To Kill a Mocking Bug: Open Source Repo Mining of Security Patches for Programming Education

Andrei-Cristian Iosif
Universität der Bundeswehr München, Germany
Siemens AG, München, Germany

Tiago Espinha Gasiba
Siemens AG, München, Germany

Ulrike Lechner
Universität der Bundeswehr München, Germany

Maria Pinto-Albuquerque
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Portugal

Abstract

The use of third-party components (TPCs) and open-source software (OSS) has become increasingly popular in software development, and this trend has also increased the chance of detecting security vulnerabilities. Understanding practical recurring vulnerabilities that occur in real-world applications (TPCs and OSS) is a very important step to educate not only aspiring software developers, but also seasoned ones. To achieve this goal, we analyze publicly available OSS software on GitHub to identify the most common security vulnerabilities and their frequency of occurrence between 2009 and 2022. Our work looks at programming language and type of vulnerability and also analyses the number of code lines needed to be changed to fix different vulnerabilities. Furthermore, our work contributes to the understanding of real-world and human-made data quality required for training machine learning algorithms by highlighting the importance of homogeneous and complete data. We provide insights for both developers and researchers seeking to improve cybersecurity in software education and mitigate risks associated with OSS and TPCs. Finally, our analysis contributes to software education by shedding light on common sources of poor code quality and the effort required to fix different vulnerabilities.

2012 ACM Subject Classification Security and privacy → Software and application security; Software and its engineering → Collaboration in software development; Information systems → Open source software; Security and privacy → Vulnerability management

Keywords and phrases Open-source software, Software quality, Cybersecurity, Repository Mining

Digital Object Identifier 10.4230/OASIcs.ICPEC.2024.16

Funding This work is partially financed by Portuguese national funds through FCT – Fundação para a Ciência e Tecnologia, I.P., under the projects FCT UIDB/04466/2020 and FCT UIDP/04466/2020. Furthermore, the third author thanks the Instituto Universitário de Lisboa and ISTAR, for their support. We acknowledge funding for project LIONS by dtec.bw. Andrei-Cristian Iosif and Tiago Gasiba acknowledge the funding provided by the Bundesministerium für Bildung und Forschung (BMBF) for the project CONTAIN (FKZ 13N16585).

Acknowledgements The authors would like to thank Kaan Oguzhan for aiding in data collection, and also for the helpful, insightful, and constructive comments and discussions about the present work.

1 Introduction

Having secure software has always been essential for developing any product or service. Additionally, cybersecurity has gained more attention in recent years due to the ever-increasing reliance on the internet. Providing secure products or services is essential to maintain customer and user trust. Moreover, it prevents potential damages resulting from security breaches. As such, it is important that the developers of the software be aware of security standards and best practices, e.g. through awareness campaigns or, in general, cybersecurity education.

Several industrial cybersecurity standards provide guidelines for the industry. One example is the IEC 62443 standard [5], which requires a secure software development life cycle (sSDLC) framework that covers all SDLC phases and provides detailed guidance to ensure software security. Another widely-used cybersecurity standard in the industry is the Common Weakness Enumeration (CWE) [12] developed by the MITRE Corporation, which is a community-developed list of both software and hardware vulnerabilities categorized by the type of vulnerability they introduce. The CWE also provides descriptions of the vulnerabilities and usually includes examples of both vulnerable and non-vulnerable code associated with the vulnerability. Apart from cybersecurity standards, there are also standards for software quality, such as ISO 25000 [6], which provides a set of metrics for evaluating the quality of software products and guidelines for producing high-quality software.

According to the U.S. Department of Homeland Security (DHS) [2], humans are responsible for more than 90% of software quality issues.

As software complexity increases, the use of third-party components, including commercial off-the-shelf (COTS) and open-source software (OSS), has become more widespread. According to a survey conducted by Black Duck Software, 78 percent of respondents reported that their companies use OSS for some or all of their operations [1]. Although OSS can benefit developers, detecting the most critical quality violations can pose a challenge. Attackers view the widespread use of TPCs as an opportunity for exploitation. Notably, TPCs have been the source of significant security vulnerabilities, such as the FREAK OpenSSL vulnerability (CVE-2015-0204) [10], Shellshock Vulnerability (CVE-2014-6271) [9], and Log4Shell (CVE-2021-44228) [11]. As the trend of using third-party components continues to rise, it is important to consider security when selecting and utilizing TPCs. This paper aims to analyze existing OSS software on GitHub and shed light on the trends of security vulnerabilities.

Security-oriented companies may prioritize their own interests when discussing software vulnerabilities, which in turn results in fewer studies with unbiased real-world data. We contribute to improving the understanding of cybersecurity issues in OSS software and providing insights for developers and researchers. In this work, we highlight the difficulty of identifying OSS software's most frequent security issues.

In this work, by exploring GitHub repositories and commits between 2009 and 2022, we aim to gather insight into the following Research Questions:

RQ1: What are the most common security vulnerabilities in OSS software?
RQ2: How do these vulnerabilities vary by programming language and over time?
RQ3: What is the effort required to fix these vulnerabilities in terms of code changes?

To conduct our analysis, we scraped publicly available GitHub repositories, identified commits aimed at fixing known software vulnerabilities by looking at their commit messages, and analyzed trends and distributions of CWE categories. Our findings can benefit development teams and researchers seeking to improve software quality by exploring the most common sources of poor code quality. Therefore, in this paper, we present a summary of the most common security vulnerabilities and their frequency of occurrence in OSS software.

This paper is structured as follows: Section 2 will introduce relevant related research. Next, we present our approach in Section 3, followed by results in Section 4. We discuss our results in Section 5. Next, Section 6 explores the threats to validity to our study. Finally, Section 7 reiterates through our work and explores possible next steps.

2 Related Work

In this section, we present similar large-scale studies which explore security through repository metadata.

Iannone et al [4] explored the effects of refactoring on security by measuring security-related technical debt. Their large-scale investigation involved running git-blame commands to analyze commits, providing insights into the correlation between refactoring practices and security implications in software development.

Li and Paxson [7] performed a comprehensive empirical study of security patches, analyzing a diverse set of repositories to generate generalizable insights about security practices. They utilized Git commit links to identify and review security-related changes, contributing significantly to understanding security patch dynamics.

Wang et al. [18] developed PatchDB, a large-scale security patch dataset to facilitate the manual checking of security-related commits. Their work highlights the consistency and challenges in identifying security-focused changes across large datasets.

Wang and Nagappan [17] characterized software developer networks by conducting a large-scale empirical study to distinguish between security and non-security-related commits. Their findings provide insights into the network dynamics of developers engaged in security-related software development.

While all these approaches explore repository metadata, our work tackles a purely quantitative analysis of security-related git commit messages across OSS repositories, where we explore: language-specific vulnerability analysis, SAST Tool integrations, and commit message quality.

3 Approach

In this section, we provide details about the steps taken to collect the data driving our study. Our search space includes all public GitHub repositories created between January 1, 2009, and December 31, 2022. This arbitrary cut-off date is motivated by recent developments in AI-assisted programming – according to a Microsoft executive's statement from March 2023, 40% of GitHub Copilot users check in "AI-generated and unmodified" code [8]. Hence, we purposefully restrict our search space to include *mostly human* code changes.

3.1 Querying and Scraping GitHub for Repositories

To gather the necessary data for our study, we developed a scraper to query GitHub for repositories. The scraper searched for mentions of "CWE ID" in commit messages to identify repositories containing fixes for vulnerabilities listed in the CWE catalog.

Due to limitations on the branch type for queries, we could only search for commits on the master/main branch. We also employed a filter to exclude commits that contained more than a single file change, as this helped to ensure that the commit was not a general commit that happened to include a fix for a vulnerability. Changes on non-code files such as `README.md` or `LICENSE` were not counted towards the one file change limit.

To parallelize the scraping process, we use Terraform to deploy multiple instances of our scraper on the cloud. We selected instances optimized for high network bandwidth, with a reported capacity of *up to 10 Gigabit*[1], as well as 8GB of RAM and four vCPUs. These specifications helped ensure the instances could efficiently clone and analyze repositories with high commit volumes.

[1] Available instance bandwidth reported by Amazon AWS

We conducted our analysis by running 13 instances over approximately seven days, each assigned to scrape a different subset of repositories. We faced challenges with GitHub API's query limit, which restricts results to the first 1000 entries per query. We implemented 6-hour query intervals to mitigate this, allowing us to cover most of the relevant repositories. Despite the time windowing approach, some commits were still missed, but we deemed this an acceptable trade-off between the number of repositories we could scrape and the time it took to scrape them.

3.2 Data Extraction

Once our scraper finished detecting all candidate repositories within the given year range, it began cloning the repositories and analyzing their commits. We applied strict rules to classify a commit as a "vulnerability fixing commit." These rules required a commit message containing the strings "fix" and "CWE ID," where ID is any possible number in the CWE catalog. We also imposed a limit of 1 file change per commit, excluding changes on non-programming-language files such as `Readme.md` or `License`. These restrictions were implemented to keep the results as relevant as possible. For each commit detected as fixing a CWE vulnerability, we kept small metadata about the commit, date, repository, and the before and after versions of the file(s) that were changed.

3.3 Analysis Method

In this section, we will describe our data analysis approach. We processed the scraped data to extract insights such as the popularity of CWE vulnerabilities across different programming languages and the number of commits per year. Our analysis involved plotting the results for visualization and applying data manipulation techniques such as grouping and sorting to determine the frequency of CWE vulnerabilities by programming language and year.

We analyzed whether some vulnerabilities were more common than others, whether the frequency of specific vulnerabilities varied over time or varied by programming language, and whether some programming languages were more prone to certain types of vulnerabilities. Additionally, we statistically analyzed the number of lines changed in the commits to fix specific vulnerabilities and their variation between different CWEs.

Regarding limitations, it is worth noting that our results are based solely on the commit messages of the commits. This approach may result in some limitations in the accuracy of our data, as we would miss all commits that don't mention the fix in the commit message. However, we believe this limitation is acceptable, given the size of our dataset and the constraints of our approach. The results of our analysis are presented in the next section.

4 Results

In this section, we present the results of our analysis of the commits aimed at fixing CWE vulnerabilities in open-source repositories found on GitHub. By focusing on repositories created from 2009 onwards, we were able to acquire 1934 repositories that contained at least one commit aimed at fixing a CWE vulnerability. Overall, we observed 7093 such commits, as shown in Table 1.

Upon analyzing public GitHub repositories, we found that some programming languages have received more Common Weakness Enumeration (CWE) vulnerability fixes compared to others. Notably, the languages *C*, *Java*, and *C++* received the highest number of CWE vulnerability fixes, followed by *JavaScript* and *Python*. The number of CWE vulnerability fixes for each language is shown in Table 2.

Table 1 Number of GitHub repositories containing at least one commit aimed at addressing CWE vulnerabilities, and the overall count of commits dedicated to fixing CWE vulnerabilities on the platform.

Type	Count
Number of Repositories	1934
Number of Commits	7093

Table 2 A breakdown of the top programming languages that were detected in the commits, along with their corresponding overall percentages. Only the top 5 languages are shown.

Programming Language	Count
C	1784
Java	772
C++	547
JavaScript	301
Python	265

Table 3 Top-5 popular programming languages according to multiple sources for the year 2022.

Rank	Octoverse [3]	TIOBE [16]	RedMonk [14]
1	JavaScript	Python	JavaScript
2	Python	C	Python
3	Java	Java	Java
4	TypeScript	C++	PHP
5	C#	C#	C#

In 2022, *JavaScript* and *Python* were the most popular programming languages, according to Octoverse, TIOBE, and RedMonk. Surprisingly, *C*, which is the top language for receiving CWE vulnerability fixes, is not even in the Top-5 of Octoverse and RedMonk. This indicates a significant disparity between the popularity of programming languages on GitHub and the number of security vulnerability-fixing commits. It is possible that developers using popular programming languages lack cybersecurity awareness or are fixing vulnerabilities without explicitly mentioning them in their commit messages. Notably, this analysis is limited to publicly available repositories on GitHub, and further investigation is necessary to comprehend the correlation between the popularity of programming languages and the number of CWE vulnerability fixes, as well as the security implications of language choice.

Additional analysis was conducted on the commits within GitHub repositories to determine basic statistics on the number of lines changed in each commit. The results indicated that the average number of lines changed in a commit was 112.89, while the median was 7, demonstrating that the distribution of the number of files changed is skewed to the right. Investigation revealed that the skew was due to several factors, including the use of XML files with named versioning in some repositories and Notebook-style coding. To minimize the impact of outliers on the results commits with more than 100 lines changed were excluded, shifting the mean and median to 13.86 and 6.00, respectively. Although this exclusion may have affected the accuracy of the analysis, it was necessary to reduce the impact of outliers.

4.1 Number of Commits with mentions of CWE per Year

Figure 1 shows the number of publicly available GitHub commits that aimed to fix software vulnerabilities and were tagged with the associated CWE ID. From 2009 to 2022, the number of commits steadily increased from about 20 to around 1000. This trend indicates that the awareness of cybersecurity is on the rise, and people are taking steps to address vulnerabilities in their code.

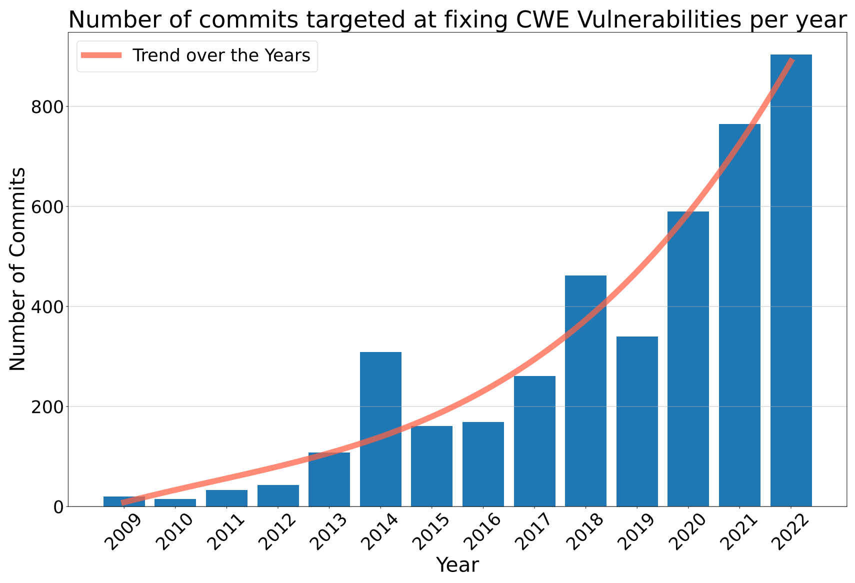

Figure 1 Trend in the number of commits made on publicly available repositories on GitHub, aimed at fixing software vulnerabilities and tagged with the associated CWE ID. The figure displays a clear upward trend in the number of commits over the years. Over the years, the number of commits has steadily increased from around 20 in 2009 to approximately 1000 in 2022.

However, our analysis suggests that the number of commits should be higher, especially in recent years. This may be due to two reasons: either people are fixing vulnerabilities without mentioning them or not enough people are addressing them. Additionally, our data collection method may have limited the number of commits we collected, which highlights the need for further research on this critical issue.

According to GitHub, there are approximately 28 million repositories, and assuming that all 800 commits collected are from different repositories, the percentage of repositories that had at least one software vulnerability fixing commit is only 0.003%. This suggests that our data collection method may not be exhaustive, and more commits may have been missed. Nevertheless, our study emphasizes the importance of addressing software vulnerabilities and encourages further research on this critical issue.

Alternatively, it is possible that people are indeed correcting vulnerabilities but not mentioning it specifically in their commits. This would explain the low number of commits, despite the increasing trend. It could be due to code analysis tools reporting vulnerabilities and people fixing them but do not mention the related CWE ID in their commits. Overall, the number of commits is increasing, which indicates that people are taking steps to address software vulnerabilities.

4.2 CWE per year

We repeat the trend analysis, this time only considering the top-10 CWEs.

Figure 2 shows a clear upward trend in the number of commits, indicating increasing cybersecurity awareness and action taken to detect, report, and fix vulnerabilities.

In 2018, a significant upward spike was observed, mainly due to a spike in CWE-772. The exact reasons for this spike are unclear, and further research is necessary to establish causality and determine the underlying reasons behind this observation.

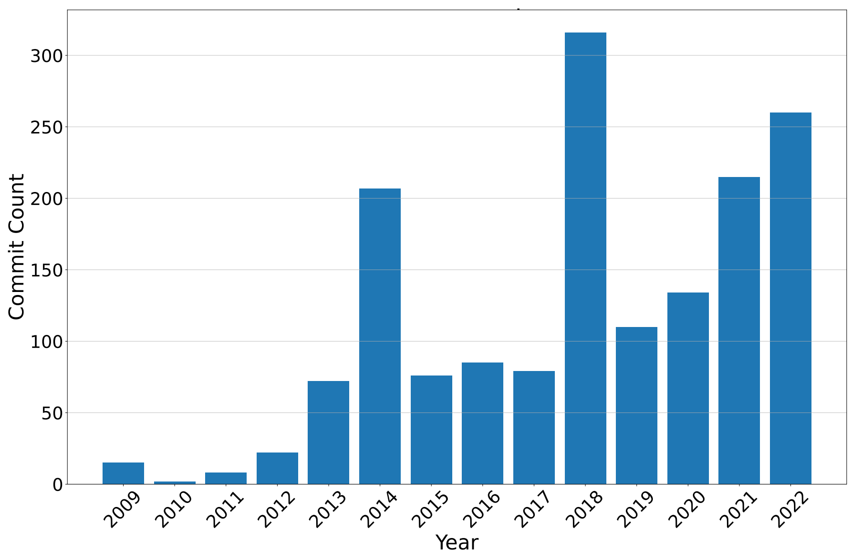

Figure 2 Number of commits between 2009 and 2022 targeted at fixing software vulnerabilities and tagged with the associated CWE ID, filtered for the top-10 CWEs.

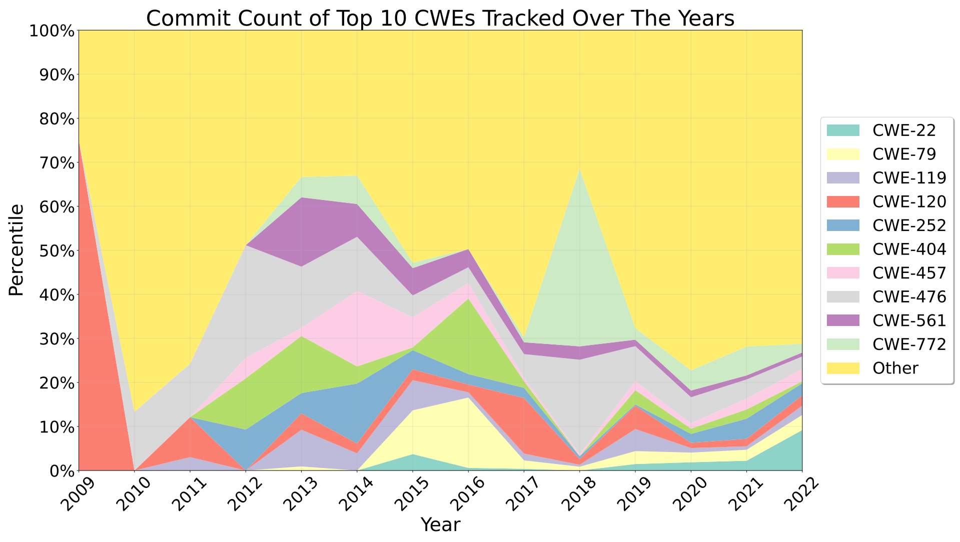

Figure 3 Percentile distribution of the Top-10 CWE categories from 2009 to 2022, along with the 'Others' category representing all remaining CWEs. The graph illustrates how the distribution of the Top-10 CWEs changes over time as a percentile. The graph provides valuable insights into whether the top-10 CWEs are decreasing or increasing over the years.

To the best of our knowledge, the spike could be due to high-profile cybersecurity incidents like Meltdown and Spectre or data scandals like Scandal, all of which may have led developers to review their code and fix vulnerabilities. Additionally, the General Data Protection Regulation (GDPR), which came into effect on May 25, 2018, may have contributed to an increase in vulnerability fixes, particularly those related to CWE-772.

We investigated whether the software development community is learning from the vulnerabilities present in the top-10 CWEs by grouping the top-10 CWEs and the remaining CWEs into eleven categories. Figure 3 suggests that the percentage of top-10 CWEs has been decreasing over time, while the "Others" category has been increasing. The significant spike observed in CWE-772 in 2018 is also visible in Figure 3.

The trends observed in Figure 3 indicate that lessons are being learned from the top-10 CWEs over time, and the software development community is focusing on addressing vulnerabilities other than the most common ones, which is positive. This suggests that the community is becoming more proactive in identifying and fixing vulnerabilities beyond the top-10 CWEs, which is crucial for improving software security.

Both Figure 2 and Figure 3 suggest that the software development community is becoming more effective in addressing vulnerabilities beyond the top-10 CWEs. Although the spike observed in CWE-772 in 2018 requires further investigation, the overall trends are positive and indicate proactive measures being taken to improve software security, which is also supported by Figure 1.

4.3 Programming Language vs CWE

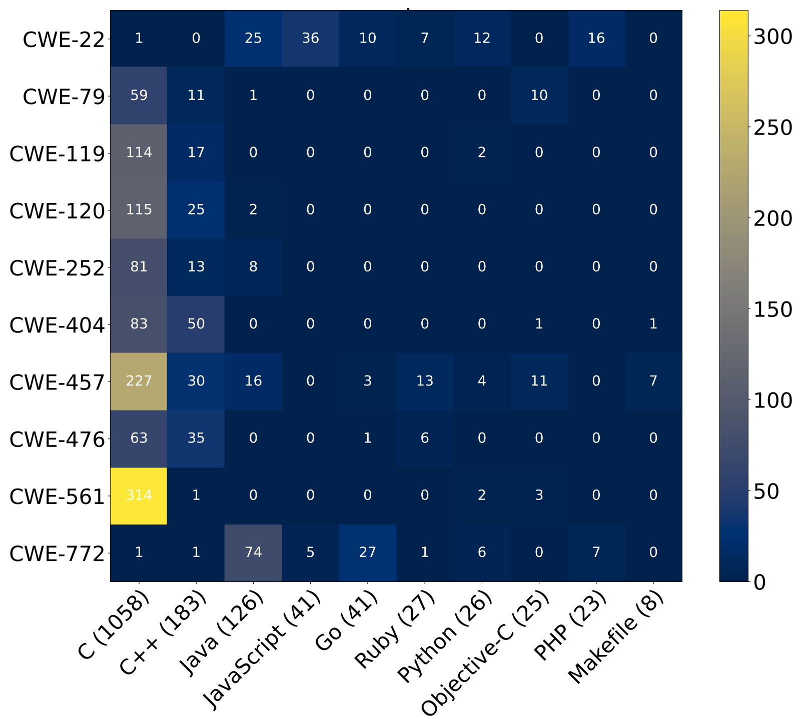

Figure 4 Commit count for the Top-10 CWEs across the Top-10 programming languages. The x-axis represents the programming language, the y-axis represents the CWE, and the color of the cells represents the number of commits.

The heatmap in Figure 4 shows commit counts for the Top-10 CWEs across the Top-10 programming languages. The x-axis represents the programming language, the y-axis represents the CWE, and the color of the cells indicates the number of commits. Results show that C has significantly more vulnerability fixes compared to other programming languages.

This could be due to C's popularity in low-level programming, which is more prone to vulnerabilities. C is also widely used in operating systems, embedded systems, and other low-level programming applications, contributing to its high number of vulnerability fixes. Additionally, counting all .h files as C files has resulted in fewer C++ files, making C++ the second-highest in number of commits for the Top-10 CWEs. This could be due to C++'s popularity in low-level programming combined with C, resulting in a majority of the Top-10 CWEs being attributed to C and C++.

The heatmap also highlights a noticeable trend of high counts for C and C++ for many CWEs, but a count of zero for other programming languages. This further supports the hypothesis that results are skewed towards C and C++ for the Top-10 CWEs. The presence of many zeros also suggests that some programming languages may be inherently immune to certain CWEs. For example, the common "Buffer Overflow" vulnerability (CWE-120, CWE-121, CWE-122) is common in C/C++, but is 0 for Python, Go, Ruby, and other languages.

4.4 Lines changed vs CWE

We analyze the number of lines changed for each of the Top-10 CWEs, to provide insight into the effort required to fix these vulnerabilities.

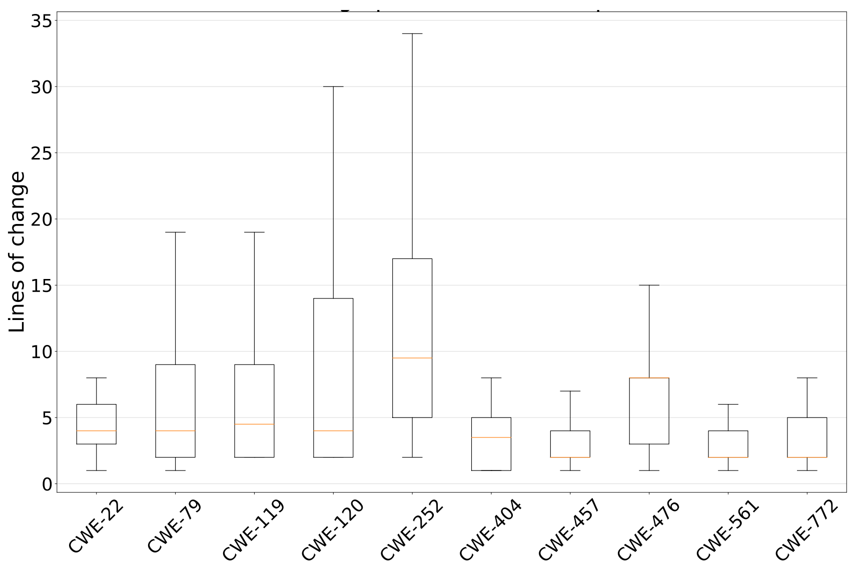

Figure 5 Number of code line that were changed to address a certain vulnerability (CWE).

Results from Figure 5 show that for 8 out of 10 CWEs, the mean lines of change is less than 5, with the exceptions being CWE-252 and CWE-476. The interquartile range (IQR) usually does not exceed 10. These findings indicate that addressing these CWEs can be done efficiently and effectively with relatively small changes to the codebase. The small IQR values suggest that the fixes are consistent across different instances of the same CWE. This indicates that there is a clear and standardized approach to addressing these vulnerabilities.

Overall, these findings provide valuable insights into the nature of common software vulnerabilities and how to address them effectively. However, it is important to note that this data alone is not sufficient to draw conclusions about the effort required for vulnerability detection and fixing. Additional research is necessary to investigate the required effort.

5 Discussion

In addressing our research questions, we explore our results to provide insights into the common security vulnerabilities, their variations, and the effort required to fix them. Next, we will discuss the results to provide an understanding of security vulnerabilities in OSS, their distribution, variation by programming language, evolution across time, and the effort required to mitigate them.

5.1 Implications for AI-Assisted Patching

Training machine learning (ML) algorithms to detect cybersecurity vulnerabilities is challenging due to the lack of a large and diverse dataset classified by experts [15]. ML applications require vast amounts of data for optimal performance, which is not always readily available.

To overcome the lack of diverse datasets, synthetically generated datasets such as Juliet [13] can be used. However, the use of synthetic datasets poses a potential risk where ML algorithms may learn characteristics of the dataset rather than vulnerability features, leading to data leakage towards the test set. To address this issue, it is important to evaluate the effectiveness of ML models on real-world examples rather than relying solely on synthetic datasets. Investigating the results of ML training on synthetic datasets and testing on real-world data to detect cybersecurity vulnerabilities would be interesting in the future.

5.2 Commits with CWE

The analysis shows an increase in commits with CWE over time, indicating a rise in awareness of software vulnerabilities and more software fixes. Future research could investigate factors contributing to this trend, such as the adoption of secure coding practices, improved vulnerability scanning tools, and cybersecurity training for developers. Examining the correlation between commits aimed at fixing vulnerabilities and repository numbers would yield valuable insights into the effectiveness of current practices and policies in mitigating software vulnerabilities.

5.3 Programming Language vs CWE

The results suggest that certain programming languages are more prone to specific CWEs than others due to their design and nature. C and C++ are susceptible to buffer overflow vulnerabilities because of their low-level nature and use in operating and embedded systems. In contrast, Python is immune to buffer overflow CWE-120 vulnerability, provided third-party libraries are not used. Developers should consider the programming language and its more prevalent vulnerabilities when designing and developing software. By understanding the likelihood of different vulnerabilities in different programming languages, developers can mitigate risks and ensure the security and integrity of their software. Staying up-to-date with the latest vulnerabilities and security trends is also important, as new vulnerabilities can emerge and existing ones can evolve over time.

5.4 Lines changed vs CWE

Overall, the average number of changed lines is usually less than 5, implying that the fixes are simple and can be done without introducing big changes to the codebase. Even for complex CWEs, such as CWE-120 and CWE-252, average number of changed lines is still relatively low. It is worth noting that some vulnerabilities may require more time to address their root cause, but overall, when the vulnerability is detected, the fix can be done efficiently.

6 Threats to Validity

We base our results solely on commit messages, which may limit data accuracy. It is possible that changes were collected, which should not have been accounted for due to bad comments, or that changes in our scope were missed, due to missing commit message information.

Throughout the data collection, we used a strict filter that specifically checks for the mention of CWE in the commit messages to eliminate false positives. However, this approach also has limitations. Specifically, we may have discarded changes relevant to vulnerability fixes that did not mention the related CWE-ID in the commit message. As a result, the data on GitHub may contain more vulnerability-fixing commits than what we captured, but we were not able to detect them due to the filtering process. Although we aim to increase the quality of our data by discarding changes that do not specifically mention a CWE-ID, this may result in a smaller sample size. Therefore, future studies could explore alternative methods for collecting data on vulnerability fixes in software development, such as scraping and labeling the code directly with static analysis tools.

7 Conclusions and Future Work

Software quality is crucial and must be addressed throughout the software lifecycle. The increase in commits targeting cybersecurity flaws in OSS repositories indicates developers are more aware of the need to address security concerns in software development. Evolving security standards mandate the implementation of secure software development processes, such as the secure Software Development Life Cycle (sSDLC) outlined in the IEC 62443 standard, to improve software quality. However, the increasing usage of OSS brings unknown vulnerabilities into one's own software. Our analysis shows that the awareness of software vulnerabilities in OSS is increasing over the years but still low, suggesting that there is room for improvement.

We found a correlation between the programming language and the type of vulnerabilities, indicating that developers should consider the strengths and weaknesses of the programming language in terms of security. By understanding the prevalent vulnerabilities in different programming languages, developers can mitigate risks and ensure the security and integrity of their software. Staying up-to-date with the latest vulnerabilities and security trends is also crucial as new vulnerabilities can emerge and existing ones can phase out over time.

Once vulnerabilities are detected, our research shows that fixes can be done efficiently without introducing significant changes or rebasing to the codebase. Our database of fixes can be used to cross-check the results of Static Application Security Testing (SAST) tools, to check whether software quality problems are detected by the tool and if vulnerabilities found match the mentioned CWE-ID in the commit message.

By outlining the findings and challenges associated with our large-scale data acquisition approach, our research also contributes to understanding the quality of data required for training machine learning algorithms, as data quality can significantly impact algorithm performance. Homogeneous and complete datasets are essential for training models that can accurately detect and classify software security vulnerabilities.

In a future work, the authors would like to extend their analysis by exploring a comparative analysis with post-2022 commits. Additionally, we would like to explore how the acquired data can be leveraged for AI-assisted vulnerability detection and classification.

References

1 North Bridge / Blackduck. `https://tinyurl.com/blackduck2k15`. [Accessed: 24 Apr. 2024].

2 Department of Homeland Security, US-CERT. Software Assurance. Online, Accessed 27 September 2020. URL: `https://tinyurl.com/y6pr9v42`.

3 GitHub Octoverse. Top programming languages in 2022. `https://tinyurl.com/octoverse2k22`, 2022. [Accessed: 4 Apr. 2024].

4 Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, and Fabio Palomba. Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. *Empirical Software Engineering*, 28(4), May 2023. `doi:10.1007/s10664-023-10287-x`.

5 International Electrotechnical Commission. IEC 62443-4-1 – Security for industrial automation and control systems – Part 4-1: Secure product development lifecycle requirements. Technical report, International Electrotechnical Commission, Geneval Switzerland, January 2018.

6 International Organization for Standardization. ISO/IEC 25000:2014 – Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. Technical report, International Organization for Standardization, Geneva, CH, March 2014. URL: `http://iso25000.com/index.php/en/iso-25000-standards`.

7 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*, CCS '17, pages 2201–2215, New York, NY, USA, 2017. ACM. `doi:10.1145/3133956.3134072`.

8 Microsoft. Morgan Stanley Technology, Media & Telecom Conference - FY2023. `https://tinyurl.com/ynepy7jw`, 2023. Accessed: 15. Apr. 2024.

9 MITRE. CVE-2014-6271. `https://tinyurl.com/4dk6yfzp`. [Accessed: 15 April 2024].

10 MITRE. CVE-2015-0204. `https://tinyurl.com/3prfckfj`. [Accessed: 15 Apr. 2024].

11 MITRE. CVE-2021-44228. `https://tinyurl.com/2dejmr3e`. [Accessed: 15 Apr. 2024].

12 MITRE. Common Weakness Enumeration. `cwe.mitre.org`, 2023. [Accessed: 22 Apr. 2024].

13 National Security Agency Center for Assured Software. Juliet Test Suite C/C++ 1.3. `https://tinyurl.com/bdd9csvz`, 2023. [Accessed: 20 Apr. 2023].

14 Stephen O'Grady. The redmonk programming language rankings: June 2022. `https://tinyurl.com/4xpdr83z`, 2022. [Accessed: 20 Apr. 2024].

15 Kaan Oguzhan, Tiago Espinha Gasiba, and Akram Louati. How good is openly available code snippets containing software vulnerabilities to train machine learning algorithms? In *CYBER 2022, The Seventh International Conference on Cyber-Technologies and Cyber-Systems*, volume ISBN: 978-1-61208-996-6, pages 25–33. ThinkMind, 2022. [ISSN: 2519-8599].

16 TIOBE. Tiobe index. `https://tiobe.com/tiobe-index/`, 2023. [Accessed: 25 Apr. 2024].

17 Song Wang and Nachiappan Nagappan. Characterizing and understanding software developer networks in security development. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 534–545, 2021. `doi:10.1109/ISSRE52982.2021.00061`.

18 Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 149–160, 2021. `doi:10.1109/DSN48987.2021.00030`.