Computers & Security 148 (2025) 104181

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

A large-scale analysis of the effectiveness of publicly reported security patches

Seunghoon Woo, Eunjin Choi, Heejo Lee *

Department of Computer Science and Engineering, Korea University, Seoul, 02841, Republic of Korea

ARTICLE INFO

Dataset link: https://github.com/wooseunghoon/COSE_patchStudy

Keywords:
Security patch
Software vulnerability
Vulnerability propagation
Patch reliability and flexibility
Open-source software security

ABSTRACT

Public vulnerability reports assist developers in mitigating recurring threats caused by software vulnerabilities. However, security patches that lack effectiveness (1) may fail to completely resolve target vulnerabilities after application (*i.e.*, require supplementary patches), or (2) cannot be directly applied to the codebase without modifying the patch code snippets. In this study, we systematically assessed the effectiveness of security patches from the perspective of their reliability and flexibility. We define a security patch as reliable or flexible, respectively, if it can resolve the vulnerability (1) without being complemented by additional patches or (2) without modifying the patch code snippets. Unlike previous studies that relied on manual inspection, we assess the reliability of a security patch by determining the presence of supplementary patches that complement the security patch. To evaluate flexibility, we first locate vulnerable codes in popular open-source software programs and then determine whether the security patch can be applied without any modifications. Our experiments on 8,100 security patches obtained from the National Vulnerability Database confirmed that one in ten of the collected patches lacked effectiveness. We discovered 476 (5.9%) unreliable patches that could still produce security issues after application; for 84.6% of the detected unreliable patches, the fact that a supplementary patch is required is not disclosed through public security reports. Furthermore, 377 (4.6%) security patches were observed to lack flexibility; we confirmed that 49.1% of the detected vulnerable codes required patch modifications owing to syntax diversity. Our findings revealed that the effectiveness of security patches can directly affect software security, suggesting the need to enhance the vulnerability reporting process.

1. Introduction

The reuse of open-source software (OSS) plays a key role in innovative software development. Developers can easily reuse desired functionalities from a reliable OSS rather than re-inventing the wheel, which in turn leads to a faster release of their software products in the ever-competitive markets (Woo et al., 2021b; Zhan et al., 2021).

Meanwhile, the reuse of OSS without proper management can threaten the entire system. One representative problem is the propagation of vulnerabilities, which has occurred frequently in recent code-sharing cultures (Kim et al., 2017; Woo et al., 2022; Kang et al., 2022; Woo et al., 2023). To prevent recurring threats caused by propagated vulnerabilities, relevant information about discovered vulnerabilities (*e.g.*, security patches, severity, and vulnerability types) is maintained through public vulnerability databases in the Common Vulnerabilities and Exposures (CVE) system.

Because developers rely on CVE information to mitigate threats imposed by vulnerabilities, the quality control of public vulnerability reports (*e.g.*, Dong et al., 2019; Woo et al., 2021a; Shi et al., 2022) has become an important issue in software security. We focused on the *effectiveness* of the disclosed security patches from the information provided in public vulnerability reports. We determined that a security patch was effective if the target vulnerability was resolved by applying it without requiring any additional conditions.

A security patch can modify various code parts of the codebase, and when additional measures are required in any of these parts, we define the security patch as inefficient. Specifically, we assessed the effectiveness of the security patches from two perspectives: *reliability* and *flexibility*.

- **Patch reliability.** We define a security patch as *reliable* if the target vulnerability is resolved by applying a disclosed security patch ***without requiring any additional patches.***
- **Patch flexibility.** We define a security patch as *flexible* if a propagated vulnerability is resolved by applying a disclosed security patch ***without modifying any patch code snippets.***

* Corresponding author.
E-mail address: heejo@korea.ac.kr (H. Lee).

https://doi.org/10.1016/j.cose.2024.104181
Received 16 September 2023; Received in revised form 15 August 2024; Accepted 23 October 2024
Available online 29 October 2024

If a security patch lacks reliability, the target vulnerability may not be completely resolved even after the patch is applied (Li and Paxson, 2017; Liu et al., 2020). To make matters worse, unreliable patches may generate other security issues (Section 3.3.6). Consequently, developers may misinterpret software that contains incompletely resolved vulnerabilities as safe to use. Furthermore, most public vulnerability reports disclose only security patches applied to the main branch of a vulnerable OSS. When developers reuse OSS codes from branches other than the main branch, the disclosed patches may not be applied directly because of the syntax diversity of vulnerable codes (Tan et al., 2022).

To our knowledge, no previous studies have examined the reliability and flexibility of security patches on a large scale. Previous studies that attempted to analyze patch reliability were limited in scope because they examined a small number of software programs (*e.g.*, An et al., 2014; Park et al., 2012; Piantadosi et al., 2019), manually identified unreliable patches among a limited number of vulnerabilities (*e.g.*, Li and Paxson, 2017; Le et al., 2019; Liu et al., 2020), or focused on a specific type of vulnerability (*e.g.*, Kim et al., 2010). Although some studies have attempted to examine the presence of security patches from downstream vendors (*e.g.*, Jiang et al., 2020; Zhang et al., 2021) or analyze patch management within multiple branches of a single OSS (*e.g.*, Tan et al., 2022), none have examined the flexibility of security patches from the perspective of addressing vulnerabilities propagated to various OSS projects.

In this study, we perform a large-scale analysis of the effectiveness of security patches. This comprises the following three main steps: (1) security patch collection, (2) unreliable and inflexible patch detection, and (3) result analysis. For the experiments, we gathered 8100 security patches from the National Vulnerability Database (NVD), written in the top seven programming languages (including C, C++, and PHP) that reported the most vulnerabilities (Section 2.2).

Thereafter, we detected unreliable and inflexible patches among the collected security patches. To detect unreliable patches, we focus on the complementarity between code commits. Instead of manually examining each security patch, which requires considerable time and effort, we determined that a security patch is unreliable by checking whether any supplementary patches have been applied to complement the security patch; this allowed us to assess patch reliability on a large scale (Section 3.1). Next, to identify inflexible security patches, we examined whether a security patch could be applied to the propagated vulnerabilities without any modification. Here, we leverage a state-of-the-art technique (Woo et al., 2022), which can precisely discover vulnerable codes propagated in various code syntaxes (Section 4.1).

In our experiments, we observed that one in ten security patches lacked effectiveness. Specifically, we identified 476 (5.9%) unreliable patches (Section 3.2), of which 239 were confirmed as requiring supplementary patches because they could cause further security issues. For 84.6% of the detected unreliable patches, the fact that supplementary patches were required to resolve security issues was not disclosed in the public vulnerability databases. This makes it difficult to address the risks posed by unreliable patches in real-world software ecosystems. We discovered 302 incompletely patched codes in the latest versions of 95 popular OSS projects (Section 3.4).

Furthermore, we identified 377 (4.6%) inflexible security patches that could not be applied directly to address the propagated vulnerabilities (Section 4.2). In fact, 49.1% of the detected vulnerabilities required patch modification because of the diversity in the syntax of the propagated vulnerable codes, which frequently occurs when developers (1) reuse vulnerable code from a branch of an OSS other than the main branch, or (2) modify some code lines in the propagated vulnerable functions (Section 4.3.1).

Our findings provide insights into the effectiveness of the current security patches and suggest directions for improving the current public vulnerability reporting process (Section 5.1). Furthermore, our experimental results can be applied to various fields to enhance the security of real-world software systems, such as vulnerable code detection approaches (Section 5.2).

This study makes the following four main contributions.

- *Large-scale analysis.* For the first time, we present a study on the effectiveness of security patches on a large scale, by devising automated code analysis techniques that consider the complementarity between code commits and leveraging a state-of-the-art vulnerable code clone detection technique. Consequently, we identified 476 unreliable and 377 inflexible patches from 8100 security patches obtained from the NVD.
- *Practical insights.* We not only identified unreliable and inflexible patches but also examined them from various perspectives to gain practical insights. Furthermore, we investigated the potential threats that these patches may pose to popular OSS projects.
- *Proposal for improvement.* Based on the experimental results, we proposed a method for vulnerability reporting and security threat management.
- *Disclosure of datasets and results.* The dataset and results of our study are publicly available on GitHub (https://github.com/wooseunghoon/COSE_patchStudy.)

2. Modeling and dataset

In this section, we introduce our modeling and datasets for assessing the effectiveness of security patches.

2.1. Modeling and motivation

First, we introduce the modeling and the motivation behind the study.

2.1.1. Definition

Security patch. We define the patch that should be applied to address a CVE vulnerability as the *security patch*. In particular, all patches mentioned in the public vulnerability database (*e.g.*, references) of a specific CVE are considered as security patches for that CVE. Even if a security patch modifies multiple sections of the codebase, we consider it comprehensively as a security patch for a specific CVE.

Security patch effectiveness. We consider a security patch to be *effective* if it can be applied directly to a vulnerable code and fully resolve the target vulnerability, without modifying any patch code snippets or applying supplementary patches. A security patch can modify multiple sections of the codebase. If any of these sections require additional measures to fully resolve the vulnerability, we consider the patch to be inefficient.

2.1.2. Modeling

In this paper, we evaluate the reliability and flexibility of security patches. To this end, we intend to model the effectiveness of patches. The reliability and flexibility of a security patch depend on (1) which software it is applied to and (2) how well it can be applied. Therefore, we first modeled the patching targets and applicability of patches.

The applicability of security patches can be classified into the following three categories.

A1. Applicable. A security patch can be applied to vulnerable codes to resolve vulnerabilities.

A2. Applicable but incomplete. A security patch can be applied to vulnerable codes but fails to completely resolve the vulnerability (*e.g.*, a partial fix).

A3. Inapplicable. A security patch cannot be applied directly to a target vulnerable code (*e.g.*, when the syntax of the vulnerable code differs from that disclosed).

The applicability of a patch is closely related to the target software and branches to which it is applied. Therefore, we then classified the patching targets of the security patches.

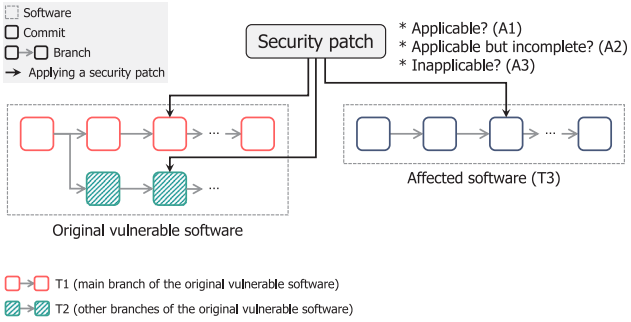


Fig. 1. Depiction of the patch application model. We model the patching targets (*i.e.*, T1, T2, and T3) and applicability (*i.e.*, A1, A2, and A3) to evaluate the reliability and flexibility of security patches.

T1. Main branch of the original vulnerable software. Security patches are first applied when a security issue is detected. Typically, security patches applied to this branch are disclosed in public vulnerability databases (Tan et al., 2022).

T2. Other branches of the original vulnerable software. This refers to branches other than the main branch of the software where the vulnerability was first discovered and patched.

T3. Affected software. This refers to software (other than the original vulnerable software) that contains vulnerabilities.

Fig. 1 depicts the patch application model considered in this study. Here, **A2** may occur in **T1**, **T2**, and **T3**, whereas **A3** may occur in **T2** and **T3**.

2.1.3. Analysis targets

Among the pairs of applicability and patching targets, we focused on two combinations that have not been extensively investigated on a large scale: (**A2**, **T1**) and (**A3**, **T3**) cases. Developers anticipate vulnerabilities to be resolved by applying disclosed security patches. However, vulnerabilities can only be resolved by applying patches other than those disclosed (**A2**, **T1**) (Li and Paxson, 2017; Liu et al., 2020). Furthermore, when a vulnerable code is propagated to other software, a disclosed security patch may not be applied without modification (**A3**, **T3**) (Woo et al., 2022; Tan et al., 2022). In these scenarios, because the target vulnerability cannot be resolved by applying the disclosed security patch, the patch is determined to be ineffective according to our definition (Section 2.1.1). Therefore, we examined the effectiveness of security patches from the perspectives of their *reliability* (**A2**, **T1**) and *flexibility* (**A3**, **T3**).

- **Unreliable patch (A2, T1).** A security patch has been disclosed but to completely resolve the vulnerability, a supplementary patch needs to be applied.
- **Inflexible patch (A3, T3).** A security patch has been disclosed but the patch code needs to be modified to apply to the responsible codebase.

The remaining cases are not discussed in depth for the following reasons.

- **A1** is ignored as it does not have any security issues.
- The cases (**A2**, **T2**) and (**A2**, **T3**) are also excluded because if a security patch turns out to be unreliable (**A2**) on **T1**, it is also unreliable in other patching targets; therefore, in the case of **A2**, we decided that it was sufficient to investigate applicability in **T1**.

- Since security patches are generated in **T1**, we excluded (**A3**, **T1**) under the assumption that there are no cases where a patch is inapplicable in **T1**.

```
1 // CVE ID: CVE-2018-1000006
2 // Electron/atom/app/command_line_args.cc
3 bool IsBlacklistedArg(...) {
4 ...
5 + if (prefix_length > 0) {
6 +   a += prefix_length;
7 +   std::string switch_name(a, strcspn(a, "="));
8 ...
```

Fig. 2. Example of an unreliable patch.

```
1 // CVE ID: CVE-2018-1000118
2 // Electron/atom/app/command_line_args.cc
3 bool IsBlacklistedArg(...) {
4 ...
5 if (prefix_length > 0) {
6   a += prefix_length;
7 -   std::string switch_name(a, strcspn(a, "="));
8 +   std::string switch_name =
9 +     base::ToLowerASCII(base::StringPiece(a,
        strcspn(a, "=")));
```

Fig. 3. Example of a supplementary patch.

- Although a patch may not be applied in **T2** due to differences in code syntax between branches (**A3**), this has been covered in previous studies (*e.g.*, Tan et al., 2022) and we can leverage the results.

Therefore, we considered it urgent to conduct a study on (**A2**, **T1**) and (**A3**, **T3**), which can demonstrate the reliability and flexibility of security patches.

2.1.4. Motivating example

We present an example in which a vulnerability is not fully resolved because of an unreliable patch. In 2018, a severe vulnerability (CVE-2018-1000006, CVSS 8.8) that enabled arbitrary command execution through crafted URLs was discovered in `Electron`, a widely used tool for creating cross-platform applications. The `Electron` team acknowledged the risk of vulnerability and applied a security patch; they attempted to block the crafted URLs using a blacklist method (Fig. 2). However, it was later discovered that blacklists were case-insensitive, indicating that attackers could still exploit this vulnerability. To completely resolve this vulnerability, the `Electron` team applied additional patches to their codebase (Fig. 3).

This example illustrates that unreliable patches pose a threat to security. In particular, the `Electron` team made an additional patch publicly available through vulnerability databases and assigned it a new CVE ID (CVE-2018-1000118). However, in several cases of unreliable patches, the facts that (1) the previous security patch is unreliable and (2) additional patches should be applied are not disclosed in public vulnerability databases (Section 3.3.4). Therefore, developers who reuse vulnerable OSSs may misinterpret that vulnerability can be resolved by applying an unreliable patch, and until the supplementary patch is applied, their software remains vulnerable to attacks (real-world cases are presented in Section 3.4).

Hence, we need to assess the effectiveness of security patches to reduce the attack surface of software containing vulnerable codes. In addition, we examined the characteristics of ineffective patches to gain a better understanding and devise more efficient methods for addressing them.

2.2. Datasets

Here, we introduce the dataset construction approaches used in the experiment.

Table 1
Security patch dataset overview.

C/C++	PHP	JavaScript	Python	Java	Ruby	Total
4420	1862	681	529	341	267	8100

2.2.1. Security patch dataset

To collect security patches, commit-like URLs were retrieved from references in public vulnerability databases, which are among the most precise patch collection methods (Li and Paxson, 2017; Tan et al., 2021; Hong et al., 2022). Specifically, we examined the CVEs in the NVD (using JSON feeds) and verified whether GitHub commit URLs were included in the references. We targeted GitHub because it is one of the most popular hosting services and provides various commands and APIs for examining vast amounts of source code. Thereafter, we collected the security patches by crawling the identified commits. Consequently, 9963 patch commits were collected as of March 2023. Among the collected security patches, C/C++ patches occupied the highest proportion (54.6%), followed by PHP (23%), and JavaScript (8.4%).

However, this dataset contained several security patches that were unrelated to programming languages (*e.g.*, image files). Thus, we focused only on a few popular programming languages to refine the dataset and obtain unbiased experimental results. Specifically, we considered only the top seven languages (C, C++, PHP, JavaScript, Python, Java, and Ruby) with more than 200 security patches. The seven selected languages represented more than 80% of the security patches collected.

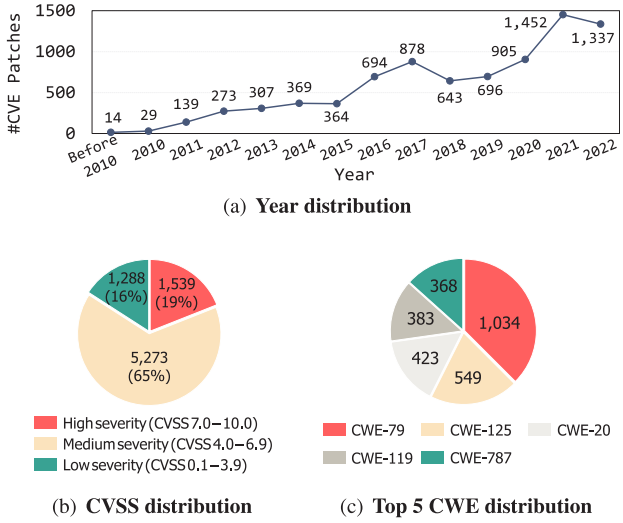
Finally, we collected 8100 security patches from the seven previously mentioned languages at the code level and used them to evaluate the effectiveness of security patches. Table 1 summarizes the number of collected security patches, and Fig. 4 shows the year, Common Vulnerability Scoring System (CVSS), and Common Weakness Enumeration (CWE) distributions of the collected security patches.

When investigating the reliability of patches, we utilize all 8100 vulnerability patches, and when examining patch flexibility, we consider only patches written in C/C++ (54%). Inflexible patches lead to issues when modifications in propagated code are assumed. (1) Such modifications predominantly occur in C/C++, where code-level reuse is more common than package reuse (Woo et al., 2021b; Na et al., 2024), and (2) since techniques for detecting modified and propagated vulnerabilities mostly target C/C++ (Xiao et al., 2020; Woo et al., 2022), we focus on analyzing patch flexibility in patches and software written in C/C++.

Note that the size of our dataset has expanded by approximately twice compared to existing relevant approaches (*e.g.*, Li and Paxson, 2017 and Woo et al., 2021a collected approximately 3000 and 4000 security patches, respectively). In particular, Li and Paxson (2017) focused on analyzing the efficiency of patches based on their metadata. It is significant that we included even more C/C++ patches in our dataset compared to the entire patch dataset of the previous study, while also analyzing the propagation of vulnerabilities and the flexibility of C/C++ security patches. Therefore, we determined that our dataset is sufficient to demonstrate the overall trend in the effectiveness of security patches.

2.2.2. Software dataset

Two software datasets were constructed for this study. The first dataset comprises software repositories that have reported the security patches that we collected, and we used this dataset to assess patch reliability. We can easily gather such software repositories by parsing the commit URLs of the collected security patches, because these URLs contain the name of the repository (Hong et al., 2022). Consequently, we obtained 2538 repositories (using the `git clone` command) that reported 8100 security patches. The repository that reported the highest number of security patches was the `Linux kernel`; other repositories such as `TensorFlow` and `Spring` were also collected.



(a) **Year distribution**

(b) **CVSS distribution**

(c) **Top 5 CWE distribution**

Fig. 4. Year, CVSS (version 2) and CWE distributions for the collected security patches.

The second dataset comprised popular GitHub repositories and was used to evaluate the patch flexibility. An inflexible patch can be detected when a propagated vulnerability has a different syntax than the disclosed vulnerable code. Therefore, we should first construct a dataset necessary for detecting propagated vulnerabilities. Specifically, we focused on C/C++ languages because they are more prevalent in source code reuse than library-level OSS reuse through package managers (Woo et al., 2021b; Xiao et al., 2020). Therefore, we collected the top 2000 C/C++ repositories on GitHub based on the number of stargazers, including OSS projects such as the `Linux kernel`, `Redis`, and `Git`.

3. Patch reliability analysis

In this section, we examine the reliability of collected security patches.

Analysis scope. In this study, we focused on identifying unreliable security patches that are indisputable but unknown to the public. This refers to instances in which developers have acknowledged that the disclosed security patch is unreliable and have applied for additional patches, but not disclosed through public vulnerability databases. If this information is not publicly available, developers may mistakenly believe that applying an unreliable patch resolves the vulnerability completely, potentially giving rise to additional security threats (see Section 2.1.4). Therefore, we focus on clearly identifying such unreliable security patches. Notably, if a security patch modifies multiple areas of the codebase, and if any code modifications lack reliability, we consider the patch to be unreliable.

3.1. Unreliable patch detection methodology

Consider the commit history (C) of the file containing a vulnerable code as a sequence c_0 (initial commit) to c_n (latest commit), in which c_i denotes an individual commit. Let c_v be a disclosed security patch.

$$C = [c_0, c_1, \ldots, c_v, \ldots, c_{n-1}, c_n]$$

To detect unreliable patches, we examine all commits applied after the disclosed security patch commit was applied: if a code commit c_s ($v < s$) that complements c_v exists, we consider c_v as an *unreliable* patch and c_s as the *supplementary* patch for c_v.

∘ ***Unreliable security patch.*** We determined that a security patch is unreliable if it is complemented by supplementary patches.

Subsequently, the following question arises: *how can supplementary patches be detected*? One approach is to manually inspect all commits applied after the security patch. However, this task is impractical for a large number of security patches, because it requires significant expertise and time. Instead, we decided to use a hint by investigating pairs of known unreliable patches and supplementary patches. We distinguish between known and hidden unreliable patches by determining whether information regarding their unreliability has been disclosed in public vulnerability databases.

3.1.1. Preliminary experiment

When an unreliable patch is detected, developers typically generate a supplementary patch and sometimes assign a new `CVE ID` to it. Here, the new CVE often specifies an incompletely resolved `CVE ID` in the description (*e.g.*, "caused by an incomplete fix for `CVE ID`").

Accordingly, we scanned all CVEs that included the keyword "incomplete fix for CVE", and obtained 488 pairs of known unreliable and supplementary patches. After manual inspection, we targeted 236 pairs that released code-level security patches via external references (*e.g.*, Git and Bugzilla). Thereafter, we analyzed the code change patterns between known unreliable and supplementary patch pairs, and examined the code lines that were added or deleted in each patch to investigate whether any patterns existed that could be used to detect hidden unreliable patches.

Interestingly, we confirmed that 72% (169) of the known supplementary patches corrected code lines modified in the unreliable patch *again*, in the same function. In the remaining cases, supplementary patches modified different code areas with the same function or modified code lines with different functions. Because most of the remaining cases did not exhibit a specific code change pattern, it was difficult to consider them for automatically detecting hidden unreliable patches (this is discussed in Section 6). By contrast, we determined that hidden unreliable patches could be detected by leveraging the code change patterns observed in the majority of disclosed supplementary patches.

3.1.2. Unreliable patch detection

To detect unreliable security patches, we verified whether a commit changed the code lines modified in the disclosed patches. However, if we simply consider a commit that re-modifies code lines deleted (added) from the disclosed patch (c_v) as a supplementary patch (c_s), many false alarms can be produced when c_v modifies short and general code lines. Moreover, it is error-prone to consider the line numbers of modified codes; the line numbers may vary as code changes proceed in commits between c_v and c_s.

To reduce false alarms, we thus tracked only commits that modified the same code locations. However, using a function name to detect the same function may produce false alarms when the function name changes. In addition, using a Git command (*e.g.*, `git log` and `git blame`) that can trace the commit history of a function may fail to provide a precise result, for instance, when the function is moved to another file. Furthermore, even in a function, code lines with the same syntax can exist in multiple locations.

To overcome the aforementioned issues, we consider both of the following two factors to determine whether two commits modify the same function: (1) the name of the function and (2) the syntax similarity between the modified functions in each commit. Instead of relying solely on easily changeable metadata, we aimed to determine the same function by considering the syntax similarity. Furthermore, we considered the nearby code lines of the code lines modified in each commit to clarify the code locations.

Let us consider the notations in Table 2. If c_s that satisfies the following three conditions is identified, we decide that c_s is a supplementary patch and c_v is an unreliable patch.

Table 2
Defined notations.

Notation	Description
`H(i,j)`	A hunk for function f_i modified in c_j.
`ADD(i,j)`, `DEL(i,j)`	Code lines added or deleted in `H(i,j)`.
`NR(l,n)`	Nearby n code lines of the code line l.
`NM`(f_i)	The function name of f_i.
`SIM`(f_i, f_j)	The syntax similarity between f_i and f_j.

• ***Condition 1: Common function.*** There should exist a function (f_c) that is commonly modified in both c_s and c_v.

$$\exists\, f_v \in c_v, \exists\, f_s \in c_s \mid \left(\texttt{NM}(f_v) = \texttt{NM}(f_s)\right) \vee \left(\texttt{SIM}(f_v, f_s) \geq \theta\right)$$

• ***Condition 2: Same code location.*** The hunk $\texttt{H}(c, s)$, excluding $\texttt{ADD}(c, s)$, should contain at least one code line adjacent to the code lines added or deleted in $\texttt{H}(c, v)$.

$$\exists l \in \left(\texttt{H}(c,s) \backslash \texttt{ADD}(c,s)\right) \mid l \in \texttt{NR}(l_v, n), \\ where\ l_v \in \left(\texttt{ADD}(c,v) \cup \texttt{DEL}(c,v)\right)$$

• ***Condition 3: Same code modification.*** A minimum deletion (*resp.* addition) of one code line from $\texttt{H}(c, v)$ should be included in the set of code lines added (*resp. deleted*) from $\texttt{H}(c, s)$.

$$\left(\exists\, l \in \texttt{ADD}(c,v) \mid l \in \texttt{DEL}(c,s)\right) \vee \\ \left(\exists\, l \in \texttt{DEL}(c,v) \mid l \in \texttt{ADD}(c,s)\right)$$

Fig. 5 depicts the workflow for unreliable patch detection. Under **Conditions 2** and **3**, code lines containing only whitespaces and curly braces were ignored to reduce false alarms. When extracting functions modified in commits, we utilized the universal Ctags (2022) function parser, which has advantages in as concerns speed and language extensibility and can parse functions from various programming languages within a short time. In addition, we used the Jaccard index (Jaccard, 1912; Murphy, 1996) to measure the function similarity (considering a function as a set of code lines). We conducted the experiment by changing θ (used under **Condition 1**) from 0 to 1 in increments of 0.1. We introduced the analysis results of unreliable patches based on the correct answers obtained by manual validation; experiments related to the parameter are presented in Section 3.2.3. We used n as the default value of three provided by the GitHub patch.

3.2. Unreliable patch detection results

3.2.1. Unreliable patch distribution

In our setup, we discovered that 476 (5.9%) of the 8100 collected security patches lacked reliability, excluding the known unreliable and supplementary patch pairs examined in Section 3.2. An unreliable patch can be complemented multiple times; thus, 557 supplementary patches were detected. In addition, we observed that 128 supplementary patches completely reverted the code lines modified in the unreliable security patch (*i.e.*, regression cases). Notably, our approach has successfully detected a significantly larger number of unreliable patches compared to the existing approaches, which manually identified less than 50 unreliable patches (*e.g.*, Li and Paxson, 2017 discovered 43 unreliable patches from 4000 security patches).

Specifically, more than 90% of the unreliable patches were found in the C/C++ repositories, and the repository containing the highest number of unreliable patches was the `Linux kernel` (152), followed by `rdesktop` (114) and `ImageMagick` (66). Based on our manual analysis, most unreliable patches frequently occur in memory-related

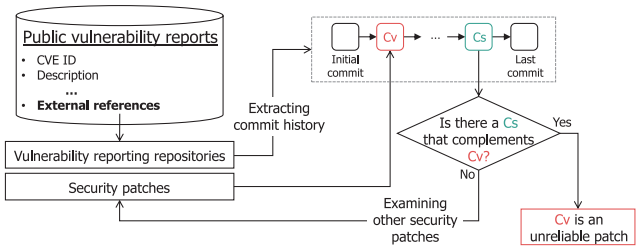
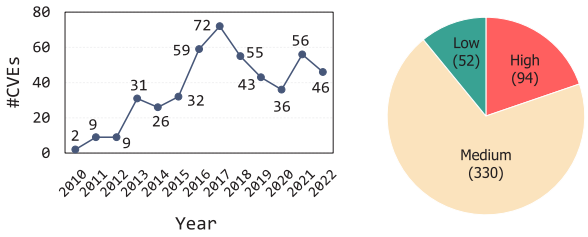


Fig. 5. High-level overview of unreliable patch detection.



(a) **Year distribution** (b) **CVSS distribution**

Fig. 6. Distributions of the detected unreliable patches by year (Fig. 6(a)) and by CVSS (Fig. 6(b)).

vulnerabilities (see Section 3.3.3). Since most memory-related vulnerabilities are reported in C/C++, a considerable number of unreliable patches were consequently found in C/C++. Fig. 6 shows the distributions of discovered unreliable patches by year and CVSS. In many cases, the severity was medium, but nearly 20% of unreliable patches had a high severity, indicating the need for appropriate measures to address them.

Interestingly, the CVSS distribution was very similar between the overall security patches (see Fig. 4(b)) and the unreliable patches. The difference was that the proportion of low-severity vulnerabilities was slightly lower in unreliable patches, while the proportions of high and medium-severity vulnerabilities were slightly higher. Although the difference is not significant, the fact that the proportion of higher-risk vulnerabilities is greater in unreliable patches compared to the distribution of general security patches highlights the importance of addressing unreliable patches. Additionally, there was no significant difference in the distribution by year (see Fig. 4(a)). However, since vulnerabilities reported more recently have relatively fewer subsequent commits, the proportion of unreliable patches detected by our method was somewhat lower compared to the overall distribution of security patches.

Finding 1. We observed that 476 (5.9%) disclosed security patches lacked reliability. To address the responsible vulnerabilities completely, the disclosed security patch should be complemented by supplementary patches.

3.2.2. Supplementation to resolve security issues

Not all detected supplementary patches are required for security purposes. For instance, a supplementary patch may be applied to increase the efficiency of the previous patch (*e.g.*, code refactoring).

Hence, we identified supplementary patches applied to address security issues through the following criteria: whether the commit description (1) specifies the CVE ID or seven-character prefix of the commit ID of the unreliable patch (Li and Paxson, 2017), or (2) contains the keywords "fix", "vulnerab", "CVE", "bug", "incomplete", or "incorrect". We can use more keywords (*e.g.*, "overflow") as used in the existing studies (Hong et al., 2021; Islam and Zibran, 2021), but we intended to strictly deduce only commits concerning security issues by considering the six keywords.

Consequently, we observed that 354 (63.6%) supplementary patches (239 unreliable patches) were applied for security purposes; 72 supplementary patches specified either the CVE ID or the commit ID of unreliable patches, and 348 supplementary patches contained one or more of the aforementioned keywords. This case is critical because the application of disclosed security patches may produce further security issues, such as the generation of new vulnerabilities (Section 3.3.6). The remaining supplementary patches were primarily applied for (1) code refactoring, (2) fixing compilation errors, and (3) code cleanup because vulnerable codes were no longer used. Although these may not produce security threats, they may cause functional issues in the entire software; therefore, supplementary patches should be applied.

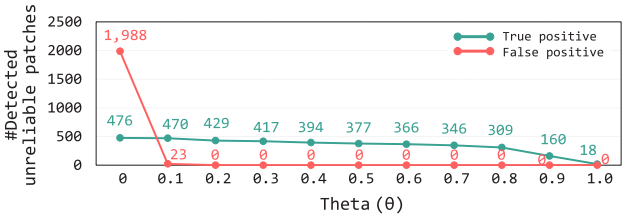


Fig. 7. Results of measuring efficiency of θ.

Finding 2. We confirmed that 354 (63.6%) supplementary patches (239 unreliable patches) were intended to resolve the security issues. If such supplementary patches are not applied, then the security of the entire software may be compromised (Section 3.4).

3.2.3. Parameter sensitivity.

We use θ to determine whether two commits are targeting the same function (Section 3.1). To measure threshold sensitivity, we evaluated each unreliable patch detection result while increasing θ by 0.1 from 0 to 1.0. We measured the efficiency of θ by manually analyzing the detection results: a detected supplementary patch was determined to be correct if it changed again the same code locations modified in the unreliable patch. Fig. 7 presents the measurement results. First, as we argued in Section 3.1, many false alarms were detected when θ was 0 (80.7% false positive rate). In contrast, when θ was greater than or equal to 0.1, false positives significantly dropped. Instead, as θ increased, the number of detected incorrect patches slightly decreased. Because we manually examined all the detection results in our experiment, we could cover 476 unreliable security patches. Nonetheless, we confirmed that the optimal values for detecting unreliable patches in an automated manner in our experimental setup are θ of 0.1 or 0.2.

3.3. Characteristics of unreliable patches

We then provide answers to several questions related to the characteristics of the detected unreliable patches.

3.3.1. Who mainly complements unreliable patches?

To answer this question, we examined the authors of the unreliable and corresponding supplementary patches. When analyzing the detected 557 unreliable and supplementary patch pairs (Section 3.2.1), we observed that 257 (46.1%) supplementary patches were applied by the same author of the unreliable patch, and for the remaining patches, we confirmed that a different developer discovered issues in the unreliable and applied supplementary patches. This indicates that a considerable number of developers evaluated the reliability of security patches to a certain extent after applying them. In addition, there was a tendency for developers who did not apply for the security patch to participate in testing and evaluating it.

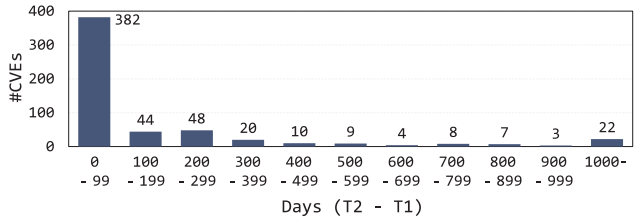


Fig. 8. Representation of elapsed time (days) between the application of unreliable and supplementary patches.

3.3.2. How fast are unreliable patches complemented?

To investigate this, we examined the time elapsed between the unreliable and supplementary patch commit dates. Fig. 8 shows the measurement results. We confirmed that many supplementary patches (68.6%) were applied within 100 days after the unreliable patches were applied. On average, supplementary patches were applied 168 days (a median of 64 days) after unreliable patches were applied. Although some supplementary patches were applied for more than 1000 days after the application of unreliable patches, OSS developers managed the effectiveness of the security patches to some extent after the initial security patch was applied.

Finding 3. More than half of the supplementary patches were applied within 100 days of unreliable patch application (an average of 168 days and a median of 64 days).

3.3.3. In what types of vulnerabilities do unreliable patches occur frequently?

To answer this question, we examined the CWEs of the identified unreliable patches. The five vulnerability types listed in Table 3 frequently appeared in the unreliable patches. All five CWEs in Table 3 belong to the most dangerous vulnerability type in 2023 (Common Weakness Enumeration, 2023). In addition, these five CWEs matched five of the six most-collected CWEs in the security patch dataset (Fig. 4(c)). Interestingly, only five unreliable patches (out of 1034 security patches) were found in CWE-79 (Cross-site Scripting), which was the most collected vulnerability type in the dataset.

In fact, memory-related vulnerabilities are not easily resolved at once and generally require multiple security patches (Hong et al., 2020). Especially in the `Linux kernel`, which can be considered as a representative OSS community, memory-related vulnerabilities are typically patched through multiple security patches (Lee et al., 2018). Additionally, while Cross-site Scripting vulnerabilities can become more complex to resolve in intricate codebases, the solutions, such as escaping, are generally well-known. Consequently, we determined that it is rare for ineffective patches to be applied. For this reason, we concluded that many unreliable patches had been identified for types of vulnerabilities caused by memory management issues (*e.g.*, Out-of-bounds Read and Write; see Table 3), whereas unreliable patches are rarely found for Cross-site Scripting vulnerabilities.

3.3.4. Is it publicly disclosed that the security patch is unreliable and thus a supplementary patch is required?

We analyzed the disclosure statuses of the unreliable and supplementary patches discovered in the experiment, excluding known unreliable and supplementary patch pairs (Section 3.1). Previously, we used the keyword "incomplete fix for CVE" to scan known incomplete patches. Here, we used the `CVE ID`s of the detected incomplete patches and the commit IDs of the supplementary patches. Specifically, we scanned CVEs that (1) included the `CVE ID` of unreliable patches in their descriptions and (2) contained the commit ID of the supplementary patches in their references (*e.g.*, GitHub commit URLs).

Table 3
Frequently appeared vulnerability types in unreliable patches.

Vulnerability types	#Unreliable patches
• Out-of-bounds Read (CWE-125)	136
• Out-of-bounds Write (CWE-787)	84
• Improper Input Validation (CWE-20)	64
• Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119)	50
• NULL Pointer Dereference (CWE-476)	30

Our experiment confirmed that only 86 (15.4%) of the detected 557 unreliable and supplementary patch pairs (Section 3.2) satisfied at least one condition. In particular, 35 CVEs satisfied the CVE ID condition, and 54 CVEs contained the commit IDs of unreliable patches; only three CVEs satisfying these two conditions were discovered.

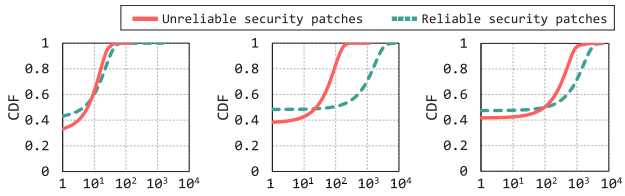
Notably, among the remaining 471 unpublished unreliable and supplementary patch pairs, 65.65% (309) were cases where supplementary patches were needed for security purposes. In other words, out of the total supplementary patches that should be applied for security purposes, only 12.71% were explicitly provided through public vulnerability databases. This result implies that, although supplementary patches to complement unreliable security patches already exist in the commit history, in most cases, it was not publicly disclosed that the security patch was unreliable and that a supplementary patch should be applied. The fact that, even in cases where supplementary patches are needed for security purposes, most of the information about these patches is not publicly available highlights a challenge we need to address for a safer software ecosystem (see Section 5).

Finding 4. Among the detected supplementary patches, only 86 (15.4%) were disclosed in public vulnerability databases. In particular, out of the 471 undisclosed unreliable and supplementary patch pairs, 65.61% (309) were cases where supplementary patches were applied for security purposes. Such inactive disclosures may become an obstacle to resolving threats caused by unreliable patches.

3.3.5. What about the amount of code change in unreliable patches?

We answer this question by examining the number of (1) modified functions, (2) deleted code lines, and (3) added code lines in both reliable (*i.e.*, security patches that are not discovered as unreliable) and unreliable security patches. We leveraged the Ctags (2022) function parser to determine the number of modified functions in the patch. Fig. 9 shows the experimental results.

First, there was no significant difference in the number of functions modified by reliable and unreliable patches (Fig. 9(a)). The average number of modified functions was 4.6 and 6.1 in reliable and unreliable patches, respectively (a median of two in both cases). However, we observed a considerable difference in the number of code lines deleted in each patch (Fig. 9(b)). In particular, unreliable patches often deleted far fewer code lines (average of 24 and median of three code lines) than reliable patches (average of 72 and median of three code lines). Moreover, as shown in Fig. 9(c), we observed that the reliable patches added more code lines (average of 103 and median of 13 code lines) than the unreliable patches (average of 97 and median of 11 code lines). In brief, although the number of modified code lines may not directly affect the patch reliability, we observed that unreliable patches generally modify fewer code lines than reliable patches.



(a) CDFs of the #functions modified in security patches **(b) CDFs of the #code lines deleted in security patches** **(c) CDFs of the #code lines added in security patches**

Fig. 9. CDFs related to reliable and unreliable patches.

```
1 // CVE ID: CVE-2021-32762
2 // Redis/deps/hiredis/hiredis.c
3 static void *createArrayObject (...) {
4 ...
5   r = createReplyObject(task->type);
6 ...
7   if (elements > 0) {
8 +   if (SIZE_MAX / sizeof(redisReply*) < elements)
        return NULL; /* Don't overflow */
```

Fig. 10. Disclosed security patch for CVE-2021-32762.

```
1  // Commit ID: 922ef86a3b1c15292e1f35338a0ac137a08a11b4
2  // Redis/deps/hiredis/hiredis.c
3  static void *createArrayObject (...) {
4  ...
5    if (elements > 0) {
6  -   if (SIZE_MAX / sizeof(redisReply*) < elements)
         return NULL; /* Don't overflow */
7
8  // Redis/deps/hiredis/alloc.c
9  void *hi_calloc(size_t nmemb, size_t size) {
10 + /* Overflow check as the user can specify any arbitrary
       allocator */
11 + if (SIZE_MAX / size < nmemb)
12 +   return NULL;
```

Fig. 11. Supplementary patch for CVE-2021-32762.

```
1 // CVE ID: CVE-2022-26490
2 // Linux/drivers/nfc/st21nfca/se.c
3 int st21nfca_connectivity_event_received(...) {
4 ...
5   transaction->aid_len = skb->data[1];
6 +
7 + /* Checking if the length of the AID is valid */
8 + if (transaction->aid_len > sizeof(transaction->aid))
9 +   return -EINVAL;
```

Fig. 12. Disclosed security patch for CVE-2022-26490.

3.3.6. Case studies

In our detection results, most cases required supplementary patches because the existing security patches did not fully address the intended vulnerabilities (*i.e.*, partial fix). However, in some cases, unreliable patches inadvertently produce new vulnerabilities. For better understanding, we present representative examples for each case.

Patch regression case. In 2021, the `Redis`[1] team discovered an integer overflow vulnerability (CVE-2021-32762, CVSS 8.8) in `Hiredis`, which was contained in the `Redis` codebase. The patches were disclosed using a public vulnerability database after applying the initial security patch (Fig. 10). However, they confirmed that the patch code unintentionally produced a memory leak vulnerability; the variable `r` in line #5 in Fig. 10 was allocated by the `hi_calloc` function, but the `createArrayObject` function can be terminated without freeing `r`. Hence, the `Redis` team reverted the security patch and applied a new patch (Fig. 11) for resolving both integer overflow and memory leak vulnerabilities. The latter patch (Fig. 11) is a case in which a CVE ID was not assigned but was detected in our experiments. We reported this to the `Redis` team and asked them to include a supplementary patch for CVE-2021-32762 (currently under discussion). This example is a representative case where a security patch unintentionally produces new vulnerabilities, making the application of supplementary patches essential. Developers need to review whether a security patch produces new issues once applied. Additionally, for vulnerabilities in reused third-party libraries, it is important to verify the reliability of the security patch using methods such as those we propose before applying it to the reused codebase.

Incomplete patch case. In 2022, a vulnerable code that could cause a buffer overflow vulnerability (CVE-2022-26490, CVSS 7.8) in the `Linux kernel` was discovered. To resolve this vulnerability, the `Linux kernel` team added logic to verify whether the length of the variable `AID` was valid (Fig. 12). However, they later confirmed that a memory leak vulnerability had occurred and that the memory allocated in the code was not properly freed. Therefore, a supplementary patch was applied (Fig. 13) to resolve the memory leak vulnerability. This supplementary patch is not managed as a CVE and exists only as a commit in the `Linux kernel` repository. We requested the `Linux kernel` team to include the supplementary patch as a reference for CVE-2022-26490, and are currently reconciling it. This is the most common case found among unreliable patches, where the initial security patch only partially addressed the issue, necessitating a supplementary patch. Similarly, in this case, applying the supplementary patch is essential to fully resolve the security issue.

[1] https://github.com/redis/redis.

3.4. Impact of unreliable patches

To understand the effect of unreliable patches on software security, we examined the existence of incompletely patched codes in popular GitHub projects.

3.4.1. Methodology

We first extracted incompletely patched functions using unreliable patches and then detected incompletely patched function clones from the latest versions (as of November 2022) of 2000 popular C/C++ OSS projects (Section 2.2.2). We targeted only C/C++ languages because most of the detected unreliable patches were written in C/C++, and code copying and pasting are prevalent in C/C++ languages (Kim et al., 2017; Xiao et al., 2020; Woo et al., 2021a). The detailed process is as follows.

S1. Extracting incompletely patched functions. In this study, we leveraged the methods used in existing vulnerable code detection approaches (Kim et al., 2017; Xiao et al., 2020; Woo et al., 2022). Because we collected security patches in the form of GitHub commits (Section 2.2.1), we used (1) the Git index of the patched source files and (2) the code line numbers to which unreliable patches were applied. Hence, we first accessed the index of the patched source file from the repository reporting the security patch (*e.g.*, using the `git show` command), and thereafter extracted functions (f_v) that contain patched code lines of the unreliable security patches using a function parser (*e.g.*, Ctags, 2022).

S2. Discovering propagated incompletely patched functions. To detect code clones of f_v, we leveraged VUDDY (Kim et al., 2017),

```
// Commit ID: 996419e0594abb311fb958553809f24f38e7abbe
// Linux/drivers/nfc/st21nfca/se.c
int st21nfca_connectivity_event_received(...) {
...
  transaction->aid_len = skb->data[1];

  /* Checking if the length of the AID is valid */
+ if (transaction->aid_len > sizeof(transaction->aid))
+ if (transaction->aid_len > sizeof(transaction->aid)) {
+   devm_kfree(dev, transaction);
    return -EINVAL;
```

Fig. 13. Supplementary patch for CVE-2022-26490.

which has the potential for scalable detection of vulnerable code clones with slight code modifications. Because VUDDY is sensitive to code changes, we determined that it is suitable for precisely detecting code clones of incompletely patched functions in which other patches (*e.g.*, supplementary patches) are not applied. VUDDY calculates the MD5 hash value of each f_v after applying normalization (removing whitespaces and comments) and abstraction (replacing every occurrence of parameters, variable names and types, and function calls to specific symbols) to f_v. VUDDY then calculates the hash values of all functions in the target program. Finally, VUDDY detects code clones of f_v by discovering a function that shows the same hash value as that of f_v, in the target program.

Note that VUDDY is not applicable to detect vulnerabilities where the code has been modified outside its abstraction targets. However, this characteristic is rather effective in our experiment for detecting the propagated vulnerabilities we intend to find (*i.e.*, incompletely patched function clones). If we use previous approaches that are capable of discovering vulnerabilities propagated with syntax modifications (*e.g.*, (Xiao et al., 2020; Woo et al., 2022)), codes with supplementary patches applied can be included in the detection results. Because our goal is to detect vulnerabilities with only unreliable patches applied, a strict propagated vulnerability detection technique is necessary. Therefore, we used VUDDY in this experiment.

3.4.2. Result analysis

From our experiment, we confirmed that 95 (4.75%) OSS projects contained at least one incompletely patched function; 302 incompletely patched functions were discovered. Upon manually verifying the detection results, we observed that VUDDY precisely identified function clones to which the unreliable patches were applied, but the supplementary patches were not applied.

Several OSS projects contain incompletely patched functions that do not pose a security threat (*e.g.*, code refactoring issues). However, in some popular OSS projects, incompletely patched functions have been discovered, which can compromise the security of the entire system. For instance, we detected that the `Greenplum database` (GPDB) contains an incompletely resolved integer overflow vulnerability, `FreeBSD` includes an incompletely addressed buffer over-read vulnerability, and `OpeenToonz` contains an incompletely patched denial-of-service vulnerability, all of which have the potential to be exploited by attackers. In all three cases, the supplementary patches were not disclosed in public vulnerability databases. We responsibly reported the vulnerabilities to the GPDB, `FreeBSD`, and `OpeenToonz` teams and responded that they would resolve the vulnerabilities through subsequent OSS updates. We provide a detailed explanation of how we integrate our patch effectiveness analysis mechanism with existing vulnerability detection techniques to address such threats in Section 5.2. Our experimental results confirm that an incompletely patched code can compromise the security of an entire system.

```
// CVE ID: CVE-2017-14107
// Commit ID: 9b46957ec98d85a572e9ef98301247f39338a3b5
// Libzip/lib/zip_open.c
static zip_cdir_t * _zip_read_eocd64(...)
{
    ...
    zip_error_set(error, ZIP_ER_SEEK, EFBIG);
    return NULL;
}
- if ((flags & ZIP_CHECKCONS) && offset+size != eocd_offset) {
+ if (offset+size > buf_offset + eocd_offset) {
+   zip_error_set(error, ZIP_ER_INCONS, 0);
+   return NULL;
+ }
```

Fig. 14. Disclosed security patch for CVE-2017-14107.

Finding 5. Among the 2000 popular C/C++ OSS projects on GitHub, 95 (4.75%) contained at least one incompletely patched code. Specifically, in some OSS projects, the security of the entire software can be compromised because of incompletely patched codes. This supports our argument regarding the need to identify and address unreliable patches.

4. Patch flexibility analysis

In this section, we examine patch flexibility. An inflexible patch is one in which the disclosed security patch cannot be directly applied to resolve a vulnerability.

Motivating example. A subtle code syntax change results in a case in which the entire disclosed patch cannot be applied. In 2017, a vulnerability that could cause a denial-of-service attack was discovered in `Libzip` (CVE-2017-14107, CVSS 6.5). The PHP team confirmed that the vulnerable code was included in their codebase, and attempted to resolve this vulnerability. However, because PHP reused the older version of `Libzip` with code modifications, the disclosed patch could not be applied directly to vulnerable codes in the PHP codebase. In particular, the code lines added by the disclosed security patch include calls to functions that are not used in PHP (line #12 in Fig. 14). Thus, the PHP team applied the patch by modifying the function call code line to fit their codebase and resolve the vulnerability (line #11 in Fig. 15).

4.1. Inflexible patch detection methodology

Discovering inflexible patches involves the following two steps: (1) detecting propagated vulnerabilities and (2) examining whether disclosed security patches can be applied to detected vulnerabilities. Because code-level vulnerability propagation is predominant in C/C++ languages (as explained in Section 3.4), we focused on the 4420 C/C++ vulnerabilities collected from our dataset (Table 1) to examine the flexibility of the security patches. If a security patch modifies multiple areas of the codebase and the patch is not applied directly in any area, we consider this patch to be inflexible.

4.1.1. Propagated vulnerability detection

To detect propagated vulnerabilities, we used MOVERY (Woo et al., 2022), a precise approach that can detect propagated vulnerabilities even if the syntax is significantly different from the disclosed vulnerable code. Unlike the previous vulnerability propagation detection experiments (see Section 3.4), our goal is to detect vulnerabilities that have propagated with modifications to the code syntax, where security patches cannot be directly applied. To this end, we decided to use MOVERY, which specializes in our objectives.

To this end, we need to extract vulnerable and patched functions from security patches and apply preprocessing. The process of extracting the functions modified in the security patch is described

```
// Commit ID: f6e8ce812174343b5c9fd1860f9e2e2864428567
// PHP/ext/zip/lib/zip_open.c
static zip_cdir_t * _zip_read_eocd64(...)
{
    ...
    _zip_error_set (error, ZIP_ER_SEEK, EFBIG);
    return NULL;
}
- if ((flags & ZIP_CHECKCONS) && offset+size != eocd_offset) {
+ if (offset+size > buf_offset + eocd_offset) {
+   _zip_error_set (error, ZIP_ER_INCONS, 0);
+   return NULL;
+ }
```

Fig. 15. Inflexible patch example in Libzip and PHP.

in Section 3.4.1. After extracting vulnerable and patched functions, by referring to their paper, we applied preprocessing to extract core code lines that are directly related to vulnerabilities, for example, the vulnerable code lines deleted in a security patch (essential code lines), and code lines that have control or data dependencies with the essential code lines (dependent code lines). During this process, it is necessary to generate control and data dependency graphs for each vulnerable and patched function. We used the Joern (Yamaguchi et al., 2014) parser (used by MOVERY), which can perform semantic analysis on the target function. Because the Joern parser provides control and data dependencies between code lines, we could extract the core code lines required by MOVERY.

Finally, we used the core code lines extracted for each security patch as MOVERY's dataset and detected the propagated vulnerabilities from 2000 popular C/C++ OSS projects (Section 2.2.2). To examine a sufficient number of vulnerabilities, we considered an older version of each OSS project released closer to January 2021.

4.1.2. Inflexible patch detection

Inflexible patches were detected by examining whether security patches could be applied without any changes. Let f_p be the propagated vulnerable function. We define that a security patch c_v cannot be directly applied (*e.g.*, using the "`patch`" command) if (1) the deleted code lines in the patch and (2) the code lines near those modified in the patch are not included in the propagated vulnerable function. We used the same notations listed in Table 2.

• ***Condition 1: Containing deleted code lines.*** Every code line that was deleted in the original vulnerable function (f_v) should be included in f_p.

$$\forall\, l \in \mathrm{DEL}(v, v) \mid \left(l \in f_p\right)$$

• ***Condition 2: Containing nearby code lines.*** All n code lines adjacent to the code lines added or deleted in f_v should be included in f_p.

$$\forall\, l \in \mathrm{NR}(l_v, n) \mid l \in f_p, \text{ where } \; l_v \in \left(\mathrm{ADD}(v, v) \cup \mathrm{DEL}(v, v)\right)$$

We set n to three, the default value for GitHub patches. Fig. 16 depicts the high-level workflow of inflexible patch detection.

4.2. Inflexible patch detection results

From 2000 target programs, MOVERY discovered 3989 vulnerable function clones with 693 assigned CVE IDs. Among them, 3222 (80.8%) vulnerable code clones exhibited a different syntax from the disclosed vulnerabilities, *i.e.*, at least one code line was changed. Thereafter, we detected inflexible patches under the conditions in Section 4.1.2. If at least one propagated vulnerable code required patch modification, we considered this security patch inflexible.

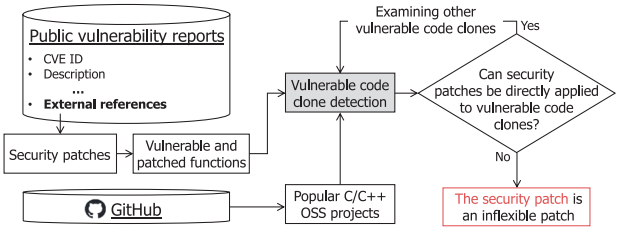


Fig. 16. Overview of inflexible patch detection.

Consequently, we detected 377 inflexible patches; we failed to apply the disclosed patches to 1959 (49.1%) vulnerable code clones (377 CVE IDs assigned) owing to syntax diversity. Specifically, 390 vulnerable code clones that did not include code lines deleted from the security patch, and the remaining 1569 vulnerable code clones did not include the code lines near those modified in the patch. The `Linux kernel` repository contained the highest number of inflexible patches (76 patches), followed by `TCPdump` (28 patches) and `FFmpeg` (21 patches). The `Linux kernel` is frequently modified to suit its purpose, especially when used in various embedded systems or operating systems. For this reason, many inflexible patches might be discovered in the `Linux kernel`.

Finding 6. Many propagated vulnerabilities (80.8%) exhibited a different syntax from the disclosed vulnerable functions. In particular, the security patches for 377 (54.4%) of the 693 CVEs could not be used directly to resolve the propagated vulnerabilities.

4.3. Characteristics of inflexible patches

Here we examined the characteristics of the detected inflexible patches by answering several questions.

4.3.1. What are the causes of syntax diversity?

We observed that the syntax of a propagated vulnerable function could vary for two main reasons.

- **Different branch case.** The vulnerable function of an OSS branch other than the branch where the disclosed vulnerable function exists is reused in the target program; thus, the syntax differs from that of the disclosed vulnerable code (Tan et al., 2022).
- **Code modification case.** The case in which developers modified code lines in the vulnerable function during or after OSS reuse (Woo et al., 2022).

To examine each case, we investigated 1959 vulnerable code clones to which security patches could not be directly applied. If the syntax of a vulnerable function clone exists in any branch of an OSS, it is defined as a *different branch* case; otherwise, it is defined as a *code modification* case. This can be easily verified by extracting all the functions from every version of the OSS included in our dataset.

Consequently, we identified 615 (31.4%) different branch cases and 1344 (68.6%) code modification cases. We observed that developers frequently managed OSS components by reflecting only the necessary code patches (*i.e.*, changing only some code lines in a function) rather than maintaining all OSS components in the latest version, *e.g.*, owing to compatibility issues. Therefore, a considerable number of vulnerable code clones exhibit the syntax of a function that exists in an older OSS branch or a completely different syntax from the disclosed vulnerable function. In addition, several propagated vulnerabilities can be resolved by slightly modifying the disclosed patches, however, there were also cases in which a completely new security patch had to be created.

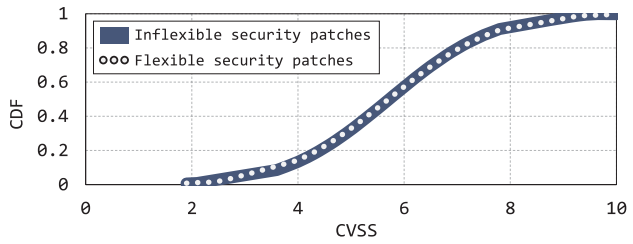


Fig. 17. CDFs of CVSS for flexible and inflexible patches.

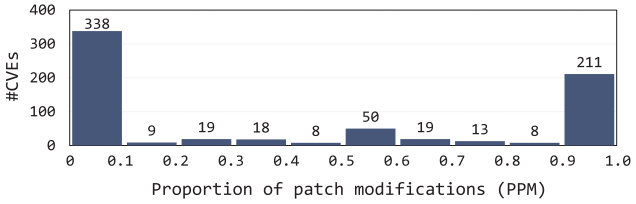


Fig. 18. PPM measurement results.

This result promotes various studies, such as automated patch generation and the detection of vulnerable codes propagated with different syntaxes.

Finding 7. Because developers frequently reuse OSS codes in older branches and modify them during the OSS reuse process, a considerable number of propagated vulnerable codes (49.1%) cannot be resolved by directly applying disclosed security patches.

4.3.2. What is the relationship between the flexibility and severity of security patches?

To answer this question, we first classified 693 CVEs discovered in 2000 target programs as flexible (316 patches) and inflexible (377 patches) and subsequently investigated the CVSS for all CVEs. We examined the relationship between patch flexibility and vulnerability severity by comparing the CVSS distribution between flexible and inflexible patches. Fig. 17 shows the measurement results. We confirmed that the flexible and inflexible security patches exhibited nearly identical CVSS distributions; the average CVSS was 5.7 and 5.72, respectively (a median of five in both cases). The results confirmed that there was no notable relationship between the severity of vulnerabilities and the flexibility of patches in our setup.

4.3.3. What is the proportion of vulnerable clones requiring patch modifications?

For each of the 693 CVEs detected in Section 4.2, we measured the proportion requiring patch modifications among the discovered vulnerable code clones. We define a metric called proportion of patch modifications (PPM) to measure the number of propagated vulnerable codes requiring patch modifications. The higher the PPM, the lower the flexibility of the security patch.

$$\texttt{PPM}(\texttt{CVE}) = \frac{\#\text{ of vulnerable code clones require modified patches}}{\#\text{ of total detected vulnerable code clones for the CVE}}$$

Fig. 18 shows the measured PPMs for 693 CVEs. There are two major cases in which the syntax of vulnerable codes is highly changed (211 CVEs) or hardly changed (338 CVEs). The 211 vulnerabilities, which are difficult to apply disclosed security patches to vulnerable clones, require more attention when attempting to resolve them. Interestingly, the former was mainly observed in different branch cases, and the latter case mostly occurred in code modification cases (Section 4.3.1), indicating that the extent to which developers modified the OSS code during the reuse process was generally less than that occurring during OSS updates.

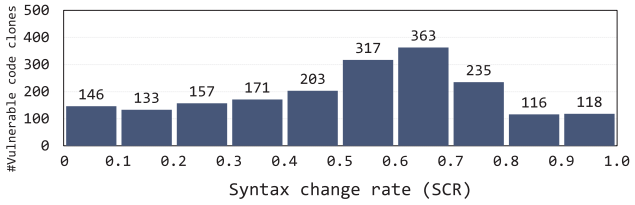


Fig. 19. SCR measurement results.

Finding 8. CVE vulnerabilities exhibited two major patterns as they propagated: either significantly changed (30.4%) or almost the same syntax (48.8%). In the former case, because it is difficult to apply a disclosed security patch to a propagated vulnerable code, more attention is required to resolve the security issue.

4.3.4. How much has the syntax of the propagated vulnerable code been modified?

Next, we examined the degree of syntax changes in vulnerable code clones that require patch modifications. Let V be a set of all code lines in the vulnerable code clone that requires patch modifications, and D be a set of all code lines in the disclosed vulnerable function. After we measure the syntax similarity (0 to 1) between V and D, we calculate the *syntax change rate* (SCR) by subtracting the similarity value from 1. We use Jaccard similarity (Murphy, 1996) to measure the syntax similarity between two functions.

$$\texttt{SCR}(V) = 1 - (|V \cap D|/|V \cup D|)$$

Fig. 19 shows the results of the SCR measurements. For the 1959 vulnerable clones that required patch modifications, the average SCR was 0.51 (with a median of 0.5). Overall, SCRs between 0.5 and 0.7 appeared most frequently (34.7%), and the remaining were evenly distributed. In particular, the SCR appeared higher (*e.g.*, more than 0.5) in different branch cases and tended to be lower (*e.g.*, less than 0.5) in code modification cases. The lowest SCR was 0.01, implying that even though only a small code fragment is changed, the disclosed patch may need to be modified.

Finding 9. The SCRs of the vulnerable code clones requiring patch modifications were the highest between 0.5 and 0.7. Even with small syntax changes (*e.g.*, SCR $\leq$ 0.2), several vulnerable code clones where patches need to be modified were discovered, confirming that more attention is required in patch modifications.

5. Suggestions and applications

In our experiments, we confirmed that 779 (9.6%) disclosed security patches lacked effectiveness; 476 unreliable and 377 inflexible patches were detected (74 patches belonged to both groups). In the case of unreliable patches, 84.6% of them were not disclosed to the public. In the experiments on inflexible patches, we confirmed that many vulnerable codes were propagated with various code syntaxes, thereby requiring patch modifications.

In this section, we present suggestions and introduce possible applications of our findings.

5.1. Suggestions

We present suggestions for mitigating threats posed by ineffective patches, from the perspectives of (1) public vulnerability databases, (2) OSS teams reporting software vulnerabilities, and (3) developers reusing OSS codebases.

(1) **Establishing ongoing vulnerability reporting process.** Because ongoing vulnerability reporting is not mandatory for OSS developers and security analysts, several unreliable and supplementary patches remain unreported in public vulnerability reports. If an ongoing vulnerability reporting process is established instead of a one-time process, more time is obtained to verify the reliability of the security patches; thus, the security of the software ecosystem can be further enhanced by reducing possible attack surfaces.

(2) **Providing security patches across various OSS branches.** To address inflexible patches, OSS teams can provide multiple security patches applicable to each OSS branch; however, for most CVEs, only security patches generated based on the main branch of the OSS are provided (Tan et al., 2022). By generating and providing security patches applicable to each OSS branch, the OSS team can address possible vulnerabilities contained in various branches, and developers who reuse the OSS codebase can resolve the propagated vulnerabilities by simply applying the provided security patches.

(3) **Devising an automated tool to address unreliable patches.** Developers can periodically check the commits of the OSS they are reusing to confirm whether supplementary patches for security patches have been applied. If this task is automated (*e.g.*, using the methodology proposed in this study), software security can be improved by supplementing unreliable security patches with minimal effort.

We believe that our suggestions are neither overly complicated nor unrealistic. In brief, efforts by various stakeholders are required to prevent threats arising from ineffective patches.

5.2. Applications

Our approach for identifying unreliable and inflexible patches and detection results can be applied to various fields.

Security patch collection. One such example is security patch collection. Patch collection studies (*e.g.*, Wang et al., 2021; Tan et al., 2021; Hong et al., 2022) that focus on collecting a wide range of security patches can leverage our unreliable and inflexible patch detection method to complement the collected security patches and expand the patch datasets.

Vulnerable code detection. Many existing approaches in this field (*e.g.*, Jang et al., 2012; Kim et al., 2017; Xiao et al., 2020; Woo et al., 2022) have detected propagated vulnerabilities using the following three steps: collecting security patches, extracting vulnerable codes, and discovering vulnerable code clones. These include incompletely patched vulnerable codes (*e.g.*, still vulnerable) in the vulnerability dataset, thereby discovering more vulnerable code clones in practice, as discussed in Section 3.4.

Verifying data quality of public vulnerability reports. Finally, our results can be used to verify the quality of data in public vulnerability databases (*e.g.*, Mu et al., 2018; Dong et al., 2019; Woo et al., 2021a). Our finding that some disclosed security patches may lack reliability and flexibility can provide new insights into related studies, supplement their results, or foster future studies.

Case study. We describe a case where the method we propose for detecting unreliable patches has been effectively applied to the real-world software ecosystem in practice.

We introduce a case found in the latest version of `Greenplum Database` (`GPDB`)[2], a widely utilized open source data warehouse based on `PostgreSQL`. In 2014, multiple integer overflow vulnerabilities were discovered in `PostgreSQL`; they immediately patched all the discovered vulnerabilities (see Fig. 20). However, in 2020, the

```
// CVE ID: CVE-2014-0064
// PostgreSQL/contrib/ltree/ltree_oi.c
@@ -67,1 +68,7 @@ ltree_in(PG_FUNCTION_ARGS)
+ if (num + 1 > MaxAllocSize / sizeof(nodeitem))
+  ereport(ERROR,
+     (errcode(ERRCODE_PROGRAM_LIMIT_EXCEEDED),
+  errmsg("number of levels (%d) exceeds the maximum allowed (%d)",
+      num + 1, (int) (MaxAllocSize / sizeof(nodeitem)))));
   list = lptr = (nodeitem *) palloc(sizeof(nodeitem) * (num + 1));
```

Fig. 20. Incomplete patch snippet for CVE-2014-0064 in PostgreSQL.

```
// PostgreSQL/contrib/ltree/ltree_oi.c
@@ -61,6 +61,6 @@ ltree_in(PG_FUNCTION_ARGS)
- if (num + 1 > MaxAllocSize / sizeof(nodeitem))
+ if (num + 1 > LTREE_MAX_LEVELS)
   ereport(ERROR,
      (errcode(ERRCODE_PROGRAM_LIMIT_EXCEEDED),
-  errmsg("number of levels (%d) exceeds the maximum allowed (%d)",
-      num + 1, (int) (MaxAllocSize / sizeof(nodeitem)))));
+  errmsg("number of ltree levels (%d) exceeds the maximum allowed
+      (%d)", num + 1, LTREE_MAX_LEVELS)));
   list = lptr = (nodeitem *) palloc(sizeof(nodeitem) * (num + 1));
```

Fig. 21. Supplementary patch snippet for CVE-2014-0064 (commit 95f7dd) identified by our approach.

`PostgreSQL` team found that an excessively long input still causes overflow even in the patched code; therefore, they further added the check code for out-of-range values (see Fig. 21).

We confirmed that the latest version of `GPDB` includes the code applied only up to the incomplete patch. We responsibly reported this issue to the `GPDB` team; they replied that they will fix this vulnerability through updating PostgreSQL in the next version of `GPDB`. As of August 2024, we have confirmed that the vulnerability has been patched in the latest version of `GPDB`.

Our experimental results showed that some popular OSS projects still contain potentially vulnerable code that requires applying a supplementary patch. From this perspective, our approach to identifying unreliable patches can be effectively applied to prevent real threats by integrating it with vulnerable code detection techniques.

6. Discussion

We discuss several considerations related to this study.

Unreliable patch detection coverage. In our experiments, we considered cases in which the supplementary patches changed the code lines to be modified into unreliable patches. This was our decision for the scalable analysis of 8100 security patches. The unreliable patches we have identified undoubtedly require additional patches. However, the critical factor is whether this has an impact on software security. We have confirmed that 63.6% of the detected supplementary patches, including cases where vulnerabilities were triggered only with unreliable patches applied, were implemented owing to security issues.

Meanwhile, 28% of known unreliable patches were complemented differently as confirmed in the preliminary experiment (see Section 3.1.1), and our current study hardly covers such unreliable patches (*i.e.*, false negatives). Several methods can be used to identify unreliable patches that our experiments failed to cover. For instance, after modeling vulnerability manifestation patterns, such as those performed by Huang et al. (2019), we can verify whether these patterns still emerge after applying the disclosed patches. As another example, by leveraging (Kwon et al., 2021), we can test whether the Proofs of Concept for vulnerability still work after applying the disclosed patches. Nevertheless, we believe that our method of detecting 476 previously hidden unreliable patches is valuable from the perspective of examining

[2] https://github.com/greenplum-db/gpdb-archive.

the reliability of security patches on a large scale. Expansion of the detection coverage will be performed in future.

Threats to validity. First, although we consider all collectible patches using clear criteria (Section 2.2), the benchmark security patches used in this study may not be representative. For instance, inflexible patch detection results may be obtained differently depending on the software dataset used to scan the vulnerable code clones. Second, we did not consider false positives and negatives of MOVERY in the inflexible patch discovery because MOVERY exhibited more than 96% precision and recall in their accuracy evaluation (Woo et al., 2022). Nonetheless, depending on the vulnerable code clone detection results of MOVERY, the inflexible patch detection results may also differ. Finally, when detecting inflexible patches, we considered only C/C++ languages because of the issue of available external tools; other languages may exhibit different trends.

Responsible disclosure. For the cases of unreliable patches caused by a security issue, we continue to request to include the commit links of supplementary patches in the "references" of the corresponding CVEs. We also reported the vulnerabilities mentioned in Section 3.4 to the responsible OSS teams. Currently, we have submitted ten reports; although most of the response teams were providing negative answers (*e.g.*, no obligation to modify CVE because the code is safe in the latest version), we plan to continue with the request.

7. Related works

In this section, we introduce a number of related studies.

Analyzing the effectiveness of security patches. Several studies have examined the reliability of security patches. Liu et al. (2020) and Piantadosi et al. (2019) evaluated the security of real-world OSS projects and observed that an incomplete patch significantly impacted the recurrence of vulnerabilities. Li and Paxson (2017) evaluated more than 4000 disclosed security patches and manually analyzed 26 incomplete and 17 regressive security patches. Park et al. (2012) examined bugs that were fixed multiple times in three OSS projects, and observed that majority of them were caused by incomplete patches. All were aware of the importance of unreliable patch analysis, and some attempted to detect unreliable patches manually. However, they are limited in their ability to discover and examine unreliable patches from thousands of security patches in an automated manner, as was performed in this study.

In addition, some studies have attempted to analyze the flexibility of security patches. Frei et al. (2006), Shahzad et al. (2012), Farhang et al. (2019), and Alexopoulos et al. (2022) conducted patch studies focusing on the life cycle of vulnerabilities. Zhang et al. (2021) and Jiang et al. (2020) examined the patch propagation from Linux kernels to downstream vendors. Tan et al. (2022) investigated the security patch management within multiple branches of a single OSS project. They provided valuable insights into the propagation of security patches, but none investigated the flexibility of security patches from the perspective of resolving vulnerabilities propagated to a wide range of OSS projects.

Verifying public vulnerability reports. Several studies have attempted to verify the data quality of public vulnerability reports. Woo et al. (2021a) discovered the correct origin of vulnerabilities to resolve propagated vulnerabilities in a timely manner. Bettenburg et al. (2008), Chaparro et al. (2017), and Nappa et al. (2015) attempted to identify missing information in public vulnerability reports to help mitigate vulnerabilities. Guo et al. (2010) analyzed the characteristics of public reports to effectively resolve bugs, and Mu et al. (2018) suggested that the current public vulnerability reports are insufficient to reproduce and analyze vulnerabilities. Dong et al. (2019) attempted to alleviate the inconsistencies between descriptions and affected software information in public vulnerability reports. Although their goal was to verify the data quality of public vulnerability reports, none of the studies attempted to examine the reliability of disclosed security patches. If the disclosed security patches lack reliability, the credibility of their findings (*e.g.*, Woo et al., 2021a) may also be impaired. Therefore, our study examines public vulnerability reports from a different perspective to complement their results (Section 6).

8. Conclusion

In this study, we assessed the effectiveness of 8100 security patches obtained from an NVD, with 4538 OSS repositories. In our experiments, we found that one in ten security patches lacked effectiveness. Although unreliable patches can compromise the security of an entire system, supplementary patches are rarely disclosed (15.4%) through public vulnerability reports. Moreover, in many cases (49.1%), we confirmed that security patches cannot be directly applied to propagated vulnerable code owing to syntax diversity. Our findings on the effectiveness of security patches, which have not been extensively investigated, will contribute toward improving the data quality of public vulnerability databases and help developers mitigate potential threats. The experimental data and results are publicly available at https://github.com/WOOSEUNGHOON/COSE_patchStudy.

CRediT authorship contribution statement

Seunghoon Woo: Writing – original draft, Validation, Software, Methodology, Conceptualization. **Eunjin Choi:** Validation, Methodology, Conceptualization. **Heejo Lee:** Writing – review & editing, Project administration, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP), Republic of Korea grant funded by the Korea government (MSIT) (No. RS-2022-II220277, Development of SBOM Technologies for Securing Software Supply Chains, No. IITP-2024-RS-2022-II221198, Convergence Security Core Talent Training Business (Korea University), IITP-2024-RS-2020-II201819 (10%) ICT Creative Consilience program, and No. RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains). In addition, this research was supported by Korea Creative Content Agency, Republic of Korea grant funded by the Ministry of Culture, Sports and Tourism in 2024 (Project Name: International Collaborative Research and Global Talent Development for the Development of Copyright Management and Protection Technologies for Generative AI, Project Number: RS-2024-00345025).

Data availability

The dataset and results of our study are publicly available on GitHub https://github.com/wooseunghoon/COSE_patchStudy.

References

Alexopoulos, N., Brack, M., Wagner, J.P., Grube, T., Mühlhäuser, M., 2022. How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes. In: 31st USENIX Security Symposium. USENIX Security 22, pp. 359–376.

An, L., Khomh, F., Adams, B., 2014. Supplementary bug fixes vs. Re-opened bugs. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 205–214.

Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S., 2008. Duplicate bug reports considered harmful... Really? In: 2008 IEEE International Conference on Software Maintenance. pp. 337–345.

Chaparro, O., Lu, J., Zampetti, F., Moreno, L., Di Penta, M., Marcus, A., Bavota, G., Ng, V., 2017. Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE, pp. 396–407.

Common Weakness Enumeration, 2023. 2023 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

Ctags, 2022. Universal ctags. https://github.com/universal-ctags/ctags.

Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., Wang, G., 2019. Towards the detection of inconsistencies in public security vulnerability reports. In: 28th USENIX Security Symposium. USENIX Security 19, pp. 869–885.

Farhang, S., Kirdan, M.B., Laszka, A., Grossklags, J., 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.

Frei, S., May, M., Fiedler, U., Plattner, B., 2006. Large-scale vulnerability analysis. In: Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense. pp. 131–138.

Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B., 2010. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proceedings of the 32nd International Conference on Software Engineering. ICSE, pp. 495–504.

Hong, S., Lee, J., Lee, J., Oh, H., 2020. SAVER: Scalable, precise, and safe memory-error repair. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 271–283.

Hong, H., Woo, S., Choi, E., Choi, J., Lee, H., 2022. xVDB: A high-coverage approach for constructing a vulnerability database. IEEE Access 10, 85050–85063.

Hong, H., Woo, S., Lee, H., 2021. Dicos: Discovering insecure code snippets from stack overflow posts by leveraging user discussions. In: Annual Computer Security Applications Conference. ACSAC, pp. 194–206.

Huang, Z., Lie, D., Tan, G., Jaeger, T., 2019. Using safety properties to generate vulnerability patches. In: 2019 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 539–554.

Islam, M.R., Zibran, M.F., 2021. What changes in where? An empirical study of bug-fixing change patterns. ACM SIGAPP Appl. Comput. Rev. 20 (4), 18–34.

Jaccard, P., 1912. The distribution of the flora in the alpine zone. 1. New Phytol. 11 (2), 37–50.

Jang, J., Agrawal, A., Brumley, D., 2012. ReDeBug: Finding unpatched code clones in entire OS distributions. In: 2012 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 48–62.

Jiang, Z., Zhang, Y., Xu, J., Wen, Q., Wang, Z., Zhang, X., Xing, X., Yang, M., Yang, Z., 2020. Pdiff: Semantic-based patch presence testing for downstream kernels.. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. CCS, pp. 1149–1163.

Kang, W., Son, B., Heo, K., 2022. TRACER: Signature-based static analysis for detecting recurring vulnerabilities. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. CCS, pp. 1695–1708.

Kim, M., Sinha, S., Görg, C., Shah, H., Harrold, M.J., Nanda, M.G., 2010. Automated bug neighborhood analysis for identifying incomplete bug fixes. In: 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, pp. 383–392.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 595–614.

Kwon, S., Woo, S., Seong, G., Lee, H., 2021. OCTOPOCS: Automatic verification of propagated vulnerable code using reformed proofs of concept. In: Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 174–185.

Le, X.-B.D., Bao, L., Lo, D., Xia, X., Li, S., Pasareanu, C., 2019. On reliability of patch correctness assessment. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 524–535.

Lee, J., Hong, S., Oh, H., 2018. MemFix: Static analysis-based repair of memory deallocation errors for C. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 95–106.

Li, F., Paxson, V., 2017. A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS, pp. 2201–2215.

Liu, B., Meng, G., Zou, W., Gong, Q., Li, F., Lin, M., Sun, D., Huo, W., Zhang, C., 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. ICSE, IEEE, pp. 1547–1559.

Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G., 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In: 27th USENIX Security Symposium. USENIX Security 18, pp. 919–936.

Murphy, A.H., 1996. The finley affair: A signal event in the history of forecast verification. Weather Forecast. 11 (1), 3–20.

Na, Y., Woo, S., Lee, J., Lee, H., 2024. CNEPS: A precise approach for examining dependencies among third-party C/C++ open-source components. In: Proceedings of the 46th International Conference on Software Engineering. ICSE, pp. 2918–2929.

Nappa, A., Johnson, R., Bilge, L., Caballero, J., Dumitras, T., 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In: 2015 IEEE Symposium on Security and Privacy. SP, pp. 692–708.

Park, J., Kim, M., Ray, B., Bae, D.-H., 2012. An empirical study of supplementary bug fixes. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. MSR, IEEE, pp. 40–49.

Piantadosi, V., Scalabrino, S., Oliveto, R., 2019. Fixing of security vulnerabilities in open source projects: A case study of apache HTTP server and apache tomcat. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification. ICST, IEEE, pp. 68–78.

Shahzad, M., Shafiq, M.Z., Liu, A.X., 2012. A large scale exploratory analysis of software vulnerability life cycles. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 771–781.

Shi, Y., Zhang, Y., Luo, T., Mao, X., Yang, M., 2022. Precise (un) affected version analysis for web vulnerabilities. In: 37th IEEE/ACM International Conference on Automated Software Engineering. ASE.

Tan, X., Zhang, Y., Cao, J., Sun, K., Zhang, M., Yang, M., 2022. Understanding the practice of security patch management across multiple branches in OSS projects. In: Proceedings of the ACM Web Conference 2022. WWW, pp. 767–777.

Tan, X., Zhang, Y., Mi, C., Cao, J., Sun, K., Lin, Y., Yang, M., 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, pp. 3282–3299.

Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S., 2021. PatchDB: A large-scale security patch dataset. In: Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 149–160.

Woo, S., Choi, E., Lee, H., Oh, H., 2023. V1SCAN: Discovering 1-day vulnerabilities in reused c/c++ open-source software components using code classification techniques. In: 32nd USENIX Security Symposium. USENIX Security 23, pp. 6541–6556.

Woo, S., Hong, H., Choi, E., Lee, H., 2022. MOVERY: A precise approach for modified vulnerable code clone discovery from modified open-source software components. In: 31st USENIX Security Symposium. USENIX Security 22, pp. 3037–3053.

Woo, S., Lee, D., Park, S., Lee, H., Dietrich, S., 2021a. V0Finder: Discovering the correct origin of publicly reported software vulnerabilities. In: 30th USENIX Security Symposium. USENIX Security 21, pp. 3041–3058.

Woo, S., Park, S., Kim, S., Lee, H., Oh, H., 2021b. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 860–872.

Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., Shi, W., 2020. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 1165–1182.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 590–604.

Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., Liu, Y., 2021. Research on third-party libraries in android apps: A taxonomy and systematic literature review. IEEE Trans. Softw. Eng..

Zhang, Z., Zhang, H., Qian, Z., Lau, B., 2021. An investigation of the android kernel patch ecosystem. In: 30th USENIX Security Symposium. USENIX Security 21, pp. 3649–3666.

Seunghoon Woo received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Korea University. He served as Chief Scientist at Labrador Labs from 2022 to 2023. He has published papers on software security and software engineering in top conferences, such as S&P, USENIX Security, and ICSE. His research interests include software security, vulnerability detection, and supply chain security.

Eunjin Choi received the B.S. degree in Computer Science and Engineering from Inha University, in 2020, and M.S. degree in the Department of Computer Science and Engineering from Korea University, Seoul, South Korea, in 2022. Her research interests include vulnerability detection, vulnerability analysis, and digital forensics.

Heejo Lee (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from POSTECH, South Korea. He is currently a Professor with the Department of Computer Science and Engineering, Korea University, and the Director of the Center for Software Security and Assurance (CSSA). He is a Founding Member and the Co-CEO of Labrador Labs. Before joining Korea University, he was the CTO with AhnLab Inc., from 2001 to 2003, and a Postdoctoral Researcher with Purdue University, from 2000 to 2001. He is the Editor of the Journal of Communications and Networks and the IEEE Transactions on Vehicular Technology.