generated by bots updating dependencies. We tried to remove these by filtering for commits with changes in code files. While it worked to some point, some automatic updates still contained code changes made by autoformatter or were referenced by unrelated code-changing commits reducing the precision which came out to be 77%. We find that only around 4% of projects use security labels to coordinate security tasks in their repository. Moreover, we were able to find a vulnerability disclosure only for 20% of the issues. This suggests that even then the security aspect of an issue is known, the problem isn't usually propagated to a vulnerability database.

Next, we looked at many metrics for security-related commits compared to background commits. For the metrics we mined, clear patterns can be seen for security-related commits. Sadly, the same patterns are expressed by the majority of background commits. This was further confirmed by machine learning models where almost no particular feature was much more important than the others. The only exception was the *'secur' in message* feature for the JavaScript commits which at an importance of 0.05 had twice the importance of other features possibly pointing to worse practices in the npm ecosystem.

Then, judging by the relative performance on the sample level of the deep learning approach, it doesn't matter whether the sampling method is very localized (rolling window method), contains the entire change (added code method), or contains parts of both the change and the entire file (AST path method). In all these cases the F1 score of the model was similar at 0.27, suggesting the same amount of information was extracted. The sampling may have a larger impact if the code is interpreted by a less complex embedding tool. Furthermore, based on the low F1 score we also believe that some parts of the security-relevant commits are the same as some parts of regular commits.

Finally, judging by the lower performance of the VulCurator models [31] on our dataset of security-related commits we suspect that the task is poorly generalizable. The same commit in one repository might be security-related while being a background commit in another one. It also may be that the models are learning only a little about the general patterns of security-related commits (if there are such) and instead are focusing on the tendencies of the vulnerabilities within the projects. For example, it has been shown that often multiple vulnerabilities are follow-ups to a vulnerability that was published beforehand. Such a behavior can be learned by the model and it will recognize the vulnerability fixes based on the fact they modify the same file.

Regarding the second goal, we can state that **it's possible to automatically spot security-related issues and commits but with limited performance** as expressed by the low F1 scores below. In the case of all evaluated methods a final, manual review step is required as the performance was not sufficient to use the output of the tool directly. Furthermore, the highest recall was achieved by the phrase search, but it still caught only about half of the possible vulnerabilities. The task of finding these traces also involves a common drawback of rare class classification, where to increase recall, one has to sacrifice precision.

First, the method based on the phrases search suffered from low precision caused by the overwhelming number of mentions of security-related phrases in reactions to disclosed vulnerabilities in the projects' dependencies. In the majority of cases, these mentions were generated by bots which could make the filtering easier but sometimes they were added by human users. Additionally, many phrases were showing up in discussions related to security, but not in the context of fixing but rather improving it. Even with the relaxed review requirements, which produced an unrealistic precision of 0.35, the F1-score of the method is only 0.42. Nonetheless, the method could be used to scan what security topics are discussed within a repository which could give hints on how to handle the dependency within own projects.

The problem of low precision was partially solved in the second experiment where filtering based on commits containing code changes was applied as the precision jumped to 77%. This sadly means pushing the discovery closer to the disclosure date, as one needs to wait for any commits to be pushed. However, alternative filtering methods can be used to remove the majority of the bot-created activity. The label-based method of prealerting yielded a really promising result and we believe it could be applied in practice. However, it is limited to only a handful of projects that do use security labels to coordinate work and the overall F1 score is 0.14.

Moving to the feature-based commit classification the model performed surprisingly well given the apparent lack of standing-out features. The major obstacle for the models

was the heavily imbalanced classes, which we tried to counteract by adapting the class ratio. We believe that the performance of the models can be further improved by creating artificial features being combinations of the ones calculated. Filtering the data might also improve the performance which in our experiments came out to be a 0.14 F1 score on the entire dataset.

Finally, we were unable to reliably evaluate our own deep learning models. However, experiments show a similar or better performance as in the case of the VulCurator models, which in the end were used to evaluate the DL method. While the results were better than the results of the feature-based models, the improvements were not as significant as expected and disproportionate to the additional complexity and computational effort. The models didn't yield results as good as on the dataset they were initially evaluated on suggesting that the patterns within the SAP-KB dataset don't transfer to our dataset. Of course, it's also possible that our dataset is much noisier as no significant filtering was applied to the commits extracted from vulnerabilities. Nonetheless, the model offered the best precision and recall combination achieving an F1 score of 0.36.

We additionally evaluated the methods' performance across vulnerabilities disclosed in different years and on vulnerabilities in different severity classes. First, the recall of all methods is falling in the last 2 years which can be caused both by the improving security practices or by methods being too focused on older vulnerabilities. An improvement in security practices is also hinted at by the fact that the recall of the phrase-search method is falling since its peak in 2018. On the other hand, it seems that critical vulnerabilities are easier to spot, which we believe is due to the ad-hoc nature of fixing them and a larger response being generated for the critical issues.

In this evaluation, we also found that the top third of largest commits is easier to spot than the bottom and mid thirds, which was a surprise as the expectation was for the models to work best around the middle of the commit size distribution. We believe that this gives the advantage to the models when used in pair with manual review as it's the larger commits that are the hardest to check. Lastly, we split the data up also according to the star count of the referenced repository. Here, besides the label-based approach, all methods performed worse on vulnerabilities in more starred repositories. This especially affected the deep learning approach, which we believe is due to different practices in some popular repositories (e.g. Linux, React, Vue) that are different from the training data of the SAP-KB dataset.

9.1 Future Work

Here we would like to discuss possible remediation for the early traces of vulnerabilities and suggest ways to improve the methods. First, maintainers and developers of client projects should consider more responsible disclosure and fixing coordination processes. Not enough projects have disclosure policies and these aren't always followed – either intentionally or by accident. Also, when an issue is recognized to have security implications it should be fixed in a confidential matter, merged and released as soon as possible. For example OpenSSL[1] recently disclosed two connected vulnerabilities: CVE-2022-3786 and CVE-

[1] https://www.openssl.org/

2022-3602. First, the issue was disclosed to the maintainer privately. After the issue was reported it was probably validated and fixed in private. After that, the team announced that a security release is planned for 2022.11.01 that fixes a critical vulnerability,[2] which was further promoted by the news and on Twitter. Until that moment, no new changes were pushed to the main branch and it remained frozen. On the 1st of November a new release was available and two commits[3,4] were merged into the main branch. Of course, such an approach is not possible in all cases, but with the warning, all security teams can be prepared for the update to the new version or at least improve their monitoring. Nonetheless, this process shows that it's possible to coordinate the process responsively so that no traces are visible just until the disclosure

Addressing the fact that public issues might address security flaws, it could be considered for the issues first to be secret until a maintainer verifies that the issue doesn't disclose anything that it shouldn't. This period of confidentiality could be applied to issues (or commits) in which content was classified as security-related with issue (or commit) classification. This way the vulnerability is discoverable only much closer to the disclosure date. On the other hand, if the security flaw is already traceable online, maintainers should consider disclosing the vulnerability even before the fix is available.

Finally, a finding not coming directly from the results of this project, but realized while investigating the world of open source. Many modern open source packages offer a vast amount of features that in the majority of cases aren't utilized and sometimes are enabled by default. This flaw was for example the cause of the Log4Shell vulnerability (CVE-2021-44228). As such we would recommend splitting the functionality of the package into small sub-packages and disabling features by default. On the side of the client projects, they should employ zero trust architectures and validate and escape the handled data in all appropriate places.

On the side of improving the proposed methods, an obvious improvement to the phrases search method would be to replace it with NLP models to determine the nature of the issue or discussion. However, filtering of the bot-generated content or content containing vulnerabilities from other projects still would be required. Similar approaches could be utilized in the label-based method. As already mentioned the feature-based commit classification could be improved by more advanced feature engineering. We also suggest including more project-wide metrics or profiling the models to a specific group of projects e.g. servers or single-page applications. Similar improvement can also be made in deep learning models where a representation of the context of the project, extracted from the project's readme or dependencies, could help the model put the change into perspective and distinguish a crash of a server from a crash of a desktop application. Alternatively, some projects suggest using a graph neural network on the repository structure to capture the purpose of the file as done in LineVD [18]. Lastly, our approach was also fairly naive, with typical supervised learning to tackle the problem. Other approaches that focus more on the differences between positive and background commits could work better in the heavily imbalanced scenario.

[2] https://mta.openssl.org/pipermail/openssl-announce/2022-October/000238.html
[3] https://github.com/openssl/openssl/commit/3b421ebc64c7b52f1b9feb3812bdc7781c784332
[4] https://github.com/openssl/openssl/commit/680e65b94c916af259bfdc2e25f1ab6e0c7a97d6

9.2 Threats to validity

While many efforts were made to validate our methodologies and assumptions beforehand, some of them may be wrong. First of all, almost all research done in this project relies on the assumption that GitHub objects linked in the vulnerabilities are security related. One would hope that the references point only to issues where the fixing work was coordinated or to the commits patching the exploit, but references can also point to releases, documentation issues, etc. To solve this issue, researchers apply various automatic filtering or utilize hand-reviewed datasets. In our approach we decided to go with a fairly relaxed approach, filtering only based on the number of commits pulled in by each reference to avoid parsing too general objects like releases. While this means we are not introducing any 'fake' patterns in the data with our filtering, it means that our models have to work on a stretched space of values, decreasing the performance of our data-driven approaches. The same applies from the other side – some commits from the background dataset could have been security-related commits, which were never referenced in any vulnerability or were implementing security guards or features preemptively. While the assumption is that these commits are sparse, likely some of them were randomly included in the background datasets.

This project also included many applications of manual tasks. In particular the manual validation in Chapters 4 and 5 or manual selection of metrics in Chapter 6. We have taken an effort to perform these manual steps as accurately as possible and we are confident in our results, but it's always possible that some errors were introduced. To limit these errors we assumed fairly relaxed review criteria to get the upper bound of the precision and interpret the result with that in mind.

Lastly, an external threat to the validity of the project is that developers are not significantly motivated to update packages due to vulnerabilities in them [16, 48], which is a worrying finding not only for our project but for the entire software industry. If developers update either way months after the disclosure of the vulnerability – there is no point in gaining an additional couple of days before the disclosure. While this might be the case in many companies, there are software projects that are security critical and do update as soon as possible and would be interested in pushing their security even further. Second, we hope that as more work is being done in the security field, to which this work contributes, more awareness will be put on the topic, possibly changing developers' behavior. When that happens, the vulnerability systems should be more mature so that the vulnerability disclosure is really the first public information on a security exploit. Otherwise, why should developers care to patch them urgently, since it was already discoverable e.g. for a week on GitHub? We hope that as more work is done in the field this issue becomes less and less relevant as it's already seen by the year-by-year plots in Chapter 8.

9.3 Ethics

Lastly, as the project is security related it's important to take a look at the bigger picture and discuss the ethical side of found implications. In this research, we show ways how vulnerabilities can be spotted before their disclosure, which can be utilized by malicious parties to target exploits the general public is not yet informed about and not prepared for.

Note however that we are using data that is already publicly available and is commonly used in the security field. It's highly probable that more established hacker groups, advanced persistent threats (APTs), are already monitoring open source packages for security-related issues, especially if they are aware of the technological stack used by the target company or government. Similarly, the same applies to security departments within these entities. The details of how this monitoring is utilized and the damages caused by attackers are usually kept confidential, but there are documented cases where APTs utilized old vulnerabilities to attack systems.[5,6]

We believe that our research is essentially leveling the playing field for businesses that cannot afford to have dedicated personnel for keeping track of current proceedings in their package dependencies or security news, as they are focusing on developing the product. We firmly believe that in the vast majority of cases when traces are already public, it is beneficial for the vulnerability to be shared as soon as possible so that projects can react to it, as it can be already known to people interested in exploiting it.

[5] https://techcrunch.com/2022/09/08/north-korea-lazarus-united-states-energy/
[6] https://www.wired.com/story/north-korea-hacker-internet-outage/

Chapter 10

Related Work

In this chapter, we analyze similar approaches and results achieved by other researchers working in the field. We separate the related works into two parts, first we discuss the latest models performing security commit classification and then we discuss findings about delays in vulnerability disclosures.

10.1 Commit classification

In the last 2 years, many papers have appeared tackling the task of vulnerability prediction or security-related code classification. Table 10.1 sums up the performance metrics reported by selected papers relevant to this project. Note, that these values cannot be compared directly as they were achieved on different datasets, with different prevalences and with different scopes. Apart from the popular classification metrics of precision(Pr), recall(Rc) and F1 score, the area under the precision-recall curve(PR-AUC) is utilized as a threshold-independent metric, and finally, the recall at 5%(Rc-5) of reviewed lines of code as an effort aware metric accounting for the need to review the result of the model.

Table 10.1: Reported performance metric of state-of-the-art models.

#		Pr	Rc	F1	Rc-5	PR-AUC
1	VulFixMiner, Zhou et al. [59]				0.49	0.81
2	VulCurator, Nguyen et al. [31]			0.81		
3	Commit2Vec, Cabreara et al. [5]	0.73	0.71	0.73		0.82
4	LineVul, Fu et al. [12]	0.97	0.80	0.91		
5	LineVD, Hin et al. [18]	0.56	0.27	0.36		0.642
6	E-SPI, Wu et al. [54]	0.88	0.92	0.89		

First, many of the papers [59, 31, 18, 12] use the established CodeBERT [11] model to generate the embeddings of code samples. The model is used in various setups. Some models use it both to embed the commit message and code, while others parse it specifically for code. Interestingly no model uses the transformer in the initial way how it was

conceived where the NL and the PL inputs are concatenated as a single input to the model. Furthermore, since its release in 2020, a new iteration of the model was released – GraphCodeBERT which we test out in this project. We address these limitations by applying the new model and concatenating the commit message together with the code sample. The approaches of other papers usually include graph neural networks or other embedding techniques.

Second, sampling methods vary more across different projects. The most common approach is the simplest by just taking the current or added code and processing it as a whole, function by function or line by line [12, 18, 59, 31, 5]. The major limitation of this approach is in handling bigger changes/functions where the input is cropped. But, as we found, the majority of commits and functions are small causing this situation to affect only selected commits. On the other hand, other papers propose more advanced sampling methods based on AST parsing or control flow graphs [5, 54]. While it's suspected that the sampling methods based on AST paths produce the best results, we investigate additionally the added code and the rolling window methods as in some cases an AST parser is not available. In our experiments, we found the impact of the sampling methods on the performance to be limited when using an advanced embedding model.

As pointed out in the recent survey [25], one of the current limitations of many papers is a lack of an evaluation that would truly reflect the industry application of the models. This is one of the limitations we tried to address by evaluating the VulCurator models trained on SAP-KB dataset [37] on the commits we extracted from vulnerabilities.

10.2 Vulnerability delays

Our approach to investigating the delays in vulnerability delays was fairly general and tool oriented. We applied a sense of practicality by analyzing only totals of the vulnerabilities that we were able to spot using methods, but lost other information that can be gathered from more in-depth analysis. By focusing on one specific system one can get more insight into the particular practices of the project and manually review a larger (percentage-vise) set of commits. These case studies were done on other systems like Apache Tomcat, Apache HTTP Server or Linux Kernel [35, 39]. They utilized linked commits in vulnerabilities or changes integrated without many discussions to find commits 12, 54 and 48 days before the disclosure (median time for Tomcat, median time for HTTP Server and average time for Linux).

Next, in our research, we have only used GitHub as an external data source. This limitation isn't that substantial as GitHub is the main development platform for the majority of open source packages and other research showed that the traces usually first appear here. However, to get the full picture and catch more traces other data sources should be considered. Other research has taken a look at other development platforms, social media (e.g. Twitter, Reddit) or mailing lists [47, 20, 42]. They find consistent results with our research for example showing that 17% of CVE mentions on these social sites are before the vulnerability disclosure or that the median coordination delay is about 15 days.

Finally, Imtiaz et al. [22] analyzed the nature of security releases and found that the

median time that security fixes require to be released is 4 days (0, 6, 13 days for npm, PyPI and Maven respectively). Moreover, the advisory was found in their data before the fix was implemented 10.1% of the time and 22.4% of the time before the fix was released. In 39.5% of cases, the security releases weren't mentioning any security fixes hinting at a high prevalence of silent fixes. Finally, they analyzed the time between the security release and the disclosure concluding its median is at 25 days, which is a similar result to our results of 21 days between the disclosure date and the creation of referenced GitHub objects.

In this project, we have utilized the relatively new OSV and GHSA vulnerability databases which currently aren't popular in research. Seeing that the overall results are similar to ours we can conclude that there is a consensus regarding the delays in publishing vulnerabilities compared to the first traces being available. Hopefully, this delay will get smaller over the years and the traceable vulnerabilities rarer.

Chapter 11

Conclusion

In this project, we show that it's possible to spot vulnerability-fixing efforts or initial exploit reports in open source repositories before the disclosure of the vulnerability. We find that the task of spotting these traces automatically is made difficult by the similar nature of security-related activity compared to activity originating from everyday development. On one side, this is good news when considering that the fixing is somewhat hidden from attackers. On the other hand, it's bad news for companies that need to stay in front of vulnerabilities in their dependency chain, as they need to have people spending time reviewing the activity in open source repositories and other security feeds.

In our experiments we have investigated methods based on filtering the activity with security keywords, looking for specific labels in issues, applying feature-based commit classification and evaluating state-of-the-art deep learning models. None of the evaluated methods had sufficient recall and precision at the same time to solve the problem automatically, however, all methods can be used to assist in spotting vulnerabilities or gaining insights into the specific repository.

We also discussed possible actions that one could take to improve open source security. We conclude that the most important factor is for both the maintainers and the developers of client projects to be responsible – follow disclosure policies and handle security issues with utmost priority. Sadly, as this not always happens, maintainers should consider abandoning the coordinated vulnerability disclosure model and notify developers as soon as possible. We suggest using issue and patch classifiers similar to the ones we investigated to initially hide potentially security-related objects until a maintainer approves them.

We hope that our experiments might inspire other researchers and that this work will bring more attention to open source security.

Bibliography

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] Noah Bühlmann and Mohammad Ghafari. How do developers deal with security issue reports on github? In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1580–1589, 2022.

[4] Jordi Cabot, Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Belén Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 550–554. IEEE, 2015.

[5] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. *SN Computer Science*, 2(3):1–16, 2021.

[6] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[7] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *Acm Sigmod Record*, 25(2):493–504, 1996.

[8] Jinfu Chen, Patrick Kwaku Kudjo, Solomon Mensah, Selasie Aformaley Brown, and George Akorfu. An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection. *Journal of Systems and Software*, 167:110616, 2020.

[9] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. The delta maintainability model: Measuring maintainability of fine-grained code changes. In

2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pages 113–122. IEEE, 2019.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014. doi: 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. *Mining Software Repositories Conference 2022*, 2022.

[13] Daniel E Geer Jr. Complexity is the enemy. *IEEE Security & Privacy*, 6(6):88–88, 2008.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[15] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1):389–422, 2002.

[16] Rens Heddes. Vulnerability risk modelling in open source software systems. *TU Delft Master Thesis*, 2022.

[17] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*, pages 392–401. IEEE, 2013.

[18] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. *arXiv preprint arXiv:2203.05181*, 2022.

[19] Daan Hommersom, Antonino Sabetta, Bonaventura Coppola, and Damian A. Tamburri. Automated mapping of vulnerability advisories onto their fix commits in open source repositories, March 2021.

[20] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O. Hall, and Adriana Iamnitchi. Mentions of security vulnerabilities on reddit, twitter and github. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 200–207, 2019.

[21] Allen D Householder, Garret Wassermann, Art Manion, and Chris King. The cert guide to coordinated vulnerability disclosure. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Pittsburgh United States, 2017. https://vuls.cert.org/confluence/display/CVD/4.5+Gaining+Public+Awareness.

[22] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 2022.

[23] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Predicting issue types on github. *Science of Computer Programming*, 205:102598, 2021.

[24] Unicode fundation Ken Whistler. Unicode normalization forms, unicode standard annex 15, 2022.

[25] Triet HM Le, Huaming Chen, and M Ali Babar. A survey on data-driven software vulnerability assessment and prioritization. *ACM Computing Surveys (CSUR)*, 2022.

[26] Triet Huynh Minh Le, Bushra Sabir, and Muhammad Ali Babar. Automated software vulnerability assessment with concept drift. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 371–382. IEEE, 2019.

[27] Triet Huynh Minh Le, Roland Croft, David Hin, and Muhammad Ali Babar. A large-scale study of security vulnerability support on developer q&a websites. In *Evaluation and assessment in software engineering*, pages 109–118. Evaluation and assessment in software engineering, 2021.

[28] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 97–106, 2017.

[29] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[31] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. Vulcurator: A vulnerability-fixing commit detector. *arXiv preprint arXiv:2209.03260*, 2022.

[32] Naz Zarreen Zarreen Oishie and Banani Roy. Commit-checker: A human-centric approach for adopting bug inducing commit detection using machine learning models. In *15th Innovations in Software Engineering Conference*, pages 1–3, 2022.

[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[35] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat. In *2019 12th IEEE Conference on software testing, validation and verification (ICST)*, pages 68–78. IEEE, 2019.

[36] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 348–351, 2014.

[37] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE, 2019.

[38] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48. Citeseer, 2003.

[39] Ralf Ramsauer, Lukas Bulwahn, Daniel Lohmann, and Wolfgang Mauerer. The sound of silence: Mining security vulnerabilities from secret integration channels in open-source projects. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 147–157, 2020.

[40] Herimanitra Ranaivoson, Mourad Badri, Jianyu Hu, Jing Zhang, Hongrui Nian, Wenjiang Chen, Jinxing Hu, Diping Yuan, and Jiaoyang Liu. Software fault severity prediction using git history metrics and commits. *J. Softw.*, 17(2):36–47, 2022.

[41] Sofia Reis, Rui Abreu, and Luis Cruz. Fixing vulnerabilities potentially hinders maintainability. *Empirical Software Engineering*, 26(6):1–27, 2021.

[42] Jukka Ruohonen, Sampsa Rauti, Sami Hyrynsalmi, and Ville Leppänen. A case study on software vulnerability coordination. *Information and Software Technology*, 103: 239–257, 2018.

[43] 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 (ICSME)*, pages 579–582. IEEE, 2018.

[44] Arthur D Sawadogo, Tegawendé F Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. Learning to catch security patches. *arXiv preprint arXiv:2001.09148*, 2020.

[45] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[46] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50, 2008.

[47] Prasha Shrestha, Arun Sathanur, Suraj Maharjan, Emily Saldanha, Dustin Arendt, and Svitlana Volkova. Multiple social platforms reveal actionable signals for software vulnerability awareness: A study of github, twitter and reddit. *Plos one*, 15(3):e0230250, 2020.

[48] L. Brandts Software Improvement Group, M. Bruntink. 2022 sig benchmark report — through the sig looking glass, 2022.

[49] Sonatype. 2022 state of the software supply chain, 2022.

[50] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018. doi: 10.1145/3236024.3264598.

[51] Black Duck Audit Services Synopsys. 2022 open source security and risk analysis report, 2022.

[52] Andrzej Westfalewicz. Vulnerability prealerting by monitoring the online repositories of open source projects - reproduction package, 2023. doi: 10.5281/zenodo.7457615.

[53] Jason L Wright, Jason W Larsen, and Miles McQueen. Estimating software vulnerabilities: A case study based on the misclassification of bugs in mysql server. In *2013 International Conference on Availability, Reliability and Security*, pages 72–81. IEEE, 2013.

[54] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[55] Terry Yin. Lizard: An extensible cyclomatic complexity analyzer. *Astrophysics Source Code Library*, pages ascl–1906, 2019.

[56] Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. An empirical study of security issues posted in open source projects. In *Proceedings of the 51st Hawaii international conference on system sciences*, 2018.

[57] McAfee Zhanna Malekos Smith, Eugenia Lostri. The hidden costs of cybercrime, 2020.

[58] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software*, 168: 110659, 2020.

[59] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716. IEEE, 2021.

[60] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.

[61] Yuxiang Zhu, Minxue Pan, Yu Pei, and Tian Zhang. A bug or a suggestion? an automatic way to label issues. *arXiv preprint arXiv:1909.00934*, 2019.

[62] Yufan Zhuang, Sahil Suneja, Veronika Thost, Giacomo Domeniconi, Alessandro Morari, and Jim Laredo. Software vulnerability detection via deep learning over disaggregated code graph representation. *arXiv preprint arXiv:2109.03341*, 2021.

Appendix A

Commit classification features

In this section, we describe all features used in the feature-based models in the third experiment, together with their motivation and aggregation method. Furthermore, we provide the list of ecosystems where each of these features was used in the final model.

A.1 Feature descriptions

A.1.1 Labels

label_repo_full_name Label for repository distinction.

label_sha Label for commit distinction.

label_commit_date Label for historical analysis.

label_securiry_related Label for indicating which class commit belongs to.

A.1.2 Author and committer information

author_to_commiter_date_diff The difference between the committer and author data potentially shows how fast the commit was merged.
Used in: npm, PyPI, Maven, ALL

same_author_as_commiter Indicates if the author is the same as the committer potentially indicating the commit was merged manually without a pull request.
Used in: npm, Maven, ALL

committed_by_bot Was the commit committed (usually merged) by a bot.
Used in: Maven, ALL

authored_by_bot Feature indicating whenever the commit was authored by a bot potentially indicating an automatic code change.
Used in: ALL

author_in_top_100 Feature indicating whenever the author of the commit is in the top 100 contributors of the repository. Since contributors are given by logins, while the commits are signed with emails we compare the Levenstein distance between the login and the first segment of the email to the length of the login. If the ratio is lower than 0.33 the identifiers likely match.
Used in: npm, Maven, ALL

A.1.3 Delta maintainability model

dmm_unit_complexity DMM metric related to McCabe's cyclomatic complexity of methods and/or function changed in the commit.
Used in: npm, PyPI, Maven, ALL

dmm_unit_interfacing DMM metric related to sizes of interfaces changed or declared in the commit.
Used in: npm, PyPI, Maven, ALL

dmm_unit_size DMM metric related to lines of code of methods and/or functions changed in the commit.
Used in: npm, PyPI, Maven, ALL

A.1.4 Delta maintainability model

KEYWORD_in_message Feature indicating whenever a specific keyword from a predefined list appears in the commit message. Keywords that were used in all ecosystems were 'attack', 'secur' and 'vulnerab'.
Used in: 3 in npm, 3 in PyPI, 3 in Maven, 13 in ALL

KEYWORD_in_title Feature indicating whenever a specific keyword from a predefined list appears in the commit title, i.e. the first 72 characters of the first line of the commit message.
Used in: 0 in npm, 0 in PyPI, 0 in Maven, 3 in ALL

A.1.5 Delta maintainability model

commits_prev_7_days Number of commits authored in the 7 days preceding the commit.
Used in: npm, PyPI, Maven, ALL

commits_next_7_days Number of commits authored in the 7 days following the commit.
Used in: npm, PyPI, Maven, ALL

commits_next_30_days Number of commits authored in the 30 days following the commit.
Used in: npm, PyPI, Maven, ALL

time_to_next_merge The time difference to the next merge commit.
Used in: npm, PyPI, Maven, ALL

commits_to_next_merge Number of commits to the next merge commit.
Used in: npm, PyPI, Maven, ALL

commits_since_last_merge Number of commits since the last merge commit.
Used in: npm, PyPI, Maven, ALL

time_to_prev_commit The difference between the author date and the author date of the previous commit.
Used in: npm, PyPI, Maven, ALL

time_to_next_commit The difference between the author date and the author date of the next commit.
Used in: npm, PyPI, Maven, ALL

A.1.6 Change size

changed_files Number of files changed in the commit.
Used in: npm, PyPI, Maven, ALL

A.2 File level features

A.2.1 Ecosystem association

has_ECOSYSTEM_code Aggregation: flag
The file has an extension matching one of the predefined extensions for given ecosystems. Used for filtering, but also provided to the model so that it can estimate how many code files were modified in the commit.
Used in: npm (npm, PyPI), PyPI (PyPI), Maven (npm, PyPI, Maven), ALL (npm, PyPI, Maven)

has_ECOSYSTEM_like_code Aggregation: flag
The file has an extension matching one of the predefined extensions that are common additions in given ecosystems. Similar to above but includes file extensions that are related to the main extensions of the language.
Used in: npm (npm), Maven (Maven), ALL (Maven)

A.2.2 Change type

is_add Aggregation: flag
Whenever the file was added to the commit. Allows the model to reason on how much the commit is modifying the file structure of the project.
Used in: npm, PyPI, Maven, ALL

is_rename Aggregation: flag
Whenever the file was renamed or moved in the commit. Allows the model to reason on how much the commit is modifying the file structure of the project.
Used in: npm, ALL

is_delete Aggregation: flag
Whenever the file was deleted in the commit. Allows the model to reason on how much the commit is modifying the file structure of the project.
Used in: Maven, ALL

is_modify Aggregation: flag
Whenever the file was modified in the commit. Allows the model to reason on how much the commit is modifying the file structure of the project.
Used in: npm, PyPI, Maven, ALL

A.2.3 Bag of words features

test_in_filename Aggregation: flag
Indicates if the filename contains the string 'test'. Provided to the model for examining the relationship between test files and security-related commits.
Used in: npm

test_in_path Aggregation: flag
Indicates if the full path of the file contains the string 'test'. Provided to the model for examining the relationship between test files and security-related commits.
Used in: npm, PyPI, Maven, ALL

methods_with_KEYWORD_count Aggregation: avg/max
The number of methods with the given security keyword in their name. Only possible when the programming language of the file is handled by Lizard. Security-relevant commits should contain more of these than background commits.
Used in: Maven (4), ALL (3)

KEYWORD_in_file_content - Aggregation: flag
Indicates if a given keyword appears in the file. May hint whenever the file handles security-relevant logic.
Used in: npm (9), PyPI (5), Maven (12), ALL (13)

KEYWORD_in_patch Aggregation: flag
Indicates if a given keyword appears in the file. May hint whenever a commit is security related.
Used in: npm (4), PyPI (2), Maven(6), ALL (12)

A.2.4 Change size

removed_lines_count Aggregation: avg/max
The number of lines removed in the file. We expect security-relevant commits to be smaller and add more code than is removed.
Used in: npm (avg, max), PyPI (max), Maven (avg, max), ALL (avg, max)

added_lines_count Aggregation: avg/max
The number of lines added in the file. We expect security-relevant commits to be

smaller and add more code than is removed.
Used in: npm (max)

changed_lines_count Aggregation: avg/max
The count of added and removed lines. We expect security-relevant commits to be smaller and add more code than is removed.
Used in: npm (avg, max), PyPI (max), Maven (avg, max), ALL (avg, max)

removed_lines_ratio Aggregation: avg/max
The ratio of removed lines counts to the file size. We expect security-relevant commits to be smaller and add more code than is removed.
Used in: npm (avg, max), PyPI (max), Maven (avg, max), ALL (max)

added_lines_ratio Aggregation: avg/max
The ratio of added lines counts to the file size. We expect security-relevant commits to be smaller and add more code than is removed.
Used in: npm (max)

modified_lines_count Aggregation: avg/max
The number of lines where added line number and removed line number matches - suggesting the modification of the line rather than its addition or removal. Especially small fixes, like off-by-one errors, will exhibit mostly these kinds of changes.
Used in: npm (avg, max), PyPI (avg, max), Maven (avg, max), ALL (avg, max)

modified_ratio_count Aggregation: avg/max
The ratio of modified lines as described above to the entire lines in the file.
Used in: None

file_size Aggregation: avg/max
The size of the file. Security fixes may appear more in large files and this feature will allow the model to perform reasoning compared to the filesize.
Used in: npm (avg, max), PyPI (avg, max), Maven (max), ALL (avg, max)

A.2.5 History related

changes_to_file_in_prev_50_commits Aggregation: avg/max
Indicates whenever how many times the file was changed in the last 50 commits. Security fixes may be more present in frequently changed files that are hubs for functionality.
Used in: npm (max), PyPI (max), Maven (avg, max), ALL (avg)

changes_to_file_in_next_50_commits Aggregation: avg/max
Indicates whenever how many times the file was changed in the next 50 commits. Security fixes may be more present in frequently changed files that are hubs for functionality.
Used in: npm (max), PyPI (max), Maven (avg, max), ALL (avg)

is_file_recently_added Aggregation: flag
Indicates whenever the file was added in the previous 50 commits. We expect security to change either old big files that are hubs for changes, or recently added files that weren't tested as much yet.
Used in: npm, PyPI, Maven, ALL

is_file_recently_removed Aggregation: flag
Indicates whenever the file was removed in the following 50 commits. We expect security-relevant changes to stay in the repository.
Used in: npm, Maven, ALL

A.2.6 Code metrics

changed_methods_count Aggregation: avg/max
The number of changed methods in the file in the commit. We expect this value to be lower for security-relevant commits.
Used in: npm (avg, max), PyPI (avg, max), Maven (avg, max), ALL (avg, max)

total_methods_count Aggregation: avg/max
The number of methods in the file. It may suggest that large and complex files are more likely to contain security-related flaws, but also it may be used as comparison bases for other features.
Used in: npm (avg), PyPI (avg, max), Maven (avg), ALL (avg, max)

file_complexity Aggregation: avg/max
The cyclomatic complexity of the file. Motivated as above.
Used in: ALL (avg)

file_nloc Aggregation: avg/max
The number of lines of code in the file. Motivated as above.
Used in: npm (avg, max), Maven (avg, max)

file_token_count Aggregation: avg/max
The token count in the file. Motivated as above.
Used in: PyPI (max), ALL (avg, max)

(avg/max)_method_token_count Aggregation: avg/max
The average and the maximum number of tokens in specific methods. Used as a feature in determining the code quality in greater granularity than the delta maintainability model.
Used in: npm (3), PyPI (2), Maven (2), ALL (3)

(avg/max)_method_complexity Aggregation: avg/max
The average and the maximum cyclomatic complexity in specific methods. Motivated as above.
Used in: npm (2), PyPI (1), Maven (3), ALL (2)

(avg/max)_method_nloc Aggregation: avg/max
The average and the maximum number of lines of code in specific methods. Motivated as above.
Used in: npm (3), PyPI (1), Maven (2)

(avg/max)_method_parameter_count Aggregation: avg/max
The average and the maximum parameter count in specific methods. Motivated as above.
Used in: npm (3), PyPI (2), Maven (2), ALL (2)

Appendix B

Reproduction package content

In this appendix, we enumerate the content of the reproduction package available at [52]. The reproduction is supplemented with the repository available at `https://github.com/NewWwest/masters-project`. Additional descriptions are available in markdown files in the reproduction package and in the repository itself.

Repositories The list of repositories described in Chapter 3 used in case studies in the first two experiments

Vulnerability datasets Precise information on the versions of vulnerabilities used in this project.

Extracted references Vulnerability references pointing to GitHub that we extracted from the databases and resolved using the GitHub API.

Security related commits The datasets for security-related commits as extracted from vulnerabilities, the SAP-KB dataset, and security-labeled issues.

Phrase search The input, query, and result of the phrase search experiment.

Security labels The results of the security-labeled issues experiment, including downloaded issues and manually annotated data.

Mined features Features mined for the feature-based commit classification experiment.

Models and results Evaluation of the best parameters for the models, the models themselves, and the final predictions in the feature-based commit classification experiment.

Deep learning samples The samples mined for the deep learning commit classification experiment.

Deep learning models The results of the final training sessions of our deep learning models.

Deep learning results The classification results of VulCurator models used in the evaluation of the deep learning commit classification experiment.

Recall data Additional data that was collected for calculating the recall of the methods in the evaluation chapter and the results that were used to generate the recall plots.