# Understanding the Practice of Security Patch Management across Multiple Branches in OSS Projects

Xin Tan*
Fudan University
China
18212010028@fudan.edu.cn

Yuan Zhang*
Fudan University
China
yuanxzhang@fudan.edu.cn

Jiajun Cao
Fudan University
China
20210240046@fudan.edu.cn

Kun Sun
George Mason University
United States
ksun3@gmu.edu

Mi Zhang
Fudan University
China
mi_zhang@fudan.edu.cn

Min Yang
Fudan University
China
m_yang@fudan.edu.cn

## ABSTRACT

Since the users of open source software (OSS) projects may not use the latest version all the time, OSS development teams often support code maintenance for old versions through maintaining multiple stable branches. Typically, the developers create a stable branch for each old stable version, deploy security patches on the branch, and release fixed versions at regular intervals. As such, old-version applications in production environments are protected from the disclosed vulnerabilities in a long time. However, the rapidly growing number of OSS vulnerabilities has greatly strained this patch deployment model, and a critical need has arisen for the security community to understand the practice of security patch management across stable branches. In this work, we conduct a large-scale empirical study of stable branches in OSS projects and the security patches deployed on them via investigating 608 stable branches belonging to 26 popular OSS projects as well as more than 2,000 security fixes for 806 CVEs deployed on stable branches.

Our study distills several important findings: (i) more than 80% affected CVE-Branch pairs are unpatched; (ii) the unpatched vulnerabilities could pose a serious security risk to applications in use, with 47.39% of them achieving a CVSS score over 7 (High or Critical Severity); and (iii) the patch porting process requires great manual efforts and takes an average of 40.46 days, significantly extending the time window for N-day vulnerability attacks. Our results reveal the worrying state of security patch management across stable branches. We hope our study can shed some light on improving the practice of patch management in OSS projects.

## CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Security and privacy** → **Software security engineering**.

*co-first authors


*WWW '22, April 25–29, 2022, Virtual Event, Lyon, France*

ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512236

## KEYWORDS

Security Patches, Patch Deployment Study, OSS Vulnerabilities

**ACM Reference Format:**
Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the Practice of Security Patch Management across Multiple Branches in OSS Projects. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3485447.3512236

## 1 INTRODUCTION

Open source software (OSS) plays an important role in all kinds of information systems, as well as the whole web infrastructure. As the development of vulnerability discovery techniques (e.g, AFL [1], Syzkaller [2]) and infrastructures (e.g, OSS-Fuzz [20], syzbot [3]), the number of disclosed OSS vulnerabilities is growing rapidly. According to a recent report [47], the number of disclosed vulnerabilities in OSS in 2020 has increased by 50%. OSS vulnerabilities have emerged as an increasingly severe threat. To fight against these N-day threats, it is particularly important to develop and deploy patches for the disclosed vulnerabilities in OSS.

Meanwhile, the OSS developers usually maintain one or more stable branches at the same time. Typically, when a major version is released, the developers may fork the version as a stable branch from the mainline. Then, they would continuously apply bug and security related fixes on the stable branches, including releasing patched versions from the stable branches, but would not make further feature updates. Such a practice enables users of old versions that may not always use the latest versions due to various reasons such as legacy features or stability requirements, to obtain the fixed applications from the corresponding stable branch.

Security patch management is an essential task when managing multiple stable branches. That is, to protect the users of old versions, a security patch should not only be deployed on the mainline, but also be ported to all stable branches that are also vulnerable to the vulnerability. However, since a security patch may not be directly applicable to another branch, the patch porting is usually done manually and consumes a huge amount of resources. The lack of resources and expertise presents a huge challenge for security patch management across stable branches. Hence, stable branches may take a long time to be fixed or never be fixed, exposing users of older stable versions to "N-day" vulnerability threats.

Among the existing works that investigate the security patch deployment and lifecycle, most of them only focus on the patching process on the mainline [18, 19, 25, 37]. In addition, there are few studies [15, 49] that explored the patch propagation from the upstream Android OSS projects (e.g., Android kernels, Android framework) to downstream vendors. However, the patch management and propagation across stable branches within OSS projects have not been well explored, particularly, due to the two challenges: ❶ it requires to identify stable branches from the more non-stable branches in the OSS code repository; and ❷ it requires to assess the deployment status of the security patch on each affected stable branch of a vulnerability. For the first challenge, there is no automated method to do so. For the second challenge, no existing work could automatically locate all the security patches deployed on different branches for the same vulnerability. In particular, there is a line of works that aim to identify bug-fixing commits [21–23, 29, 40, 43] or security patch commits [45, 46] in the code repository. However, since the patches may be customized during porting, these works cannot guarantee to link a specific vulnerability with all its security patches across stable branches. Further, the existing patch database [46] only provides one patch for a disclosed vulnerability, which is usually on the mainline.

In this work, we make the first attempt to study the security patch management across the stable branches in OSS. We pay great efforts to manually collect stable branches in OSS projects. Overall, we select 26 popular OSS projects as target software, collecting a total of 608 stable branches. These projects are written in 5 popular programming languages (i.e., C, C++, Java, PHP and Python) and belong to various application types (e.g., kernels, databases, libraries). For each OSS project, we collect all its associated vulnerabilities from NVD [31] and collect the related vulnerability information, especially, its security patches. Further, we develop a semi-automatic approach to identify the patches on each stable branch vulnerable to the collected vulnerabilities. In all, we collect 806 CVEs along with 2,099 patches on the stable branches.

With the collected dataset, we perform a study from the following four aspects: reporting the distribution and characteristics of stable branches, measuring the patch deployment status across stable branches, analyzing the unpatched branches to reveal the reasons of not being patched and potential security threats, and analyzing the patched branches to reveal the challenges and efforts in patch porting. Our overall findings are worrisome. Over 80% of CVE-Branch pairs are not patched. Apart from MongoDB, Wireshark and Suricata, most of the software under our investigation did a poor job on security patch management, with a patch ratio below 60%. We uncover that there are two main reasons for stable branches not deploying corresponding patches, namely, branches no longer being maintained and patch management issues. We also reveal that the patch porting process is complicated and takes an average of 40 days to complete. In 82.36% of ported patches, the original patches cannot be directly deployed and some adjustments are required.

In summary, we make the following contributions.

- *Large-scale Dataset.* We build a dataset of security patches deployed on different stable branches, including 2,099 security patches and 608 stable branches in 26 OSS projects.
- *Empirical Study.* We perform a deep study on the patch management practice across multiple stable branches. Our study reveals the poor security patch management across stable branches in OSS projects and motivates the open source community as well as the security community for improvement.

## 2 BACKGROUND

**Multi-branch Management of OSS.** Branching is a common practice in managing the development and maintenance of OSS, and it is widely supported by popular version control systems such as Git. In a version control system, a branch is initially created as a copy of the current code snapshot. Thereafter, developers could modify the code and the version control system will keep track of all these modifications. With code branching, developers can effectively manage the software development and maintenance process by managing multiple isolated and concurrent code branches. According to the development model, there are usually multiple branches in an OSS project. Among all these branches, there is a main development branch (usually called *master* branch in Git) where developers add new features, and the other branches serve different purposes. By investigating the code repository of several popular OSS, we observe the following three specific purposes:

- *Stable branch.* Stable branches are used to maintain the released stable versions. When a new minor version (defined in semantic versioning [13]) is officially released as a stable version for users, a corresponding stable branch is created in the code repository. After that, the developers mainly work on the stable branch to fix security issues and publish patched versions (in semantic versioning [13]). Besides, a few software also apply small feature updates on the stable branch.
- *Temporary branch.* They are created to develop specific features, fix specific issues, or perform software testing separately. After the specific task on the temporary branch completes, all the new code changes should be merged into other branches (such as master branch) and the temporary branch may be removed.
- *Mirror branch.* A mirror branch is a copy of one codeline at a certain point. Unlike the stable branches, there will be no code changes on the mirror branch after the branch point.

Tagging, the ability to tag a specific point in a repository's history (such as a commit), is one important operation in version control systems. A tag is a static code snapshot of the project while a branch is a codeline that changes dynamically. Typically, developers use tagging to mark release points and make snapshots of the project as released code versions. Our measurement takes the branches as the research targets to understand when and how security patches have been deployed throughout the history of software.

**Patch Management across Stable Branches.** To make all stable branches immune to known vulnerabilities, once a vulnerability is reported, developers should deploy the security patches to all the affected stable branches quickly. Instead of developing separate patches for each branch, developers tend to develop a patch on one branch (e.g., the master branch) first and then port it to other branches. During the patch porting process, the developers may need to make minor changes to the original patch and submit it to the corresponding stable branch. After the patches are ported to all affected stable branches, all users are able to access a patched version for their used OSS versions and update their OSS to this version on their machines.

Ideally, the security patch should be ported to all vulnerable stable branches. However, patch porting may take hours, days or even months, resulting in longer attack windows against stable branches. To the best of our knowledge, no measurement has been performed to explore the patch management practice across stable branches in OSS projects. Therefore, our work is dedicated to performing the first measurement to reveal the patch deployment status on stable branches and understand the patch management practice in the multi-stable-branch scenarios.

## 3 DATA COLLECTION

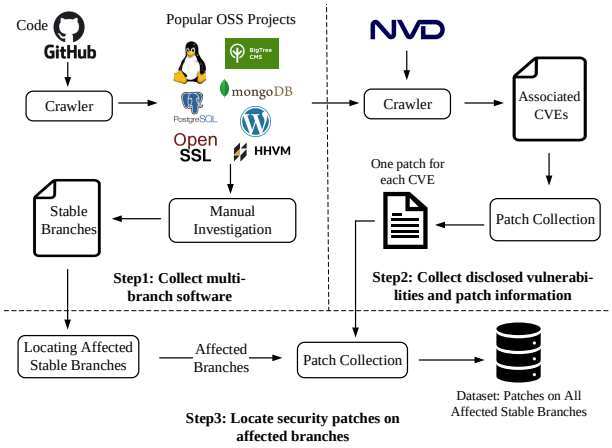


**Figure 1: The Overview of the Data Collection Process.**

An extensive dataset constructs the basis for our study. As shown in Figure 1, we follow three steps to construct the dataset. First, we collect some OSS projects that manage multiple stable branches. Second, we collect some disclosed vulnerabilities associated with these software and try to locate one security patch for each vulnerability. Third, based on a given patch, we try to collect other security patches on all affected stable branches. In the following, we present the details of each step.

**Step 1: Collecting Multi-branch Software.** We collect OSS and their code repositories from GitHub [6], because it contains millions of OSS projects. We mainly consider OSS projects written in C/C++/Java/PHP/Python due to their popularity. At first, we collect popular OSS projects in GitHub by picking the top 1,000 popular projects for each language and referring to some popular software ranking lists [44]. Then, we query the NVD to get the number of CVEs for each software, and only keep those projects with more than 24 CVEs (to balance the number of the kept OSS projects and the quantity of CVEs for each kept project). For these projects, we manually determine if the project maintains multiple stable branches in its repository. As described in §2, a code repository may contain many branches with different purposes. We manually identify the stable branches from two aspects. First, we review the official website of the software projects to find the documentation about the different purposes of code branches. Second, we analyze the branch itself, including reviewing the submitted commits and inspecting the related documentation (e.g., README in the branch), to determine the branch's role. If the roles of the branches cannot be determined by any of these methods, we discard this software. Following the manual analysis, we select 72 OSS projects that manage at least two stable branches each.

**Step 2: Collecting Disclosed Vulnerabilities and Patch Information.** We collect the disclosed vulnerabilities for an OSS project from NVD. For each vulnerability, NVD provides not only the vulnerability description and relevant external references but also much additional information, including all potentially vulnerable software versions under the Common Product Enumerator (CPE) [34], the class of weakness under the Common Weakness Enumeration (CWE) [14], and the vulnerability severity under the Common Vulnerability Scoring System (CVSS) [33]. We use the NVD XML dataset [32] collected on August 25th, 2021 as the source.

To locate all the disclosed vulnerabilities associated with an OSS project, we leverage the CPE information provided by the NVD. We scan each NVD entry disclosed from January, 2017 to August, 2021 and extract the potentially affected applications from the CPE Applicability Statement. If the affected applications contain the name of an OSS that is collected in the *Step 1*, we associate the vulnerability with this OSS.

After collecting the associated vulnerabilities, we use a web crawler and manual analysis to locate the security patches of these vulnerabilities. Several researchers [25, 45] observe that some URLs among the external references in NVD may point to particular patch commits that fix the security vulnerability in the repository. Therefore, we develop a web crawler to identify all the commit-like URLs in NVD entries and fetch the commits from the code repositories. Since such a method may introduce some non-patch commits, these crawled commits are further manually re-validated to guarantee the correctness. In all, we manually inspect 446 commits and confirm 425 of them as patch commits and the other 21 commits as non-patch commits, involving about 20 man-hours. When there is no patch information located by the crawler for some CVEs, we manually locate the patches for them, using ways such as referring to vulnerability reports of other sources, search engines.

To keep our study unbiased, we further filter out the OSS that have too few stable branches or too few disclosed vulnerabilities with located patches. Specifically, we select the target software from our collected software based on two criteria: i) the software must contain at least 4 stable-branches in its source code repository; ii) the software must have at least 20 disclosed vulnerabilities with patches located in our dataset. Furthermore, if a software has a large number of vulnerabilities, we only randomly select 50 vulnerabilities. Overall, we select 26 OSS projects from the 72 ones picked in the *Step 1* and collect 806 CVEs from them.

**Step 3: Locating Security Patches on Affected Branches.** After obtaining one patch for each vulnerability, we devise a semi-automated method to locate the security patches on all affected stable branches. Our method is based on two important observations. First, patches that fix the same vulnerability on two different branches may have similar code diff or commit messages. Second, when porting a patch to another branch, developers tend to mention the commit ID of the ported patch in the commit message. For example, when a developer uses the cherry-pick command in Git to port a patch commit from one branch to another, he or she often adds a line after the original commit message that says "cherry

picked from commit some-commit-id" to let others know where this commit comes from.

Based on these observations, we first automatically identify potential patches for a specific vulnerability on each affected stable branch and then manually verify them. In general, our semi-automated method consists of three steps.

(1) We determine the affected versions of the vulnerability from the CPE information provided by NVD and associate them to the stable branches in the code repository.
(2) Given a reference patch commit (that has been located in *Step 2*), we scan all the commits in each affected branch and locate the patch commit for the target vulnerability using a heuristic-based method. As listed in Table 8 (in Appendix), we formulate 10 rules based on the above observations, which can be divided into two categories: i) rules that can directly locate patch commits and require no further manual inspection; ii) rules that identify potential patch commits and require further manual inspection.
(3) We manually verify the potential patch commits that are identified by the semi-automated method.

In all, three security researchers spend 150 hours on verifying the potential patch commits and successfully locate 1,905 patches. For cases that our semi-automated method fails to locate patches, we further resort to manual efforts for patch locating to ensure that no patches are missed in our dataset. To manually locate the patch on a given code branch, we first analyze the root cause of the vulnerability based on the reference patch commit and then carefully audit every commits on the given branch for patch locating. If no patch is found, we deem the branch as unpatched for the vulnerability. Following this way, we further locate 194 new patches that are missed by our semi-automated method. In total, we end up locating 2,099 security patches on the affected branches.

**Summary.** Following the three steps, we successfully construct a dataset, which consists of 26 popular OSS projects (covering 5 popular programming languages), 608 stable branches, 806 CVEs, and 2,099 patches on the stable branches. The details about these OSS projects can be found in Table 10 (in Appendix). Although we have tried our best to automate the data collection steps, there are still many cases that require manual efforts. In total, it takes 340 man-hours in constructing the dataset.

# 4 STUDY

Based on the large-scale dataset, we investigate the practice of security patch management across stable branches. In general, we study the following research questions.

- *RQ1: Stable-branch distribution and characteristics.* What is the distribution of stable branches in OSS? What are their characteristics in terms of maintenance time and code commits?
- *RQ2: Patch deployment status on stable branches.* Are the security patches for disclosed vulnerabilities properly applied to all stable branches? Is there a difference between OSS projects?
- *RQ3: Unpatched branches analysis.* What is the reason for not deploying the appropriate patches on stable branches? What security risks does this cause?
- *RQ4: Patched branches analysis.* To deploy patches across stable branches, what effort is required from the developers? How long does the patch porting process take?

## 4.1 Stable Branches (RQ1)

**Stable Branch Distribution.** In total, we collect 608 stable branches in 26 OSS projects. As shown in Table 10 (column 3), the distribution of the number of stable branches is uneven. The HHVM [7] owns over 100 stable branches while some OSS projects such as BigTree CMS [4] and phpMyAdmin [11] have less than 10 stable branches. The number of stable branches is mainly affected by the development speed and release frequency of the OSS project. For example, the development of HHVM moves fast and the development team releases a new major version every week [8], resulting in HHVM having the most stable branches. In contrast, Bigtree CMS releases a major version approximately once per year [5]. As a result, it owns few stable branches. Furthermore, the longer the software lives, the larger number of stable branches it has. Since the Linux kernel is almost 30 years old, it has the second most stable branches out of 26 collected OSS projects.

**Stable Branch Characteristics.** For each stable branch, we investigate the submitted commits on them and their maintenance duration. We obtain the maintenance duration of each stable branch by subtracting the branch setup time from the time that the branch was last committed. As shown in Table 10 (the last column), most software maintain stable branches for a long time. In 20 out of 26 OSS projects, each stable branch is maintained for more than one year on average. In addition, there are 3 OSS projects (QEMU, Pillow, HHVM) that provide maintenance to the stable branches for a relatively short period, less than 3 months. We also measure the number of submitted commits on each branch during the maintenance duration. As Table 10 (column 7) shows, the average number of commits per stable branch varies considerably among software. We observe that a longer maintenance time of a branch does not mean more commits will be submitted on that branch. For example, QEMU [12] has more commits submitted on its stable branches than OpenEMR [10], though its maintenance duration is only a third of OpenEMR's. We suppose that the number of commits submitted on stable branches is influenced by a combination of factors such as software size, maintenance duration, and maintenance policy.

> **Finding 1.** Maintaining multiple stable branches is a common practice adopted in OSS projects. However, the practice of managing these stable branches differs significantly among these projects in various ways, such as branch count, maintenance period, maintenance frequency, predicating different practice in managing security patches among these branches.

## 4.2 Patch Deployment Status (RQ2)

**Overall Patch Ratio.** We first measure the deployment status of patches for disclosed vulnerabilities on all affected stable branches. During the dataset construction in §3, we have labeled the affected stable branches for all the 806 CVEs. The patch status of each CVE branch pair can be easily marked according to the presence of the patching information in our dataset. Table 1 presents the overall patch deployment results. From this table, we observe that the proportion of unpatched CVE-Branch pairs is high regardless of the programming language. In Java and PHP, more than 70% of CVE-Branch pairs are unpatched. In the other three languages, the situation is even worse, with more than 80% of CVE-Branch pairs

**Table 1: Overall Patch Deployment Status (RQ2).**

| Language | #CVEs | #CVE-Branch Pairs | #Patched Pairs | #Unpatched Pairs |
|---|---|---|---|---|
| C | 383 | 5,907 | 982 (16.62%) | 4,925 (83.38%) |
| C++ | 72 | 1,821 | 270 (14.83%) | 1,551 (85.17%) |
| Java | 60 | 391 | 108 (27.62%) | 283 (72.38%) |
| PHP | 208 | 2,240 | 618 (27.59%) | 1,622 (72.41%) |
| Python | 83 | 607 | 121 (19.93%) | 486 (80.07%) |
| **Total** | 806 | 10,966 | 2,099 (19.14%) | 8,867 (80.86%) |

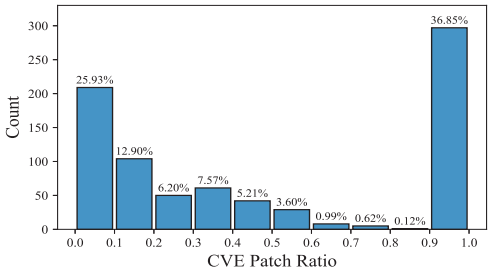


**Figure 2: CVE Patch Ratio (CVE-PR) Distribution (RQ2).**

unpatched. Our results show the patch deployment status on stable branches is worrisome and many stable branches are exposed to security risks.

> **Finding 2.** More than 80% of CVE-Branch pairs are unpatched, indicating there is much room for improvement in deploying patches across multiple branches.

**Analysis of Patch Ratio.** We further analyze the patch deployment status from the perspectives of per CVE, branch and OSS project, respectively.

❶ *Per CVE Perspective.* We define CVE Patch Ratio (CVE-PR) to measure how well a CVE is fixed on all affected branches.

$$\text{CVE-PR} = \frac{\#\ of\ patched\ branches}{\#\ of\ branches\ affected\ by\ the\ CVE}$$

As shown in Figure 2, a considerable number of CVEs (36.85%) have a CVE-PR higher than 0.9, which means they are properly fixed in affected stable branches. Meanwhile, there are also a large number of CVEs (57.82%) that have a CVE-PR lower than 0.5, which means they are not well fixed in affected stable branches. This phenomenon is somewhat similar to the Matthew effect [27, 28]. As a CVE is fixed on more stable branches, developers would realize that other affected branches should also be patched, and vice versa.

❷ *Per Branch Perspective.* We define Branch Patch Ratio (B-PR) to measure how well a stable branch is maintained against security vulnerabilities.

$$\text{B-PR} = \frac{\#\ of\ CVEs\ patched\ on\ this\ branch}{\#\ of\ CVEs\ affect\ this\ branch}$$

As shown in Figure 3, most stable branches have a low B-PR. In particular, 60.11% of stable branches have a B-PR lower than 0.1 and 74.24% of stable branches have a B-PR lower than 0.5. Only 13.06% of stable branches are patched for over 90% of CVEs. It clearly demonstrates that the patch management among most stable branches is far from good.

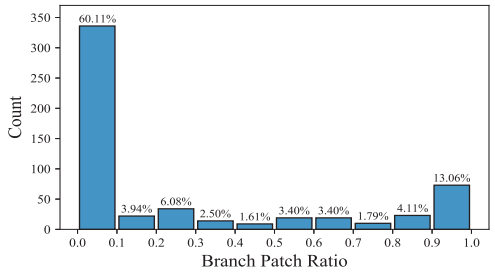


**Figure 3: Branch Patch Ratio (B-PR) Distribution (RQ2).**

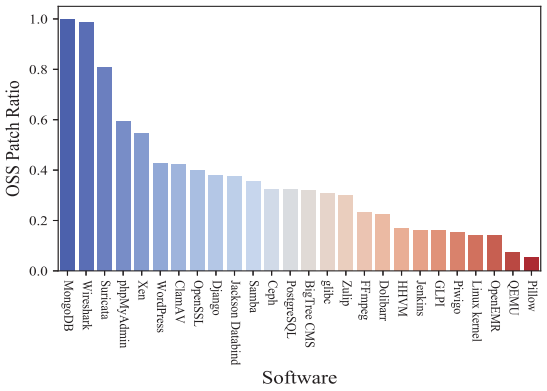


**Figure 4: OSS Patch Ratio (OSS-PR) Distribution (RQ2).**

❸ *Per OSS Perspective.* We define OSS Patch Ratio (OSS-PR) to measure the overall status of patch management on all the stable branches in an OSS project.

$$\text{OSS-PR} = \frac{\sum\ B\text{–}PR\ of\ all\ stable\ branches\ in\ this\ OSS}{\#\ of\ stable\ branches\ in\ this\ OSS}$$

As shown in Figure 4, the OSS-PR varies significantly, demonstrating a mixed picture of security patch management practice among different OSS projects. MongoDB and Wireshark have an OSS-PR close to 1 and Suricata has an OSS-PR over 0.8. However, the OSS-PRs for the remaining 23 projects are all below 0.6. Our results motivate the developers of these OSS projects to pay more attention to security patch management across stable branches.

> **Finding 3.** Security patch management across stable branches varies considerably from software to software. The OSS-PRs are below 60% for 23 out of 26 OSS projects, while only 3 projects have OSS-PRs above 80%. From the perspective of stable branches, 72.24% of the stable branches have B-PRs below 50%, indicating that most branches are poorly maintained. Besides, we observe polarized distribution in the CVE-PR, indicating that some CVEs are patched well across stable branches, but some are poorly managed.

## 4.3 Unpatched Branches (RQ3)

Through our study, we find a large part of stable branches have not been patched yet for some CVEs, making the users of these software versions at risk. We intend to uncover why these branches are not patched and what security threats are posed to users.

**Reasons for Not Patching.** We investigate all the CVE-Branch pairs that have not been patched and conclude two primary reasons.

**Table 2: Distribution of Reasons for Not Patching (RQ3).**

| Language | #CVE-Branch pairs | R1 | R2 |
|---|---|---|---|
| C | 4,925 | 4,631 (94.03%) | 294 (5.97%) |
| C++ | 1,551 | 1,156 (74.53%) | 395 (25.47%) |
| Java | 283 | 262 (92.58%) | 21 (7.42%) |
| PHP | 1,622 | 1,469 (90.57%) | 153 (9.43%) |
| Python | 486 | 478 (98.35%) | 8 (1.65%) |
| **Total** | 8,867 | 7,996 (90.18%) | 871 (9.82%) |

- *R1: out of maintenance branch.* Though the stable branch is vulnerable to the CVE, the branch is out of maintenance when the vulnerability is disclosed.
- *R2: patch management failure.* The stable branch is vulnerable to the CVE and is still maintained at the vulnerability disclosure. However, the corresponding patch is not applied to the stable branch due to poor patch management.

For a given unpatched CVE-Branch pair, the reason for it being unpatched can be pinpointed by analyzing the maintenance time of the stable branch. Specifically, we obtain the disclosure time of CVE from the CVE website. If the disclosure time is later than the last commit of the stable branch, we attribute the patch is not applied due to out of maintenance branch. Otherwise, if the disclosure time is earlier than the last commit, we attribute the patch is not applied due to patch management failure.

Following the above categorization, we analyze the reasons for the 8,867 unpatched CVE-branch pairs, and present their not-patched reasons in Table 2. Regardless of the programming language, over 70% of unpatched CVE-branch pairs are caused by out-of-maintenance branches. Especially, the percentage in Python even reaches 98.35%.

> **Finding 4.** Though a lot of branches are stable branches, they are actually out of maintenance when some vulnerabilities are disclosed, causing 90.18% of unpatched CVE-branch pairs.

Failures in patch management also led to a considerable number of unpatched CVE-branch pairs (around 10%). We further analyze the scope of these patch management failures. As shown in Table 3, patch management failures are a common phenomenon, which occurs in 80.77% of software. In Python, only a small number of stable branches (9.09%) are properly managed. Whereas in C++, patch management failures can be found in 59% of stable branches.

> **Finding 5.** The patch management failure occurs in a considerable number of OSS (80.77%).

**Table 3: Distribution of Patch Management Failures (RQ3).**

| Language | #Patch Management Failures | #Involved CVEs | #Involved Stable Branches | #Involved OSS |
|---|---|---|---|---|
| C | 294 | 122 (31.85%) | 72 (28.12%) | 10 (90.91%) |
| C++ | 395 | 39 (54.17%) | 70 (59.32%) | 2 (66.67%) |
| Java | 21 | 18 (30.00%) | 7 (21.21%) | 2 (100.00%) |
| PHP | 153 | 44 (21.15%) | 35 (32.41%) | 5 (71.43%) |
| Python | 8 | 7 (8.43%) | 4 (9.09%) | 2 (66.67%) |
| **Total** | 871 | 230 (28.54%) | 188 (33.63%) | 21 (80.77%) |

**Concerns of Patch Management Failure.** Since users prefer to fetch OSS projects from stable branches that are still under maintenance, patch management failures on stable branches are more likely to expose users to security threats. We try to infer the factors that cause the 230 not-fully-patched CVEs (see column 3 of Table 3) due to patch management failures from three aspects: vulnerability type, vulnerability severity and vulnerability exploitation. In particular, we use the CWE and CVSS Score in the NVD as the vulnerability type and vulnerability severity information respectively. To assess the likelihood of a vulnerability being exploited, we manually collect the PoCs for these 230 CVEs with the search engine. We present the overall results in Table 4.

**Table 4: Statistics of the Unpatched CVEs (RQ3).**

| Language | #Unpatched CVEs | #CWEs | Avg. CVSS Score | Median CVSS Score | #Published PoCs |
|---|---|---|---|---|---|
| C | 122 | 29 | 6.85 | 6.50 | 23 |
| C++ | 39 | 24 | 7.81 | 7.50 | 1 |
| Java | 18 | 10 | 5.39 | 4.85 | 0 |
| PHP | 44 | 19 | 7.06 | 6.50 | 22 |
| Python | 7 | 4 | 6.77 | 6.50 | 0 |
| **Total** | 230 | 54 | 6.93 | 6.50 | 46 |

❶ *Vulnerability Type.* As shown in Table 4, these unpatched CVEs cover a wide spectrum of vulnerability types. In addition, we find that the vulnerability type is naturally related to programming language. As shown in Table 9 (in Appendix), the top 5 vulnerability types for each language vary greatly and some of them may lead to serious consequences. For example, in C++, the top two vulnerability types are Out-of-bounds Read (CWE-125) and Out-of-bounds Write (CWE-787) which can be leveraged to bypass the ASLR protection or perform control flow hijacking.

❷ *Vulnerability Severity.* From Table 4, we find the average CVSS Score for the unpatched CVEs is 6.93 (7.0 is the bar for high-severity vulnerabilities in CVSS Score). We also present the severity distribution of these CVEs in Figure 5 (in Appendix). It shows that 33.48% of the unpatched CVEs belong to high severity (7.0-8.9) and 13.91% of them belong to critical severity(9.0-10.0). Surprisingly, we find the vulnerability severity of these unpatched CVEs is related to their programming language. As shown in Table 4, the average CVSS Score of unpatched CVEs in C++ and PHP is higher than 7.0, far exceeding that of Java.

❸ *Vulnerability Exploitation.* We are surprised to find that we can successfully find PoCs for 46 CVEs (as shown in Table 4), by taking only 12 man-hours. This means that attackers are able to attack applications that are built upon these unpatched stable branches at a low cost.

> **Finding 6.** 47.39% of the vulnerabilities that are not patched on all affected stable branches are of high or critical severity. For 46 (20%) of these unpatched vulnerabilities, there are publicly available PoCs.

## 4.4 Patched Branches (RQ4)

We conduct another study to understand the patch porting process across stable branches in OSS, especially, the efforts required for

developers during patch porting. To perform this study, we identify the original patch developed for each CVE and the patches that were ported to the other branches. Considering that patches are not always firstly developed on the mainline, we identify the first patch for each CVE according to the commit dates in the code repository of all its patches (on different branches). Among the remaining patches, some of them are directly inherited from the first patch via code forking, meaning they have the same commit ID as the first patch; the others are patches that are ported by developers. In all, we find 526 CVEs who have at least one ported patch, and collect 1,695 ported patches to study.

**Patch Porting Delay.** We calculate the patch porting delay of a vulnerability as the delta between the commit date of the original patch and that of the last ported patch. Figure 6 (in Appendix) shows the cumulative distribution function (CDF) for the patch porting lag (in days). From this figure, we find that 7.79% of CVEs may take more than six months to port patches. These long patching delays greatly extend the attack window for these vulnerabilities.

Table 5 presents the patch porting delays for vulnerabilities in different languages. From this table, we find that patch porting takes less than 4 days for PHP and Python, while more than 37 days in C and C++. We guess that Python and PHP developers take a more proactive attitude and more actions on patching vulnerabilities. Even worse, Java developers take an average of 223.75 days to finish the patch porting. We find the long patching delay of Java is mainly caused by Jackson Databind [9]. The developers of Jackson-Databind prefer to port several patches together to a stable branch in a single commit, sometimes delaying approximately one year after the patches are developed. These results indicate that OSS development teams should pay more attention to the patch porting process, especially for Java, C, and C++ projects.

**Table 5: Statistics of Patch Porting Delay (RQ4).**

| Language | Avg. Porting Lag | Median Porting Lag | Max. Porting Lag |
|---|---|---|---|
| C | 51.04 | 5.0 | 1,109 |
| C++ | 37.55 | 7.0 | 366 |
| Java | 223.75 | 194.5 | 632 |
| PHP | 3.74 | 0.0 | 157 |
| Python | 2.71 | 0.0 | 49 |
| **Total** | 40.46 | 1.0 | 1,109 |

**Finding 7.** For 23.19% of CVEs, it takes more than 30 days to complete the patch porting on stable branches, which greatly increases the possibility of being attacked.

**Patch Porting Mode.** By analyzing the ported patches, we observe two different modes in porting patches.

- *Mode-I: one patch at a time.* The developers port one security patch in a code commit.
- *Mode-II: multiple patches at a time.* The developers port multiple security patches together in a single commit.

We manually classify all the ported patches into these two modes and present the results in Table 6. Mode-II patch porting is only observed in C++, Java and PHP. We find that the patch porting mode affects the patch porting delay. In PHP, developers generally perform well in patch management, so there is no significant difference between the two modes. However, for C++ and Java, the patch porting lags of Mode-II are 3.2 and 4.4 times higher than those of Mode-I, respectively. This is because PHP developers typically merge and port multiple patches on a daily basis, while C++ and Java developers port multiple patches at longer intervals, greatly increasing the time delay in porting the earlier developed patches to other branches.

**Table 6: Distribution of Patch Porting Mode (RQ4).**

| Language | Mode-I | | | Mode-II | | |
|---|---|---|---|---|---|---|
| | Count | Avg. Lag | Mdn. Lag | Count | Avg. Lag | Mdn. Lag |
| C | 872 | 34.21 | 3.0 | 0 | / | / |
| C++ | 220 | 15.42 | 0.0 | 4 | 50.00 | 50.0 |
| Java | 8 | 76.25 | 62.0 | 7 | 336.43 | 326.0 |
| PHP | 457 | 0.84 | 0.0 | 47 | 0.00 | 0.0 |
| Python | 80 | 2.29 | 0.0 | 0 | / | / |
| **Total** | 1,637 | 21.01 | 0.0 | 58 | 44.05 | 0.0 |

**Patch Porting Efforts.** During the patch porting, sometimes the patch can not be directly applied which requires the developers to adjust the original patch a little. To understand the efforts that developers might put in porting patches, we compare the code between the original patch and the ported ones. We find that 82.36% of the ported patches (1,396 patches) are different from the original patches, while the other ported patches are exactly the same as the original ones in the code diff. We further study what kinds of code modifications should the developers make during patch porting. By reviewing these 1,396 ported patches, we find developers need to make four types of code changes.

- *Type-I: adjust patch positions.* When porting a patch to a given branch, the vulnerable code may be in a different file, a different function, or different code lines. As a result, the developers need to adjust the position of the patch statements to apply the patch.
- *Type-II: fit code context.* When the context of the vulnerable code on a branch differs from that of the patch-developed branch, directly applying the original patch may lead to semantic or syntax errors. Fortunately, the differences in this scenario do not require adjusting the vulnerability fix logic. The developers just need to modify the patch to fit a new code context, such as using the new namespace.
- *Type-III: change fix logic.* The code differences between the ported branch and the original patch-developed branch require adjusting the vulnerability fix logic.
- *Type-IV: irrelevant changes.* There are some irrelevant changes to vulnerability fixing, such as modifying comments and indents. These changes are not necessary during the patch porting.

Table 7 present the detailed results about the patch porting efforts. Note that there may be more than one type of changes made in a single patch. In 67.55% of ported patches, developers just need to adjust the positions of the patch statements without adjusting the vulnerability fixing logic. In 8.38% of patches, developers need to pay more effort in carefully adapting the original patch to fit the new code context. In 9.85% of patches, developers have to adjust the vulnerability fix logic due to the great code differences between two branches, which requires the most significant efforts. According to our inspection, Type-I changes are easy to automate while Type-II

**Table 7: Code Change Types in Ported Patches (RQ4).**

| Code Change Type | Count | Percentage |
|---|---|---|
| Type-I | 1,145 | 67.55% |
| Type-II | 142 | 8.38% |
| Type-III | 167 | 9.85% |
| Type-IV | 653 | 38.53% |

changes and Type-III changes are the two major challenges in patch porting. Our study motivates future research to ease these kinds of code changes in the patch porting process.

**Finding 8.** During patch porting, developers usually have to make necessary code changes to adopt the original patch. The major efforts during this process include adjusting patch positions, fitting code context and changing fix logic.

## 5 DISCUSSION

**Limitations.** There might be two limitations in our study. First, during the dataset collection, we assume the affected versions listed by NVD are correct and subsequently locate patches on the corresponding affected branches. However, recent works [17, 30] show that the affected versions provided by CVE/NVD are sometimes incorrect. The incorrect information may affect our study in two ways: 1) our dataset may have missed some affected stable branches; and 2) the branches that we have not located security patches may be not vulnerable. To the best of our knowledge, locating all affected versions of a specific vulnerability is still an open problem [16]. Second, if a vulnerability is reported when a stable branch is about to go out-of-maintenance, the project maintainers may intentionally choose to port the patch to the next branch rather than the to-be-out-of-maintenance branch. However, in §4.3, we cannot further investigate whether a patch management failure is intentional or unintentional, due to the lack of information about the maintenance decision process within the developer team. Nevertheless, we argue that either intentional or unintentional patch management failures pose security risks to end-users and should be avoided.

**Suggestions.** To improve the current poor status of security patch deployment, both the developers and the security community should pay more attention. Based on our study, we have the following suggestions. First, in §4.3, we find that some vulnerabilities are not patched probably because of management failures. Thus, we suggest that a patch deployment status monitoring mechanism should be established to avoid such failures. Second, as introduced in §4.4, patch porting is time-consuming and laborious, causing delays in patch deployment. To facilitate the patch porting process, automated patch porting tools [36, 38, 39, 42] should be incorporated and continuously improved.

## 6 RELATED WORK

**Bug-fixing Commits Identification.** Identifying bug-fixing commits is a well-received topic in mining software repositories (MSR). Firstly, Mockus and Votta [29] and Ray *et al.* [35] leverage keyword matching to recognize bug-fixing commits. Further, more features are considered by Wu *et al.* [48] and Sun *et al.* [40], such as the developer and the message of the commit. Recently, machine learning [43] and deep learning [21–23] have also been used to identify bug-fixing commits. In this paper, we aim to identify security patch commits of a specific vulnerability, which is somehow similar to identify the fixing commits of a bug report [40, 48]. Our approach is inspired by these works.

**Security Patch Identification.** Existing works make many attempts to identify security patches, i.e., a type of bug-fixing commits in the source code repository. Wang *et al.* [45] leverage machine learning while SPIDER [26] uses symbolic interpretation to classify safe patches and identifies security patches from them. Recently, Tan *et al.* [41] use a ranking method to locate the security patches for a specific vulnerability. By proposing a nearest link search method, Wang *et al.* [46] construct a large-scale patch database. In this paper, a semi-automatic method is proposed to facilitate the collection of security patches for a specific CVE on all affected stable branches rather than a or several branch(es).

**Patch Lifecycle Analysis.** Several studies have been conducted to measure the lifecycle of security patches. In particular, Li *et al.* [25], Shahzad *et al.* [37] and Frei *et al.* [19] perform patch studies on various OSS projects while Farhang *et al.* [18] target the Android system. Zheng *et al.* [49], Jiang *et al.* [24] and Dai *et al.* [15] study the patch propagation from Android upstream code repository (AOSP) to downstream vendors. Those works either focus on patch management on the mainline itself or concern the patch management between the upstream and the downstream, while this paper uniquely investigates the problem of patch management across multiple stable branches inside OSS projects.

## 7 CONCLUSION

In this paper, we conduct a large-scale empirical study of security patch management practice across multiple stable branches in OSS projects, with 806 vulnerabilities and 608 stable branches from 26 OSS projects. We uncover the poor patch management status in OSS projects and obtain many useful findings. In particular, over 80% of CVE-Branch pairs are not patched. About 90% unpatched stable branches are due to out-of-maintenance and the remaining 10% are caused by patch management failures. We also investigate the patch porting process, to shed light on the challenges and the required efforts in patch porting. Based on our study, we call for the OSS community to improve the current security patch management practice, probably by applying automated patch porting tools and establishing a mechanism to monitor patch deployment status.

## ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (U1836210, U1836213, 62172105, 61972099, 62172104, 62102091, 62102093), Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700. Kun Sun is supported by U.S. ARMY grant W56KGU-20-C-0008 and U.S. NAVY grant N00014-20-1-2407. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

# REFERENCES


[1] 2010. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.
[2] 2016. Syzkaller. https://github.com/google/syzkaller.
[3] 2018. syzbot dashboard. https://syzkaller.appspot.com/upstream.
[4] 2021. Bigtree CMS. https://www.bigtreecms.org/.
[5] 2021. Bigtree CMS release cycle. https://www.bigtreecms.org/developers/dev-guide/release-cycle/.
[6] 2021. github. https://github.com/.
[7] 2021. HHVM. https://hhvm.com/.
[8] 2021. HHVM release policy. https://docs.hhvm.com/hhvm/FAQ/faq.
[9] 2021. Jackson databind. https://github.com/FasterXML/jackson-databind.
[10] 2021. OpenEMR. https://www.open-emr.org/.
[11] 2021. phpMyAdmin. https://www.phpmyadmin.net/.
[12] 2021. QEMU. https://www.qemu.org/.
[13] 2021. Semantic Versioning 2.0.0. https://semver.org/.
[14] MITRE Corporation. 2021. CWE: Common Weakness Enumeration. https://cwe.mitre.org/.
[15] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium (USENIX Security)*.
[16] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating Vulnerability Assessment through PoC Migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.
[17] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*.
[18] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. 2019. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. *arXiv preprint arXiv:1905.09352* (2019).
[19] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. 2006. Large-Scale Vulnerability Analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense (LSAD)*.
[20] Google. 2016. OSS-Fuzz. https://github.com/google/oss-fuzz.
[21] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*.
[22] Thong Hoang, Julia Lawall, Richard J. Oentaryo, Yuan Tian, and David Lo. 2019. PatchNet: A Tool for Deep Patch Classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.
[23] Thong Hoang, Julia Lawall, Yuan Tian, Richard J. Oentaryo, and David Lo. 2021. PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel. *IEEE Transactions on Software Engineering (TSE)* (2021).
[24] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[25] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[26] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P)*.
[27] Robert K Merton. 1968. The Matthew effect in science: The reward and communication systems of science are considered. *Science* (1968).
[28] Robert K Merton. 1988. The Matthew effect in science, II: Cumulative advantage and the symbolism of intellectual property. *isis* (1988).
[29] Mockus and Votta. 2000. Identifying reasons for software changes using historic databases (ICSM). In *Proceedings 2000 International Conference on Software Maintenance*.
[30] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.
[31] U.S. National Institute of Standards and Technology. 2021. National Vulnerability Database. https://nvd.nist.gov/home.cfm.
[32] U.S. National Institute of Standards and Technology. 2021. NVD Data Feed. https://nvd.nist.gov/vuln/data-feeds.
[33] U.S. National Institute of Standards and Technology. 2021. NVD Specific CVSS Information. https://nvd.nist.gov/vuln-metrics/cvss.
[34] U.S. National Institute of Standards and Technology. 2021. Official Common Platform Enumeration Dictionary. https://nvd.nist.gov/products/cpe.
[35] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.
[36] Luis R. Rodriguez and Julia Lawall. 2015. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In *2015 11th European Dependable Computing Conference (EDCC)*.
[37] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *34th International Conference on Software Engineering (ICSE)*.
[38] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[39] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches. In *31th USENIX Security Symposium (USENIX Security)*.
[40] Yan Sun, Qing Wang, and Ye Yang. 2017. FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance. *Information and Software Technology* (2017).
[41] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[42] Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.
[43] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*.
[44] Tom Walker. 2021. 20 Most Popular Open Source Software Ever. https://www.tripwiremagazine.com/20-most-popular-open-source-software-ever-2/.
[45] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[46] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[47] WhiteSource. 2021. The State of Open Source Vulnerabilities 2021. https://www.whitesourcesoftware.com/resources/blog/2021-state-of-open-source-security-vulnerabilities-cheat-sheet/.
[48] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: Recovering Links between Bugs and Changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE/ESEC)*.
[49] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *30th USENIX Security Symposium (USENIX Security)*.

# A APPENDIX

**Table 8: Heuristic Rules Used in the Semi-automated Method for Potential Patch Commits Locating.**

| Rule Description | Manual Inspection |
|---|---|
| The commit and the reference commit have the same commit ID. | No |
| The commit shares the same code diff with the reference commit. | No |
| The commit shares the same commit title with the reference commit and has over 90% of the same lines in the code diff with the reference commit. | No |
| The commit mentions the commit ID of the reference commit and has over 90% of the same lines in the code diff with the reference commit. | No |
| The commit shares the same commit title with the reference commit. | Require |
| Both the commit and the reference commit mention the same commit ID. | Require |
| The commit mentions the commit ID of the reference commit in the commit message. | Require |
| The commit has more than 40% of the same lines in the code diff with the reference commit. | Require |
| Both the commit and the reference commit mentions the same Bug-ID. | Require |
| The commit mentions the CVE-ID of the vulnerability which the reference commit fixes. | Require |

**Table 9: The Top 5 Vulnerability Types for Not-fully-patched CVEs due to Patch Management Failures (RQ3).**

| Language | 1st Type | 2nd Type | 3rd Type | 4th Type | 5th Type |
|---|---|---|---|---|---|
| C | CWE-401 | CWE-835 | CWE-125 | CWE-787 | CWE-416 |
| C++ | CWE-125 | CWE-787 | CWE-20 | CWE-522 | CWE-119 |
| Java | CWE-200 | CWE-502 | CWE-79 | CWE-863 | CWE-326 |
| PHP | CWE-79 | CWE-89 | CWE-862 | CWE-918 | CWE-326 |
| Python | CWE-732 | CWE-125 | CWE-862 | CWE-287 | / |

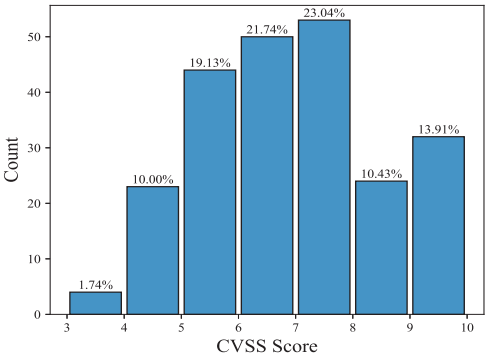


**Figure 5: Distribution of CVSS Severity Scores (RQ3).**

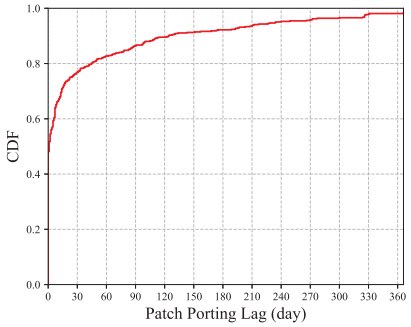


**Figure 6: Cumulative Distribution of Patch Porting Delay (RQ4).**

**Table 10: The OSS and Vulnerability Dataset for the Study (column 7 and column 8 represents the *average number of code commits* and the *average maintenance time* for all collected stable branches in each OSS, respectively).**

| Software | Application Type | #Stable-Branches | #CVEs selected | Language | Stars | # Code Commits | Maintenance Duration (Days) |
|---|---|---|---|---|---|---|---|
| Linux kernel | Operating System Kernel | 84 | 50 | C | 119k | 3,037.82 | 393.21 |
| Wireshark | Network Traffic Analyzer | 14 | 50 | C | 3.7k | 850.57 | 754.64 |
| FFmpeg | Multimedia Content Processor | 29 | 49 | C | 26.4k | 624.31 | 1,065.28 |
| QEMU | Emulator | 33 | 37 | C | 5.3k | 113.85 | 77.94 |
| Xen | Virtual Machine Monitor | 16 | 33 | C | 358 | 471.62 | 981.56 |
| glibc | GNU C Library | 24 | 30 | C | 603 | 88.17 | 691.92 |
| Samba | Windows Interoperability Suite | 20 | 30 | C | 598 | 1,440.35 | 721.10 |
| OpenSSL | TLS/SSL and Crypto Library | 8 | 29 | C | 16.6k | 1,738.25 | 2,839.62 |
| ClamAV | Antivirus Engine | 8 | 27 | C | 1.6k | 171.88 | 578.00 |
| PostgreSQL | Object-relational Database | 25 | 26 | C | 9k | 1,031.36 | 1,479.52 |
| Suricata | Threat Detection Engine | 5 | 22 | C | 2.2k | 227.40 | 419.40 |
| HHVM | Virtual Machine | 107 | 27 | C++ | 17.1k | 19.56 | 71.09 |
| Ceph | Distributed Storage System | 13 | 24 | C++ | 9.7k | 737.69 | 822.31 |
| MongoDB | Document-oriented Database | 20 | 21 | C++ | 20.5k | 785.60 | 644.55 |
| Jenkins | Automation Server | 24 | 30 | Java | 17.9k | 34.75 | 106.38 |
| Jackson Databind | Data-binding Package | 14 | 30 | Java | 2.9k | 206.43 | 901.57 |
| WordPress | Content Management System | 39 | 42 | PHP | 15.5k | 162.08 | 1,082.21 |
| Dolibarr | ERP and CRM | 24 | 30 | PHP | 2.7k | 474.83 | 798.71 |
| OpenEMR | Medical Practice Management | 14 | 30 | PHP | 1.7k | 85.86 | 245.43 |
| GLPI | Asset and IT Management | 14 | 28 | PHP | 2.2k | 439.21 | 620.07 |
| phpMyAdmin | Administration Tool | 6 | 27 | PHP | 5.7k | 849.17 | 496.83 |
| Piwigo | Photo Gallery Software | 17 | 26 | PHP | 1.6k | 145.06 | 662.18 |
| BigTree CMS | Content Management System | 4 | 25 | PHP | 193 | 315.25 | 609.25 |
| Django | Web Framework | 22 | 32 | Python | 60k | 530.55 | 780.09 |
| Pillow | Python Imaging Library | 16 | 29 | Python | 9k | 11.62 | 40.88 |
| Zulip | Group Chat Application | 8 | 22 | Python | 14.3k | 74.62 | 220.00 |