# Unveiling the Characteristics and Impact of Security Patch Evolution

Zifan Xie*†
Huazhong University of Science and Technology, Wuhan, China
xzff@hust.edu.cn

Ming Wen*†‡
Huazhong University of Science and Technology, Wuhan, China
mwenaa@hust.edu.cn

Zichao Wei*†
Huazhong University of Science and Technology, Wuhan, China
zcwei@hust.edu.cn

Hai Jin*§
Huazhong University of Science and Technology, Wuhan, China
hjin@hust.edu.cn

## ABSTRACT

The number of disclosed vulnerabilities in open-source projects has been increasing steadily over the years, and thus it is important to deploy patches to repair security vulnerabilities in a timely manner. However, due to the widespread reuse and customization of open-source software, there are often multiple versions or branches of the same project that co-exist in the ecosystem. Therefore, it is often challenging and tricky to guarantee that an exposed vulnerability can be repaired thoroughly. Driven by this, plenty of 1-day vulnerability analysis tools have been proposed recently, such as function-level vulnerability detection and patch presence test tools. Despite the fact that code evolution is common for open-source projects, existing analysis tools often neglect the important fact that the patched code is also constantly evolving. In this study, we take the first look to systematically *investigate the phenomenon of security patch evolution in open-source projects*. In particular, we performed extensive experiments on a large-scale dataset containing 1,046 distinct CVEs with 2,633 patches collected from popular open-source projects (e.g., linux, openssl). This study reveals interesting yet important findings with respect to the aspects of patch evolution frequency, patch evolution patterns, and the evolution impact on downstream 1-day vulnerability analysis tools. We believe that this study can shed important light on future researches on patch analysis.

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
‡Corresponding author
§Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

## KEYWORDS

Vulnerability Analysis, Security Patch, Code Evolution

**ACM Reference Format:**
Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. 2024. Unveiling the Characteristics and Impact of Security Patch Evolution. In . ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Software vulnerability is one of the major threats to cyber security. A recent report by Cybersecurity Ventures shows that cybercrime will cost the world $8 trillion's loss in 2023. Unfortunately, open-source software (OSS) often suffer from abundant and diverse vulnerabilities, and the number of disclosed vulnerabilities in open-source projects has been increasing steadily since 2009 [1, 13, 34]. Such vulnerabilities can cast significant security threats to the whole ecosystem. Therefore, patching and repairing exposed vulnerabilities in a timely manner is of significant importance to avoid potentially catastrophic consequences.

However, due to the widespread reuse and customization of open-source software, there are often multiple versions or branches of the same project co-existing in the ecosystem. Therefore, it is challenging and tricky to guarantee that an exposed vulnerability can be repaired thoroughly. To address this problem, many ***function-level vulnerability detection*** and ***patch presence test*** approaches have been proposed [6, 9, 12, 36, 51]. The former type of such works aims to detect the enclosed libraries of a project to see if it contains existing vulnerabilities; while the latter aims to check whether a known vulnerability has been patched in the target software. All of the above techniques function with a dataset of known vulnerabilities with the corresponding patches, which are often collected from the National Vulnerability Database (NVD [25]) as well as the code repositories hosted on GitHub. Commonly, the *commit* that fixes the corresponding vulnerability is denoted as the *patch*, and the version before the *commit* is denoted as the *vulnerable code* while the version after it is denoted as the *patched code*. Most of the existing analysis approaches are carried out based on the patches and vulnerable code, and have demonstrated promising effectiveness in guaranteeing the security of open-source ecosystems. Unfortunately, they often neglect the important fact that, similar to conventional software, the patched code is constantly evolving as well. Recent studies have revealed that the performance of existing works has degraded significantly in the real scenarios where the patched code evolves [52, 53].

In this study, we take the first look to systematically *investigate the phenomenon of security patch evolution in open-source projects*. In particular, we are interested in knowing how often security patches evolve, what are the characteristics, and the impact of patch evolution on downstream 1-day vulnerability (publicly disclosed vulnerabilities) analysis tools [7, 8, 15, 23, 39, 45–47, 52] More importantly, we further investigate whether the characteristics of security patch evolution can be exploited to detect new vulnerabilities. To achieve the above goals, we first introduce the concepts and definitions related to patch evolution (see Section 3.2) and perform extensive experiments on a large dataset of security patches collected from open-source projects. In particular, our dataset consists of 1,046 distinct CVEs with 2,633 patches that are collected from 10 large-scale open-source projects (e.g., *linux, FFmpeg* and *openjpeg*). Based on such a dataset, we investigate the problem of *security patch evolution* from the following perspectives.

***Security Patch Evolution Frequency.*** We first investigate how often security patches evolve, and whether it is pervasive among popular OSS projects. To achieve such a goal, we track the evolution of security patches via mining the corresponding repository. Our study reveals that patch evolution is pervasive, occurring in over 81.1% of the collected CVEs. More importantly, quite a few CVE patches (29.6%) undergo the first evolution quickly, particularly within the first 90 days. Such cases would cause a significant potential impact on the security insurance in the whole ecosystem since downstream software often backports patches based on the original versions, as provided in the patch links from NVD, while ignoring the subsequent evolved versions. Therefore, we are further motivated to investigate and understand such potential impact.

***Security Patch Evolution Characteristics.*** We then explore the characteristics of security patch evolution, in particular, with respect to the changes of the patching code authorship as well as the code change patterns. Prior research [3] has revealed that the more developers contributing to the same piece of code, the higher risk of the code to be buggy. Such observation motivates us to examine changes in authorship during the patch evolution. Our study shows that the authorship of 93.2% of the CVE patches has changed, suggesting possible risks to induce new security issues during the evolution since the non-original developers might be unaware of the intention of the patched code. In terms of the code characteristics, our study reveals that *condition* and *relational* code structures have been modified the most during patch evolution, mainly to refine the original constraints of the patch. Some projects (e.g., linux) also often change *lock_api* and *system_api* to enhance the correctness of the system.

***Security Patch Evolution Impact.*** Most 1-day vulnerability tools overlook security patch evolution, leading to compromised effectiveness [52, 53]. For instance, FIBER picks only the initial version of the patched function to generate patch signatures [52], while its accuracy dropped significantly compared to what was reported in the original paper due to patch evolution as indicated by a recent study [53]. Motivated by this, we further investigate to what extent security patch evolution impacts existing 1-day vulnerability analysis tools. Our study reveals that as the patch evolves, the false positive rates of function-level vulnerability detection tools SAFE [23] and jTrans [39] notably increase, rising from 0.15 to 0.28 and 0.09 to 0.19 respectively. Similar trends have also been observed for patch presence test tools, such as PMatch [15] and BinXray [47]. Such results reveal that existing 1-day vulnerability analysis tools are significantly limited in handling the problem of evolving patches.

Our empirical study reveals a concerning trend: during software evolution, developers often inadvertently alter the patches' logic. Such modifications, while being well-intentioned, might inadvertently induce new security issues. To validate this hypothesis, we further perform a case study to demonstrate that modifications to the patch can indeed induce new vulnerabilities. These findings underscore the importance of carefully monitoring and scrutinizing patch evolution, ensuring that efforts in modifying patches do not compromise their original security objectives or induce new issues.

To summarize, we make the following major contributions:

- **Perspective.** We are the first to systematically investigate the phenomenon of security patch evolution in OSS projects via performing extensive empirical evaluations.
- **Empirical Evaluation.** Our extensive empirical study investigates important aspects, including how often security patches evolve, what are the evolution code patterns, and the impact of patch evolution on downstream 1-day vulnerability analysis tools. Our study on a large-scale dataset yields interesting yet significant findings.
- **Artifacts.** We open-sourced our collected patch dataset (i.e., 1,046 distinct CVEs together with 2,633 patches) as well as the corresponding code evolutions. All the artifacts can be accessed at:

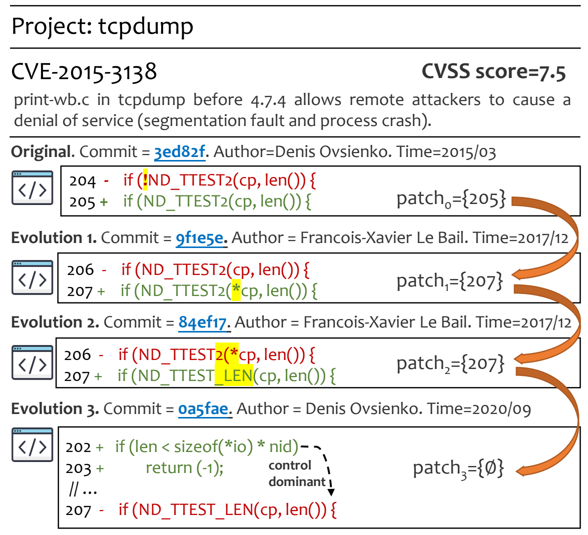
**https://github.com/PatchEvolution/PatchEvolution**

## 2 BACKGROUND AND MOTIVATION

Security patches are critical updates designed to fix vulnerabilities in software systems, thereby guaranteeing their security against potential cyber threats. Numerous 1-day vulnerability detection tools predominantly depend on security patches to discern vulnerabilities in the target software [21, 22, 47, 49], which contain the two main categories. First, ***function-level vulnerability detection*** tools [8, 11, 23, 33, 39, 54] work on the principle that vulnerabilities often have common characteristics or patterns that can be detected through code semantic comparison or pattern recognition. Specifically, if a function bears resemblance to a vulnerable function (i.e., the pre-patch version of the function) in the target software, such a function can be perceived as potentially vulnerable. Second, ***patch presence test*** tools [7, 12, 15, 47, 53] aim to identify unpatched vulnerabilities. If the target software lacks the patch addressing a specific CVE, it could be potentially exposed to security risks, thereby providing an opportunity for attackers.

Although existing tools have achieved promising results in detecting 1-day vulnerabilities, it has been noted that software/patch evolution can threaten the performance of these tools [53]. Motivated by this, we take the first look to systematically investigate the phenomenon of security patch evolution in open-source projects.

**Patch Evolution.** Open-source projects, including their security patches, evolve over time. To demonstrate how security patches are constantly modified after their initial release, we use a real example


Project: tcpdump

CVE-2015-3138 CVSS score=7.5

print-wb.c in tcpdump before 4.7.4 allows remote attackers to cause a denial of service (segmentation fault and process crash).

Original. Commit = 3ed82f. Author=Denis Ovsienko. Time=2015/03

```
204 -  if (!ND_TTEST2(cp, len()) {
205 +  if (ND_TTEST2(cp, len()) {
```

$patch_0=\{205\}$

Evolution 1. Commit = 9f1e5e. Author = Francois-Xavier Le Bail. Time=2017/12

```
206 -  if (ND_TTEST2(cp, len()) {
207 +  if (ND_TTEST2(*cp, len()) {
```

$patch_1=\{207\}$

Evolution 2. Commit = 84ef17. Author = Francois-Xavier Le Bail. Time=2017/12

```
206 -  if (ND_TTEST2(*cp, len()) {
207 +  if (ND_TTEST_LEN(cp, len()) {
```

$patch_2=\{207\}$

Evolution 3. Commit = 0a5fae. Author = Denis Ovsienko. Time=2020/09

```
202 +  if (len < sizeof(*io) * nid)
203 +      return (-1);
// ...
207 -  if (ND_TTEST_LEN(cp, len()) {
```

$patch_3=\{\emptyset\}$

**Figure 1: The evolution of security patch of CVE-2015-3138.**

in Figure 1 for demonstration. It shows the patch (commit=3ed82f) for CVE-2015-3138 [4], a vulnerability in tcpdump [37], which allows users to capture or filter TCP/IP packets that pass through the network interface. The attacker can cause tcpdump denial of service by a carefully crafted network packet. However, we note that this patch was modified twice (commit=`9f1e5e` and `84ef17`) by another author after its release. Finally, the patch snippets were deleted with a new sanitizer checker added (commit=`0a5fae`) by the original author.

In this example, we define the modifications made to the original CVE patch throughout the software evolution process as *patch evolution*. Throughout the process of patch evolution, two main changes may occur: (1) **direct changes**: the syntax of the patch itself is modified (e.g., the line of the original patch changes in commit *9f1e5e*, Figure 1. (2) **indirect changes**: the context related to the patch (i.e., the code that exhibits data/control flow relationships) is modified (e.g., an `if` statement that dominates the patch code is added in commit *0a5fae*, Figure 1.

Such patch evolution can cause severe potential side effects: First, prior research [3] has revealed that the more developers contribute to the same piece of code, the higher risk of the code containing defects. Similarly, subsequent authors modifying a patch might be unaware that the associated code aims to fix a vulnerability issue, let alone the intention of the repair logic. This lack of awareness could unintentionally disrupt the patch's fix logic, thus potentially inducing new vulnerabilities. Second, downstream softwares often backport patches to other branches based on the patches' original versions, as provided in the patch URL links from the NVD. However, our research reveals that 29.6% of the patches are altered by other developers within 90 days of their release (see Section 5.1), possibly to enhance the original patch's fix logic. If the fix logic of the original patch is flawed, it could jeopardize the security of downstream software that ported the original patches. Therefore, it motivates us to systematically investigate the impact of patch evolution on downstream tasks. In this study, we mainly investigate two types of 1-day vulnerability analysis tools based on security patches, which are introduced as follows.

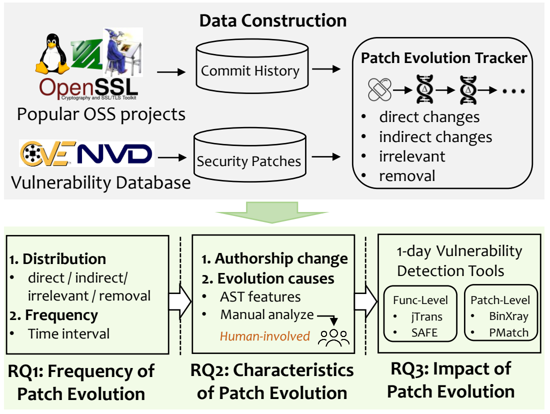


**Figure 2: Overview of this study**

**Function-Level Vulnerability Detection.** Vulnerability detection is considered one of the main applications of function similarity comparing approaches [22, 39]. Specifically, given a patch-related function in the target software, we consider it to be affected by a vulnerability if it is more similar to post-patch reference and less similar to pre-patch reference [8, 39], the idea of which is inspired by an existing work [7]. In recent years, numerous learning-based function similarity methods have been proposed. For instance, jTrans [39] employs a Transformer-based approach to learn binary code representations by integrating control flow information into language models. This is accomplished via a jump-aware representation of the analyzed binaries and a specially designed pre-training task. SAFE [23] utilizes an RNN architecture with attention mechanisms to generate a representation of the analyzed function using assembly instructions as input.

**Patch Presence Test.** The Patch Presence Test is designed to deduce whether a binary includes the patch for a specific vulnerability [7, 12, 15, 47, 53]. For instance, PDiff [12] carries out the patch presence test for downstream Linux kernels. It achieves such a goal by enumerating method paths, extracting their semantics, and determining the presence of patches based on path similarity. BinXray [47], another advanced tool, generates patch signatures by comparing pre-patch/post-patch functions, using basic block matching. Such patch signatures are then utilized to match target functions, in order to ascertain if they have been patched. PMatch [15] is a learning-based approach for detecting patches in binary functions. It works by extracting code snippets affected by patches, creating semantic representations of binary code using unsupervised sentence embedding, and subsequently matching these snippets with target blocks derived from function diffing.

The aforementioned tools show promising results but often overlook the ongoing evolution of security patches. Recent studies reveal significant performance drops when patched code evolves [52, 53]. This prompts our systematic study on the effects of security patch evolution on these tools.

## 3 STUDY DESIGN

Figure 2 provides an overview of the empirical investigations performed in this study. We begin our study by selecting 10 prominent open-source projects (e.g., *linux*, *FFmpeg*), each representing different functionalities (see Section 3.1). We gather the vulnerabilities related to these projects by querying the National Vulnerability Database (NVD [25]), with each vulnerability assigned a unique CVE number. To gather the security patches for these vulnerabilities, we query a public dataset known as PatchDB [41]. Additionally, we extract all the code commits of the target projects from GitHub to further support our detailed patch evolution tracking. Further, we locate the security patch in each evolved patch-related function by tracing the modifications from the original patch throughout the whole software evolution process.

Our research offers the first comprehensive study on security patch evolution in OSS projects. We focus on understanding the frequency and characteristics of security patch evolution.

## 3.1 Dataset Collection

We chose projects from PatchDB [41] since this study aims to investigate patch evolution. PatchDB is a widely recognized security patch dataset which contains over 12K CVE patches. We select those projects that contain the most number of CVE patches in PatchDB while meeting the following criteria: (1) *ease of compilation*: for our subsequent empirical studies, we need to compile different versions of the projects before and after applying the target patches, as well as other evolution versions, to assess the performance of existing vulnerability detection tools. Therefore, the projects should be easily compilable. (2) *complete evolution history*: we need to track patch evolutions based on OSS commit histories, so the selected projects must be continuously and actively maintained through version control systems like Git. (3) *frequent updates*: projects that are not frequently updated might contain limited evolutions to each CVE patch, making them less likely the ideal candidates for investigating the problem of patch evolution. Eventually, we selected ten C/C++ projects, including Linux [30], FFmpeg [29], OpenSSL [31], and other large-scale projects. These projects cover a wide range of categories and functionalities. For instance, OpenSSL is a robust implementation of secure communication protocols, php-src [27] is the official interpreter for the PHP language, and the Linux kernel represents an operating system. Moreover, we also found that these projects have already been extensively investigated by existing studies [35, 41, 47], making them representative choices. On the other hand, we excluded projects that did not meet our criteria. For example, despite the high star count of some projects (e.g., PhantomJS [26]), they have been discontinued and are no longer actively maintained, so they were excluded from our study. By selecting these projects, we ensure that a broader array of vulnerability types is included in our research. Table 1 summarizes the dataset we collected, which contains the number of Stars, Forks, Commits, CVEs, and patches for each project.

## 3.2 Definitions

To ease the understanding of tracking patch evolutions, we introduce the following concepts and definitions.

**Security Patch**. Following an existing study [50], a security patch refers to modifications in the code that replace vulnerable logic with safe code. Typically, a patch usually consists of one or several hunks. A patch hunk is a basic unit which consists of context statements, deleted statements, and/or added statements [44]. In particular, *context statements* denote those surrounding statements of deleted and added statements with control and data dependencies [50]. Typically, the involved operations made to each hunk can be deleted, added, and changed depending on the nature of modifications. A change type hunk typically involves modified statements, incorporating both deletion and addition operations.

**Table 1: Dataset statistics of the CVEs used in this study**

| Projects | #Stars | #Forks | #Commits | #CVEs | #Patches |
|---|---|---|---|---|---|
| FFmpeg | 37.4K | 11.3K | 110.6K | 154 | 219 |
| ImageMagick | 9.6K | 1.2K | 21.3K | 34 | 47 |
| libxml2 | 469 | 332 | 6.0K | 19 | 46 |
| linux | 154.8K | 49.0K | 1.2M | 555 | 1,490 |
| openjpeg | 881 | 433 | 3.0K | 10 | 20 |
| openssl | 22.3K | 9K | 33.0K | 32 | 59 |
| php-src | 35.7K | 7K | 132.6K | 64 | 278 |
| qemu | 8.1K | 4.8K | 105.6K | 70 | 246 |
| radare2 | 18.4K | 2.9K | 30.1K | 29 | 77 |
| tcpdump | 2.3K | 794 | 7.2K | 79 | 151 |
| **total** | **289.8K** | **88.0K** | **1.6M** | **1,046** | **2,633** |

**Patch-related Functions**. A patch-related function refers to the function where a patch hunk resides. Since a security patch might contain multiple hunks, it also often contains multiple patch-related functions, and we denote them as a set $\mathcal{F} = \{f_1, f_2, ..., f_m\}$. In this study, we choose functions as the carrier for tracking the evolution of patches since functions represent cohesive units of code that encapsulates specific behaviors, making it easier to observe and understand the impact of patch changes (e.g., identifying direct or indirect changes via performing control and data flow analysis).

**Patch Evolution.** Suppose $patch_0$ denotes the original security patch, and we can recognize the patch-related functions $\mathcal{F}$ based on the modifications it makes. We then track the patch evolution by tracing the evolution of those patch-related functions via mining the complete version histories of the project. In particular, suppose $C = \{c_1, c_2, ..., c_n\}$ denotes those commits made after the original patch to any function in $\mathcal{F}$ chronologically. By investigating the modifications made in each commit $c_i$, we can track the evolutions of $patch_0$ and obtain the corresponding evolved patch versions $\{patch_1, \dots, patch_i, \dots, patch_n\}$. Here, $patch_i$ represents the i-*th* version of the original patch, which records the location and modifications of the evolved patch. Specifically, via investigating the modifications made by each commit $c_i$ (i.e., whether it modifies the added/deleted statements directly or the context statements as defined in Security Patch), we can recognize four main types of changes to the patch: *direct changes*, *indirect changes*, *irrelevant changes* and *removal.* We provide a detailed explanation of the patch tracking process in Section 3.3.

## 3.3 Patch Evolution Tracker

Precisely tracking the evolution of patches is the key to facilitating such a large-scale study. Specifically, to track the evolution of security patches, we mine the repository's commit histories to trace the changes of the corresponding patch-related functions.

**Identifying patch-related functions**. We first identify patch-related functions as defined in Section 3.2. Specifically, given a security patch, we first identify the patch-related files by the header lines starting with `---` and +++ in the patch diff. To gather all the commits that change the patch-related files, we use the `git log` command with the option `--follow` for patch-related files. To identify the patch-related functions, we extract patch hunks of the patch and identify the changed locations. We then identify the boundaries of each function on patch-related files via clang parser [38] (i.e., the starting and ending statements of the functions). Finally, the patch-related functions $\mathcal{F} = \{f_1, f_2, ..., f_m\}$ can be identified via mapping these modified line numbers to the boundaries of each function. We then identify commits $C = \{c_1, c_2, ..., c_n\}$ that modify any of the patch-related functions based on the modifications of each commit. For each security patch in dataset 1, we iterate through all subsequent commits (from commit $c_1$ to $c_n$) that modify patch-related functions to identify the evolved patch.

**Patch Evolution Tracker**. We detail the process to track patch evolution as follows. We start our tracking based on $patch_0$, especially those statements added by $patch_0$. We focus on the addition statements as they encapsulate the crux of the bug-fixing logic, such as adding a sanitizer checker to check the runtime state of the program. Besides, deletion lines are not visible in subsequent versions, so they cannot be tracked in subsequent commits. We then aim to track the evolution and obtain $patch_i$ based on $c_i$'s modification to $patch_{i-1}$ and identify the location of evolved patches for each commit. Eventually, we can obtain a sequence of patch versions, including $patch_1$, ..., $patch_i$, ... , $patch_n$ chronologically, where $patch_0$ denotes the original patch, and $patch_i$ denotes the $ith$ version by incorporating commit $c_i$. Specifically, there are four main types of changes made by $c_i$:

(1) **direct changes**: denote those modifications (including statement changes and deletions) that are made to the statements of $patch_{i-1}$ directly. To detect change statements, we take line 206 and 207 in Evolution 1, Figure 1 as an example. Line 207 exhibits a subtle modification to line 206, while the *diff* command interprets this as a line deletion (for the old line, line 206) and a line addition (for the new line, line 207). This example motivates us to propose a heuristic-based approach to identifying change statements. Specifically, we consider adjacent deletion and addition lines with similar semantics to represent a change statement. More specifically, when the similarity between an adjacent pair of deletion and addition lines exceeds a threshold, denoted as $T_{LineSimilarity}$, we treat these two lines as a single change statement. Otherwise, we consider it a line deletion if the similarity is below the threshold or the statements are directly removed. In this study, we set $T_{LineSimilarity}$ to 0.7 following an exist study [7]. Then, we will update the $patch_i$ based on its modifications on $patch_{i-1}$ and use it for subsequent tracking iterations.

(2) **indirect changes**: denote those modifications made on context statements of $patch_{i-1}$. Specifically, the modified lines in the commit can exhibit data or control dependencies with $patch_{i-1}$ (e.g., the `if` statement on line 202 in commit `0a5fae` has a control dependency with line 207 in Figure 1). In particular, the dependency analysis is performed based on Joern [48]. We include *indirect changes* in our analysis since many studies have pointed out that the root cause of a defect might reside in those locations that possess control/data dependencies with the patched code [2, 14, 43], and thus tracking the evolution of such code is essential.

(3) **irrelevant**: denote those modifications made on patch-related functions, but have no direct or indirect relation to $patch_{i-1}$. As software evolves, developers will add new features or refactor existing code, improving its structure and readability and enhancing maintainability. Although such irrelevant changes do not modify the security patch and its context, we also include them in our study to understand how frequently patch-related functions evolve.

(4) **removal**: The commit deletes the patch-related functions or the files containing them. Note that the renaming of the patch-related file is not a case of file deletion, since file renaming can be documented by git (e.g., `--- a.c; +++ b.c` in the commit header records that the file renames from `a.c` to `b.c`).

We iterate through all commits (from commit $c_1$ to $c_n$) that modify the patch-related functions and terminate the process if $patch_i$ contains no statements (e.g., the $patch_3$ is set to $\phi$ in commit *0a5fae*, Figure 1) or is removed. In this study, we denote the commit that applies *direct changes* to $patch_i$ as ***patch-evolved commit*** and the corresponding CVE as ***patch-evolved CVE***. During the iterations, we only update $patch_i$ from *direct changes*. This ensures that the code snippet modified by direct changes is indeed derived from the original patch. Instead, we do not update indirect changes to $patch_i$ since the tracked statements can induce patch irrelevant ones after several iterations of tracking. Otherwise, too many irrelevant statements might be included in the patch, thus preventing us from understanding the evolution of patches precisely.

## 3.4 Research Question

This study aims to answer the following research questions.

***RQ1. Evolution Frequency:*** *How often do security patches evolve?* In this RQ, we intend to investigate the frequency of patch evolution, including the distribution for the four modifications as well as the time interval during patch evolution.

***RQ2. Evolution Characteristics:*** *What are the characteristics of security patch evolution?* In this RQ, we intend to investigate the changes of the patching code authorship as well as the change patterns during patch evolution.

***RQ3. Evolution Impact:*** *What are the impacts of security patch evolution?* In this RQ, we intend to investigate how patch evolution affects the existing 1-day vulnerability detection tools, including function-level vulnerability detection and patch presence test tools.

## 4 RQ1: EVOLUTION FREQUENCY

In this section, we delve into the analysis of patch evolution frequency from various perspectives. Particularly, we examine the distribution of four types of changes (direct changes, indirect changes, irrelevant, and removal) occurring during the patch evolution process, along with the time intervals between patch evolutions.

## 4.1 Overall Patch Evolution Statistics.

Investigating the frequency of patch evolution can provide a holistic view of this phenomenon. In this section, we first examine the occurrence rates of four types of modifications during patch evolution and provide a detailed analysis of their distributions.

The statistical results are presented in Table 2, the summary row provides an overall representation of the patch evolution across all projects. Overall, 81.1% (848/1046) of CVEs have undergone patch evolution (i.e., the patch of the CVEs has been modified by

**Table 2: Patch Evolution Statistics. #sec/#evo/#total denotes the number of patch-evolved commits addressing security issues CVEs v.s. patch-evolved CVEs v.s. total CVE numbers. The direct, indirect, irrelevant and removed respectively denote the average number of times corresponding changes occur for each evolved CVE.**

| Project | #sec/#evo/#total | direct | indirect | irrelevant | removed |
|---|---|---|---|---|---|
| FFmpeg | 29/128/154 | 1.2 ± 0.9 | 0.8 ± 1.7 | 6.8 ± 17.5 | 0.1 ± 0.3 |
| ImageMag. | 5/32/34 | 1.9 ± 1.6 | 0.8 ± 0.9 | 20.0 ± 25.5 | 0.1 ± 0.3 |
| libxml2 | 4/18/19 | 1.3 ± 0.6 | 0.5 ± 0.6 | 5.5 ± 6.8 | 0.1 ± 0.2 |
| linux | 287/434/555 | 1.3 ± 1.1 | 0.4 ± 0.9 | 7.5 ± 19.6 | 0.2 ± 0.4 |
| openjpeg | 1/10/10 | 1.2 ± 0.4 | 0.8 ± 0.6 | 2.1 ± 1.6 | 0.0 ± 0.0 |
| openssl | 7/29/32 | 2.1 ± 1.6 | 1.3 ± 1.7 | 12.9 ± 28.1 | 0.5 ± 0.6 |
| php-src | 28/47/64 | 1.6 ± 1.7 | 0.7 ± 1.2 | 8.3 ± 18.8 | 0.3 ± 0.5 |
| qemu | 25/53/70 | 0.9 ± 0.5 | 0.3 ± 0.5 | 6.9 ± 20.1 | 0.2 ± 0.5 |
| radare2 | 13/19/29 | 1.0 ± 1.1 | 0.5 ± 0.9 | 7.9 ± 14.9 | 0.3 ± 0.5 |
| tcpdump | 5/78/79 | 3.8 ± 3.0 | 1.7 ± 2.0 | 10.7 ± 8.8 | 0.1 ± 0.2 |
| **Summary** | **404/848/1046** | 1.8 ± 1.6 | 0.7 ± 1.3 | 10.1 ± 19.0 | 0.2 ± 0.4 |

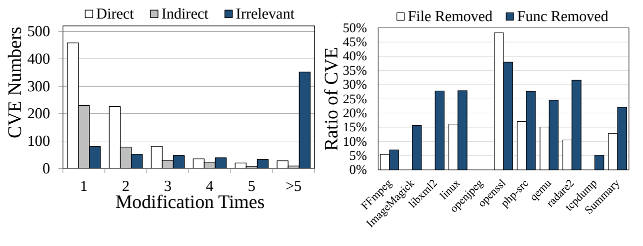
at least one direct change), which indicates that the evolution of patches is a common phenomenon. For #sec, we traversed the patch-evolved commits for each CVE and examined whether the commit messages mentioned any security issues. First, we identify those commits that involved keywords such as 'Fix', 'Bug', and 'CVE'. Subsequently, we manually verified that these commits indeed addressed security concerns. Among them, 38.62% (404/1046) of the CVEs have subsequent commits that address security issues. This means that 47.64% (404/848) of the evolving CVEs involve security-related commits. Though 'the patch-evolved commits are security-related' does not necessarily imply that subsequent developers introduce new security vulnerability by disrupting the original patch's fix logic, it would be helpful to empirically understand how often this occurs.

Then, we investigate the average frequency for the four modification types on patches among patch-evolved CVEs. On average, there are 1.8 direct changes and 0.7 indirect changes per patch-evolved CVE, with a standard deviation of 1.6 and 1.3 respectively. This suggests that direct and indirect modifications to patched statements are common during patch evolution. Irrelevant changes are most frequent with an average of 10.1, indicating that there are also a large number of modifications to the patch-related functions while do not impact the patch. Finally, the removal of the function or file containing the patch is the minority, occurring 0.2 times on average. Examining individual projects, *tcpdump* stands out with a notably high average of direct changes (3.8) compared to other projects. This possibly suggests a higher degree of refinement in the patches. *openssl* also exhibits a high average of indirect changes (1.3), implying a more complex data and control flow modification during patch evolution. Lastly, the openjpeg project has no instances of patch removal, indicating a stable patch inclusion in the project.

*Finding 1: Patch evolution is a prevalent phenomenon, occurring in over 81.1% of CVEs. Among the modifications during patch evolution, irrelevant changes are the most common, with an average of 10.1 irrelevant changes per patch-evolved CVE.*

## 4.2 Patch Modification Distribution

We then investigate the frequency distribution of the various modifications on all patch-evolved CVEs. The purpose of this experiment is twofold. First, we aim to discern patterns and trends in these change types as the number of modification increases. Second, we hope to leverage these insights to better comprehend the dynamics of patch evolution, thus enhancing our ability to manage the lifecycle of patch effectively.



**(a) Direct/Indirect/Irrelevant** **(b) Removal of Files/Functions**

**Figure 3: Patch Modification Distribution**

Figure 3a provides the distribution of three types of modifications: direct, indirect, and irrelevant during patch evolution across different numbers of occurrences. There are 458 CVEs with only one direct change. The frequency of direct changes decreases with the increase of occurrences, suggesting that most patches tend to experience a single direct change. Similarly, indirect changes are most frequent in a single occurrence with 230 CVEs. The frequency of indirect changes decreases with the increase in occurrences, implying that indirect changes, like direct changes, usually happen once during patch evolution. The frequency of irrelevant changes increases markedly for patches with more than five occurrences (352 CVEs). This suggests the complexity of software evolution and the potential for a wide range of changes to occur during the lifecycle of a patch, not all of which directly alter the patch itself.

Figure 3b provides the data on file and function removal in patch-evolved CVEs across various projects. Overall, 12.9% of the files and 22.1% of the functions related to CVE patches are removed. Examining individual projects, openssl shows the highest file removal ratio on file (48.3%) and function (37.9%), indicating significant codebase changes during patch evolution.

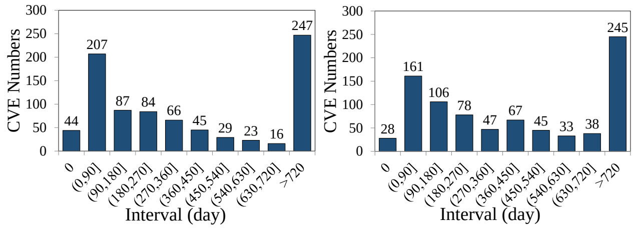
*Finding 2: Most patches typically go through just one direct or indirect alteration, with irrelevant changes cropping up more than five times. Moreover, 12.9% of patch-related files and 22.1% of functions get removed, suggesting significant codebase evolution.*

## 4.3 Patch Evolution Timeline

In this section, we investigate the timeline for patch evolution. This understanding is critical because it sheds light on the lifecycle of security patches, and the process of patch management practices.

To accomplish this, we utilized a method of tracking the patch evolution over specific intervals. The data was classified into two categories: (1) the time interval for the initial patch evolution (i.e., the interval for the first patch-evolved commit), (2) the average evolution time (i.e., the average time between the original patch and the last patch-evolved commit). The first category provides us insights into how quickly patches are modified after their initial release.

The second category provides a more holistic view of the patch evolution process, taking into account all patch-evolved commits from the first to the last.



**(a) Initial Evolution Timing** **(b) Average Evolution Duration**

**Figure 4: Distribution of Patch Evolution Timescales**

Figure 4a details the distribution of the time intervals for the first evolution of CVE patches. As an overarching trend, there is a distinct decrease in the CVE numbers as the interval expands. A total of 44 CVE patches were altered on the day of their release. This could be caused by the developers recognizing that the original patch was insufficient to fix the vulnerability, leading them to further refine the patch's logic. The (0,90] day range boasts the highest concentration with 207 CVEs, hence 29.6% (44+207/848) patches are altered within 90 days. This implies that a large portion of these patches undergo their first evolution relatively quickly after their introduction. However, in practice, downstream software often backports patches based on the original versions, as provided in the patch URL links from NVD. If the fix logic of the original patch is flawed, it could compromise the security of the downstream software. It is noteworthy to highlight the anomaly in the *>720* day interval, whichcontains 247 CVEs. This interval, representing a much longer timescale, is the second most populated interval. This could point to a subset of patches that are inherently stable, requiring their first evolution only after an extended period of time.

Figure 4b portrays the average evolution time of the CVE patches. Similar to Figure 4a, this figure also exhibits a general decline in CVE numbers as the interval lengthens. The most active range here is also the (0,90] day range with 161 CVE patches. This infers that most patches evolve frequently within less than 90 days.

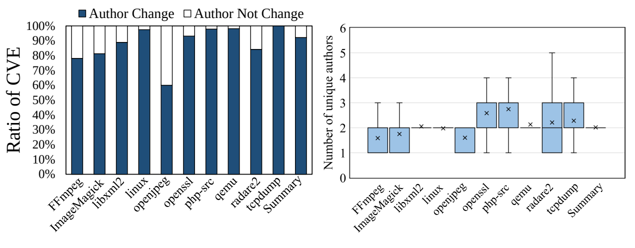
*Finding 3: Quite a few CVE patches (29.6%) undergo the first evolution quickly, particularly within the first 90 days. However, a sizable group only evolves for the first time after exceeding 720 days. The average evolution time mostly aligns with this trend.*

## 5 RQ2: EVOLUTION CHARACTERISTICS

### 5.1 Authorship Changes in Patch Evolution

Recent studies [17, 20, 28] underscore the significant influence of "human" elements, including aspects such as ownership, experience, organizational structure on software quality. Prior research [3] also suggests that a larger number of developers contributing to a single file could result in a higher possibility to induce defects. Such observation motivates us to examine the changes in the patched code's authorship during the evolution. For instance, after the release of a security patch, subsequent contributors to the patch-related functions may not realize that parts of the code actually fixed a security vulnerability. Therefore, they may inadvertently alter the data and control flows associated with the patch, or even remove the original patched code. This disruption to the patch's fix logic could potentially introduce potential security threats.

To delve deeper into how authorship changes during patch evolution, we selected patch-evolved CVEs and conducted a two-dimensional analysis: (1) We examined whether the author of a patch-evolved commit is the same as the original patch author. If they are not the same, we considered this as a change in patch authorship. For example, the author of commit `9f1e5e` in Figure 1 has been changed compared to the author of the original patch. Therefore, we calculated the proportion of CVEs in each project that experienced a change in patch authorship. (2) Further, we counted the unique number of authors for each patch-evolved CVE to understand how many unique developers have contributed to a single patch (including the author of original patch and patch-evolved commits). For example, among the four commits in Figure 1, there are two unique developers.



**(a) Authorship Changes Ratio** **(b) Unique Authorship Distribution**

**Figure 5: Authorship Changes during Patch Evolution**

**Table 3: The code features and the corresponding (sub-)tokens**

| Features | (sub-) tokens | Features | (sub-) tokens |
|---|---|---|---|
| condition | if, switch | bitwise | &, \|, ˆ et al. |
| loop | for, while | memory API | alloc, free, mem et al. |
| jump | return, goto et al. | string API | string, str |
| arithmetic | +, -, *, / et al. | lock API | lock, mutex, spin |
| relational | <, >, == et al. | system API | init, register, map et al. |
| logical | &&, \|\|, ! et al. | other func | other function calls |

Figure 5 shows the statistical result of how frequently security patches' authorship changes. The results show that there is a significantly high proportion of patches that undergo author changes during their evolution, in which 93.2% (=790/848) CVEs have undergone the changes in patch authorship. This suggests that a large number of patches are typically worked on by multiple developers. On the other hand, projects like *FFmpeg* (78.1%=100/128) and *openjpeg* (60.0%=6/10) have a relatively low proportion of author changes. This could suggest a higher degree of exclusive ownership over the patches in these projects. The boxplot as shown in Figure 5b shows the distribution of unique authors during the patch evolution across projects. An initial glance at the summary data which encapsulates the overall scenario reveals a central tendency around 2. This implies that on average, two authors are predominantly involved to contribute a patch across all projects.

*Finding 4: Most of the CVE patches (i.e., 93.2%) are modified by other developers after its initial release. On average, two authors predominantly contribute to a single patch across all projects.*

## 5.2 Code Feature Changes in Patch Evolution

To gain a deeper understanding of how patches evolve, we conducted a quantitative analysis of the changes in code features during the patch evolution process. In this section, our focus is on direct and indirect changes of the patch as they effectively reflect the transformation process of both the patch itself and its context.

Recent researches suggest that source code vulnerabilities are highly correlated with certain syntax characteristics [18, 42]. For example, the usage of pointers and arrays in the C/C++ language is likely to be more vulnerable since these operations often lead to *out-of-bounds* (OOB) access or *null pointer dereference* [19]. Thus, we extract the code features at the level of Abstract Syntax Tree (AST) following an existing patch identification work, named GraphSPD [40]. GraphSPD was originally proposed to identify whether a given commit is a security patch and propose several patch-related features that are highly related. We adopt the proposed code features from GraphSPD to quantify how patches evolve. Specifically, we begin by segmenting C/C++ code snippets into code tokens using the *clang* tool [38]. Then, the AST features and the corresponding tokens and sub-tokens are matched based on Table 3. For instance, in commit `0a5fae`, Figure 1, Lines 202-203 contribute one time on *condition*, *relational*, *arithmetic*, and *jump* for indirect changes, respectively; Line 205 contributes one and two times on the *condition* and *other_func* for direct changes.

Table 4 shows the statistical results on our dataset, which provides a detailed comparison of the changes in various code features during the patch evolution across projects. In particular, each cell in the table is denoted as `a/b`, where `a` denotes the number of direct changes and `b` denotes the number of indirect changes. The summary row displays the total count of the two changes. The *relational* feature contains the highest total changes with 1,126 direct changes and 959 indirect changes. This indicates that *relational* is the most frequently modified AST feature during patch evolution across all projects. The *condition* feature also has a high number of modifications, with 787 direct changes and 828 indirect changes. We further analyzed the underlying causes. *condition* and *relational* modifications are often linked to control flow changes. Since software patches frequently involve bug fixes or functionality improvements, they often require changes to existing conditions or the addition of a new sanitizer checker to alter the control flow. These changes are common when fixing bugs or refining fix logic as they often pertain to adjusting how different variables or states interact with each other. In contrast, AST features like *bitwise* (115/121) and *string_api* (74/62) have significantly fewer changes. This suggests that these features are less likely to be modified during patch evolution, indicating their relative stability.

In terms of each individual project, *linux* patches contain relatively more api changes, such as *lock_api* and *system_api*, probably due to the importance of these areas in ensuring that the system operates correctly, efficiently and safely.

*Finding 5: `Condition` and `relational` features have been modified the most during patch evolution, mainly to refine the original constraints of the patch. Some projects (e.g., linux) also change `lock_api` and `system_api` to enhance the correctness of system.*

## 5.3 Patch Evolution Patterns

During the patch evolution, analyzing the changes in code features allows us to quantitatively investigate how developers modify patches. To further understand the patterns of patch evolution qualitatively, we randomly selected 100 patch-evolved CVEs and their corresponding 234 patches and analyzed them manually. We noticed developers commonly employ two types of code modifications: (1) Type 1: *Adjust code without changing the fix logic.* During software evolution, developers often refactor code to enhance quality, improve readability, or boost performance. Such changes usually do not affect the fix logic of the patch. Take CVE-2016-2179 as an example, in which a patch-evolved commit (commit=81926c) changed the callee prototype from " void dtls1_clear_received_buffer(SSL *s)" to "void dtls1_clear_received_buffer(SSL_CONNECTION *s)". Such change in type was to refactor code to enhance quality, without affecting the repair logic (e.g., sanity checking the correctness of the variable). Finally, we classified this case as "Modify parameters of callees" under Type 1. (2) Type 2: *Adjust or refine the patch's fix logic.* After a patch is released, its fix logic might be further refined by the developers. For instance, if the patch initially introduced a sanitizer check to ensure that program state aligns with expectations, developers might later modify the conditions of this sanitizer check, thereby refining the patch's logic. Three individuals participated in the classification process. Specifically, to ensure the scientific rigor, two individuals, each with over four years of programming experience independently identified evolution patterns. Whenever there are discrepancies (approximately 5% of instances), a third individual joined the discussion until a consensus is reached, thus ensuring the reliability of the final classifications.

Table 5 presents the detailed results of the patch evolution patterns. For 54.7% of patch evolution cases, developers only refactor the code without changing the patch's fix logic. However, for 45.3% of cases, developers might alter the original patch's fix logic by changing the patch itself or its context. Among these cases, 13.68% involve changes to the patch context. In 14.53% of cases, conditions within the patch are modified. Additionally, 14.53% of cases involve the deletion of all or part of the patch code. Such changes should be carefully handled, otherwise, new security issues can be easily induced (see our discussion in Section 7).

*Finding 6: Substantial code changes are made to refine or adjust the patch's fix logic (45.3% of cases), which should be carefully handled since they might break the fix logic and be further exploited.*

## 6 RQ3: EVOLUTION IMPACT

In this section, we investigate how patch evolution affects existing 1-day vulnerability detection tools, especially on the accuracy of vulnerability detection. As aforementioned, we focus on two categories of detection tools: function-level tools and patch presence test tools. Following the existing study [53], we adopt the assumption that *most of the evolved patches can still preserve the fixing semantics* to understand the evolution impact for most of the cases. Under such an assumption, vulnerability detection tools should recognize the evolved versions as secure or patched. Nevertheless, it does not imply that the evolved patches will not introduce new

**Table 4: The change of code features during patch evolution. The cell consists of a/b , where a denotes the number of changes to the code feature in direct changes throughout patch evolution, and b denotes such a number in the indirect changes.**

| Projects | condition | loop | jump | arithmetic | relational | bitwise | logical | other_func | memory_api | string_api | lock_api | system_api |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FFmpeg | 76/107 | 7/16 | 34/0 | 70/73 | 164/213 | 10/18 | 58/81 | 33/44 | 3/11 | 2/13 | 0/0 | 13/45 |
| ImageMagick | 18/10 | 0/1 | 2/0 | 6/4 | 13/13 | 1/6 | 1/0 | 18/9 | 2/0 | 2/3 | 0/0 | 0/0 |
| libxml2 | 10/11 | 0/0 | 6/0 | 8/7 | 8/11 | 1/0 | 4/2 | 10/0 | 0/0 | 0/0 | 0/0 | 1/0 |
| linux | 305/259 | 12/8 | 205/2 | 92/65 | 281/171 | 73/39 | 113/125 | 465/217 | 79/12 | 4/3 | 132/16 | 153/51 |
| openjpeg | 1/14 | 0/2 | 1/0 | 12/51 | 20/23 | 12/12 | 10/7 | 2/2 | 3/0 | 0/0 | 0/0 | 2/0 |
| openssl | 30/141 | 0/1 | 9/0 | 0/3 | 58/210 | 0/7 | 6/29 | 11/40 | 22/10 | 0/0 | 0/0 | 9/18 |
| php-src | 260/129 | 10/3 | 154/24 | 75/36 | 447/156 | 5/3 | 45/38 | 101/99 | 13/3 | 25/30 | 0/0 | 10/13 |
| qemu | 47/40 | 4/1 | 24/0 | 19/13 | 73/33 | 5/13 | 14/11 | 58/20 | 8/6 | 4/6 | 2/0 | 14/6 |
| radare2 | 12/22 | 3/4 | 9/0 | 10/9 | 29/17 | 0/4 | 8/3 | 7/8 | 6/1 | 5/7 | 0/0 | 3/0 |
| tcpdump | 28/95 | 2/15 | 46/0 | 146/179 | 33/112 | 8/19 | 5/17 | 177/200 | 1/0 | 32/0 | 0/0 | 15/0 |
| **Summary** | **787/828** | **38/51** | **490/26** | **438/440** | **1,126/959** | **115/121** | **264/313** | **882/639** | **137/43** | **74/62** | **134/16** | **220/133** |

**Table 5: Patch evolution patterns**

| Types | P | Patterns | Description | Ratio |
|---|---|---|---|---|
| Type1 (54.70%) | P1 | Rename identifiers | Change the name of a variable, function, or other element inside the patch. | 14.53% |
| | P2 | Add/remove cast expression | Add or remove the cast expression for variables. | 4.27% |
| | P3 | Rewrite loop patterns | Convert a for loop statement to a while loop statement, or vice versa. | 1.71% |
| | P4 | Adjust code format | Modify the formatting of code, such as adjusting indentation or Line-Break. | 5.98% |
| | P5 | Extract and reuse code | Duplicated code is eliminated by extracting it into a distinct module for reuse. | 4.27% |
| | P6 | Define integer macros | Replace values used multiple times in code with a macro. | 1.71% |
| | P7 | Modify parameters of callees | Change types or number of parameters for callees inside the patch. | 19.66% |
| | P8 | Array indexing to pointer dereferencing | Change from array indexing to pointer access. E.g., change from p[index] to *(p+index). | 2.56% |
| Type2 (45.30%) | P9 | Adjust memory allocation | Modify the allocated memory size in the memory allocation function. | 1.71% |
| | P10 | Change patch context | Add or remove statements that have control or data flow relationship with the patch. | 13.68% |
| | P11 | Modify conditions inside patch | Change the condition expressions inside the patch to refine its logic | 14.53% |
| | P12 | Changes loop termination condition | Modify the loop termination condition to alter the number of iterations. | 0.85% |
| | P13 | Delete all/part of patch code | Part or all of the patch-related code is removed. | 14.53% |

issues. We further performed a case study to show that the fixing semantics of evolved patch can be changed and induce new vulnerabilities in Section 7.

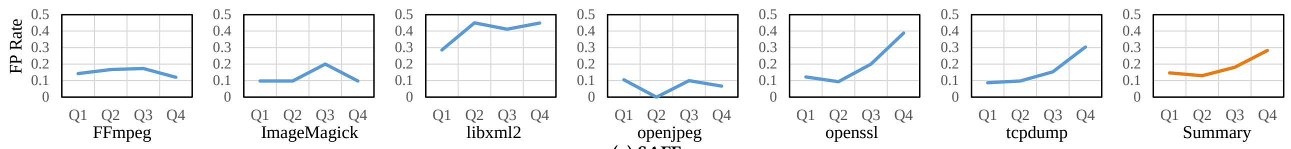
For function-level tools, we selected SAFE [23] and jTrans [39]. Existing study [22] has proven that SAFE performs well in function similarity tasks, and can withstand various code disturbances in real-world software (such as different levels of compiler optimization and system architectures). jTrans is a recently proposed tool to compare function similarity, the performance of which is shown to surpass that of other similar tools [39]. For patch presence test tools, we select PMatch [15] and BinXray [47]. PMatch is a learning-based tool that detects the patch status in the target binary using unsupervised embedding. BinXray is another advanced patch presence test tool. Specifically, it generates a patch signature by diffing the pre-patch and post-patch functions and then matches the target function with the signature to determine the patch status.

**Data preprocessing.** We selected six projects from the Dataset in Table 1 (*FFmpeg*, *ImageMagick*, *libxml2*, *openjpeg*, *openssl*, and *tcpdump*) and utilized the patch-evolved CVEs in these projects, containing 301 CVEs, in this experiment. We exclude the other projects since this experiment requires compiling multiple versions of the project, which is very time-consuming especially for those large-scale ones such as *linux*. Besides, our study aims to evaluate whether the performance of vulnerability detection tools is affected by patch evolution, and 301 CVEs are sufficient to observe such an impact. In particular, the above four tools require the binary of the pre-/post-patch versions as references. The pre-patch version refers to the latest version just before the patch is applied, while the post-patch version refers to the version with the original patch applied. Therefore, we need to compile multiple binary versions for each CVE, including two reference versions (pre-/post- patch versions) and several target versions Specifically. For each of the 301 CVEs, we first compiled the pre-/post-patch versions. Then, if the number of patch-evolved commits for a CVE is four or fewer, we compiled all the evolved versions. Otherwise, we select 4 versions by quartiles chronologically to save efforts and ensure consistency across different projects. We refer to them as *Q1*, *Q2*, *Q3* and *Q4* respectively. For example, if we have patch-evolved commits ordered from 1 to 7, we would choose versions 1 (Q1), 3 (Q2), 5 (Q3), and 7 (Q4). In total, we compiled 1,521 versions of binary for these 301 CVEs. During the compilation process, we used the command *git checkout commit* to switch to the corresponding commit and compiled the software using *gcc* with the *-O2* optimization level following existing study [47]. The compilation task took us approximately 120 hours in total. Then, we conducted the following experiments using pre-/post-patch version as references and the Q1-Q4 as target versions.

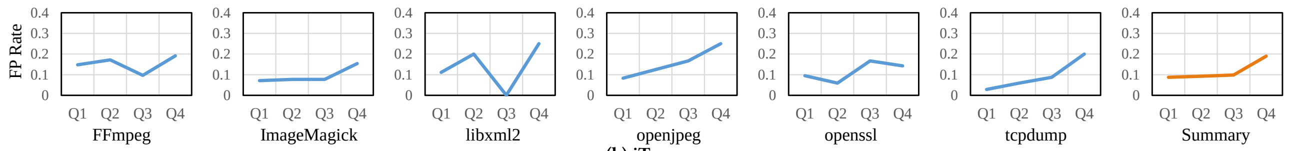
## 6.1 Function-Level Tools

Function similarity tools are widely used in 1-day vulnerability detection tasks. If a function in the target software matches the pre-patch function, it can be inferred that the target software is affected by the vulnerability. However, existing researches [15, 24, 47] suggest that the high similarity between vulnerable and patched versions leads to a large number of false positives (FPs). This is because the patches often introduce only small code changes, making it difficult for function similarity tools to distinguish patched functions from vulnerable ones.

To evaluate how the function similarity tools are affected by patch evolution, we adopt the strategy from an existing study [7].

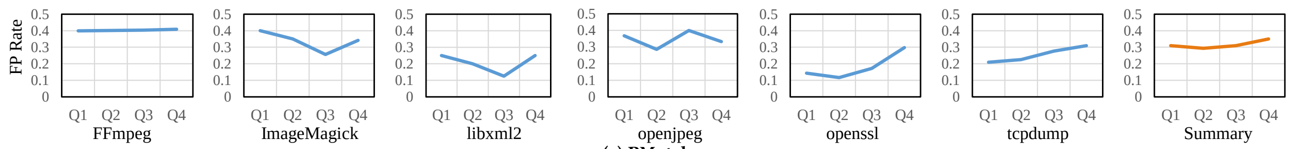


**(a) SAFE**

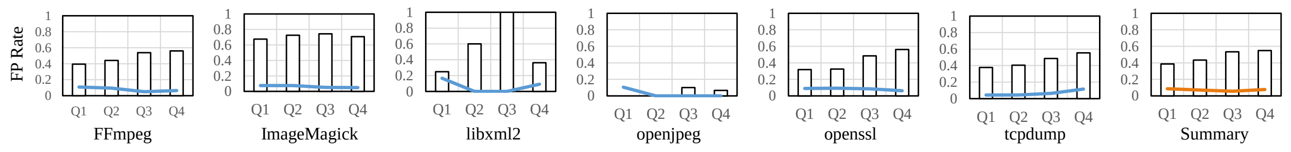


**(b) jTrans**

**Figure 6: The impact of patch evolution on the function-level 1-day vulnerability detection tools jTrans and SAFE**



**(a) PMatch**



**(b) BinXray. rectangle denotes the ratio of BinXray that cannot generate results due to significant changes.**

**Figure 7: The impact of patch evolution on the patch presence test tools PMatch and BinXray**

Specifically, we determined the safety of the target software based on which reference version it is more similar to (i.e., if a testing target is more similar to the pre-patch version than the post-patch one, it is considered unpatched; otherwise, it is deemed patched). Since patches have been evolved to four versions (i.e., Q1-Q4), if a tool reports that the target binary (one of Q1-Q4) has no patch applied, we consider there is a false positive.

In this way, we evaluate SAFE and jTrans by observing the false positive rate (FPR) changes as the patch evolves. Figure 6a shows the results for SAFE. The summary figure demonstrates an increasing trend along the evolution of patches, starting at 0.15 and ending at 0.28. This suggests that as patches evolve, SAFE tends to produce more FPs. Individual project results exhibit varying trends. For instance, the FPR of FFmpeg increases initially but then decreases. Libxml2 consistently observes the highest FPR, suggesting that SAFE is ineffective to handle patches in this project.

Figure 6b, representing jTrans, shows a slightly different trend. The summary FPR remains relatively stable across Q1-Q3 (i.e., 0.09, 0.09, 0.10) but then sees a notable increase in the Q4 (0.19). This reflects that jTrans performs better in resisting code evolution compared with SAFE. At the project level, openjpeg and libxml2 show considerable increases in the last version, while ImageMagick maintains relatively stable FPR across all versions.

In conclusion, both SAFE and jTrans demonstrate an increased tendency to generate false positives as patches evolve, more pronounced in the last version. This could potentially indicate the limitations of such tools in resisting to patch evolution.

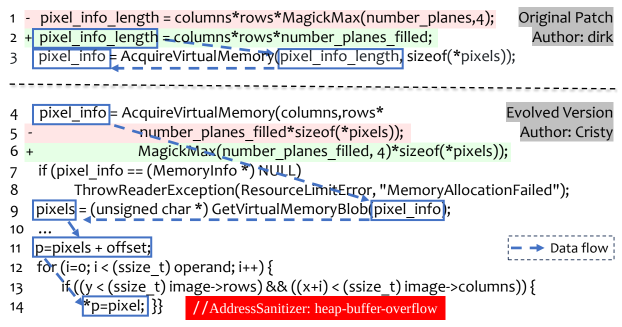
*Finding 7: As patches evolve, the false positive rates of function-level tools SAFE and jTrans increase notably, rising from 0.15 to 0.28 and 0.09 to 0.19 respectively, suggesting limitations for function-level tools in handling evolved patches.*

## 6.2 Patch Presence Test Tools

In this section, we investigate how patch evolution impacts existing patch presence test tools. Patch presence test tools are proposed to enhance the precision of the above tools at the function level by directly matching the patch signature in the target binary. As a result, these tools are particularly suited for patch detection tasks. Then, we evaluate how patch evolution affects patch presence tools. The methodologies and metrics utilized are consistent with those we used for the evaluation of function-level tools.

Figure 7a shows the statistical results for PMatch. As we can see from the summary, the average FPR for PMatch increases from 0.29 to 0.35 as the patch evolves. This indicates a rising inaccuracy over the progression of patch evolutions. This trend is also consistent with function level tools, as shown in Figure 6. Looking at individual projects, FFmpeg shows a marginal increase in false positives across various versions, while openssl and tcpdump display more substantial increases in the final version. Libxml2 and ImageMagick, on the other hand, show a significant decrease in Q1-Q3 followed by an increase in Q4. This could be a limitation in the design of PMatch, which might not account for the complexities and changes that occur in the later patch evolutions.

Figure 7b shows the results for BinXray and the rectangles indicate the ratio of the cases that BinXray cannot generate results. If BinXray finds that the target binary differs too much from the



```
-  pixel_info_length = columns*rows*MagickMax(number_planes,4);
+  pixel_info_length = columns*rows*number_planes_filled;
   pixel_info = AcquireVirtualMemory(pixel_info_length, sizeof(*pixels));
-------------------------------------------------------------------------
   pixel_info = AcquireVirtualMemory(columns,rows*
-                   number_planes_filled*sizeof(*pixels));
+                   MagickMax(number_planes_filled, 4)*sizeof(*pixels));
   if (pixel_info == (MemoryInfo *) NULL)
        ThrowReaderException(ResourceLimitError, "MemoryAllocationFailed");
   pixels = (unsigned char *) GetVirtualMemoryBlob(pixel_info);
   ...
   p=pixels + offset;
   for (i=0; i < (ssize_t) operand; i++) {
       if ((y < (ssize_t) image->rows) && ((x+i) < (ssize_t) image->columns)) {
           *p=pixel; }}   //AddressSanitizer: heap-buffer-overflow
```

**Figure 8: An example of exploiting the evolved patch of CVE-2016-7515 from project ImageMagick**

version of pre-patch and post-patch, it will report *too much diff* directly. We refer to such cases as *cannot generate results.* As we can see from the summary figure, the FPR remains consistently low (<10%), demonstrating BinXray's high precision. However, the increasing *cannot generate results* rate reveals that BinXray's recall is compromised as the patch evolves. This is because BinXray only considers the differences between the pre-patch and post-patch versions to generate patch signatures and detect patch in the target binary. However, software evolution can significantly alter the code structure related to patches, thus making it difficult for BinXray in matching patch semantics. As for individual projects, BinXray shows the lowest FPR on *openjpeg* and *libxml2*, showcasing its precision. However, *libxml2* also exhibits a proportion of 100% for *cannot generate results* in Q3, indicating its limitations in patch recognition as patches evolve.

*Finding 8: The performance of patch presence test tools degrade with patch evolution. Tools like BinXray show an uptick for "cannot generate result" cases, peaking at 55.0%. This highlights the limitations of patch presence test tools in resisting code evolution.*

## 7 DISCUSSION

### 7.1 A Case Study of Exploiting Evolved Patches

In RQ3, we follow the existing study's assumption that *most of the evolved patches can still preserve the fixing semantics.* However, as we mentioned in RQ1, during software evolution, the developers might unintentionally modify the code that is related to a CVE patch, thus disrupting the patch's fix logic. To validate this hypothesis, we further perform a case study on the evolved patch for CVE-2016-7515 (from *ImageMagick*). The original patch (commit=`2ad6d3`) is shown in Figure 8 (see Appendix). The patch implements the fix logic by modifying the variable *pixel_info_length* to align the variable *pixel_info* with its expected value. However, the variable *pixel_info* is modified by another developer during software evolution. Then, we explore whether the evolved patch can be exploited using an existing directed fuzzer AFLGo [4]. In the evolved version (commit=`13db82`), we used the original testcase of CVE-2016-7515 as starting seed and set the patch-related code as the target lines. Finally, we detected a *heap buffer overflow* as shown in Figure 8. After a detailed examination, we found that it is the other developer's commit that modified the variable *pixel_info*, ultimately triggers the error. Finally, we manually analyzed the update records of the vulnerability-relevant dataflow and confirmed that the new vulnerability was fixed in commit `e50f19f`.

The above example demonstrates that during the software evolution, developers might alter the original patch's fix logic, thus potentially induce new security threats. The occurrence of such phenomena underscores the importance of studying patch evolution. We believe that this study can shed important light on exploiting evolved patches, and we left it as our important future work.

### 7.2 Actionable Suggestions

Our study demonstrates that patch evolution indeed poses security risks and can impact the effectiveness of existing vulnerability detection tools. The results and findings highlight the need to address potential security issues during patch evolution. We provide actionable suggestions from the following aspects for further exploration.

(1) *Patch Evolution Sanitizer.* In large open-source projects, various developers often collaborate. For instance, different authors may modify code introduced from a previous security patch (i.e., in Finding-4, 93.2% of patches are modified by other developers). Moreover, in Finding-6, we found that 45.3% of code changes are made to refine or adjust the patch's fix logic. This study can inspire a new tool to detect whether such modifications indeed introduce security risks (e.g., compromising the repair logic). Specifically, we can leverage static analysis to infer whether certain patch evolutions might belong to Type 2 (i.e., as found in Finding-6, 45.3% of patches refine or adjust the patch's fix logic). If so, directed fuzzing techniques (e.g., AFLGo [5]) can then be applied to test the modified context related to the patch.

(2) *Enhancing Patch Detection Tools.* Our study found that the false positive rates of existing tools increase during patch evolution. This is because these tools generate patch signatures based solely on the original security patch. Our findings can guide the design of future tools to enhance their signature generation capabilities. In particular, Finding 5 summarizes the patterns of patch evolution and reveals that `condition` and `relational` features have been modified the most during patch evolution. Such information can be used to enhance patch signatures for better patch detection.

(3) *Facilitating Patch Backporting*: Patch backporting must consider the consistency of patched code and context across different versions. However, downstream software developers often backport patches based solely on the original patches, making it challenging to timely refine the patch logic. Our work on tracking patch evolution assesses whether the original patch has undergone logical changes. When patch evolution occurs, we can utilize program analysis to infer whether certain evolutions have refined the patch logic. If they do, the refined patch logic should also be considered when backporting to downstream software.

### 7.3 Limitations

The inability to accurately model software refactoring and patch evolution is a major limitation in our study. For instance, during software evolution, the fix logic of a security patch can shift from its original function to a new location. This can be mistakenly interpreted as the patch being deleted. Nevertheless, we argue that using static analysis to model the fix logic of security patches is inherently a challenging task, given the diverse manners in which software evolves. In this paper, we track patch evolution by analyzing the modified lines within the patch. This ensures that the evolved patches are derived from the original ones. We also analyze changes in the control and data flows related to the patch during its evolution, providing a measure of the alterations in patch fix logic to some degree.

## 8 RELATED WORK

We discussed 1-day vulnerability detection tools in Section 2. we introduce another relevant study as follows.

**Patch Lifecycle Analysis.** Numerous studies have explored the lifecycle of security patches. Notably, Tan et al. [35], Li et al. [16], Shahzad et al. [32], and Frei et al. [10] have conducted extensive studies on the vulnerability lifecycle and patching timelines, utilizing publicly accessible data from open-source repositories. In addition, studies by Zheng et al. [53], Jiang et al. [12], and Dai et al. [7] have examined patch propagation from the Android Open Source Project (AOSP) to downstream vendors. These works typically concentrate on patch management within the relationship between upstream and downstream patch management.

However, none of the existing studies have discussed the patch evolution problem, thus narrowing the scope of patch lifecycle research. In fact, according to our research, 81.1% of security patches undergo evolution after their release. Among these, 29.6% are modified by developers within 90 days, indicating potential changes in the patch's repair logic. Our work is the first systematic study of the phenomenon of security patch evolution, shedding light on future research into patch related researches.

## 9 CONCLUSION

In this paper, we conduct the first empirical study on patch evolution based on 1,046 CVE vulnerabilities and 2,633 patches, uncovering several findings. We found 81.1% CVE patches evolve, involving direct and indirect modifications and 29.6% patches evolve within 90 days. Through analysis of code feature changes and manual pattern exploration, we understood the characteristics of patch evolution. Furthermore, we found that patch evolution negatively impacts the effectiveness of 1-day vulnerability detection tools significantly. Finally, this study also reveals the evolved patches might induce new security issues, and thus can be further exploited. We believe that our research can shed light on future researches related to patch analysis.

## REFERENCES

[1] Vibhor Agarwal and Nishanth Sastry. 2022. "Way back then": A Data-driven View of 25+ years of Web Evolution. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 3471–3479. https://doi.org/10.1145/3485447.3512283

[2] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar T. Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 97–106. https://doi.org/10.1145/1882291.1882308

[3] Wai Fong Boh, Sandra Slaughter, and J. Alberto Espinosa. 2007. Learning from Experience in Software Development: A Multilevel Analysis. *Manag. Sci.* 53, 8 (2007), 1315–1331. https://doi.org/10.1287/mnsc.1060.0687

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[6] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai

[7] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai

[8] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 472–489. https://doi.org/10.1109/SP.2019.00003

[9] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/automating-patching-of-vulnerable-open-source-software-versions-in-application-binaries/

[10] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. 2006. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense, LSAD '06, Pisa, Italy, September 11-15, 2006*. ACM, 131–138. https://doi.org/10.1145/1162666.1162671

[11] Qingze Hum, Wei Jin Tan, Shi Ying Tey, Latasha Lenus, Ivan Homoliak, Yun Lin, and Jun Sun. 2020. CoinWatch: A Clone-Based Approach For Detecting Vulnerabilities in Cryptocurrencies. In *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020*. IEEE, 17–25. https://doi.org/10.1109/Blockchain50366.2020.00011

[12] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[13] Yiqiao Jin, Yunsheng Bai, Yanqiao Zhu, Yizhou Sun, and Wei Wang. 2023. Code Recommendation for Open Source Software Developers. In *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*, Ying Ding, Jie Tang, Juan F. Sequeda, Lora Aroyo, Carlos Castillo, and Geert-Jan Houben (Eds.). ACM, 1324–1333. https://doi.org/10.1145/3543507.3583503

[14] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[15] Zhe Lang, Shouguo Yang, Yiran Cheng, Xiaoling Zhang, Zhiqiang Shi, and Limin Sun. 2021. PMatch: Semantic-based Patch Detection for Binary Programs. In *IEEE International Performance, Computing, and Communications Conference, IPCCC 2021, Austin, TX, USA, October 29-31, 2021*. IEEE, 1–10. https://doi.org/10.1109/IPCCC51483.2021.9679443

[16] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2201–2215. https://doi.org/10.1145/3133956.3134072

[17] Zhen Li, Qian (Guenevere) Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1906–1918. https://doi.org/10.1145/3510003.3510181

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[20] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1547–1559. https://doi.org/10.1145/3377811.3380923

[21] Tianyue Luo, Chen Ni, Qing Han, Mutian Yang, JingZheng Wu, and Yanjun Wu. 2015. POSTER: PatchGen: Towards Automated Patch Detection and Generation for 1-Day Vulnerabilities. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1656–1658. https://doi.org/10.1145/2810103.2810122

[22] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2099–2116. https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[23] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11543)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer, 309–329. https://doi.org/10.1007/978-3-030-22038-9_15

[24] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 180–190. https://doi.org/10.1109/ASE.2013.6693078

[25] NVD. 2024. https://nvd.nist.gov/. Accessed: 2024-04.

[26] phantomjs. 2024. https://github.com/ariya/phantomjs. Accessed: 2024-04.

[27] php src. 2024. https://github.com/php/php-src. Accessed: 2024-04.

[28] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, Experience and Defects: A Fine-Grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 491–500. https://doi.org/10.1145/1985793.1985860

[29] FFmpeg Repository. 2024. https://github.com/FFmpeg/FFmpeg. Accessed: 2024-04.

[30] Linux Repository. 2024. https://github.com/torvalds/linux. Accessed: 2024-04.

[31] OpenSSL Repository. 2024. https://github.com/openssl/openssl. Accessed: 2024-04.

[32] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 771–781. https://doi.org/10.1109/ICSE.2012.6227141

[33] Xiaonan Song, Aimin Yu, Haibo Yu, Shirun Liu, Xin Bai, Lijun Cai, and Dan Meng. 2020. Program Slice based Vulnerable Code Clone Detection. In *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020, Guangzhou, China, December 29, 2020 - January 1, 2021*, Guojun Wang, Ryan K. L. Ko, Md. Zakirul Alam Bhuiyan, and Yi Pan (Eds.). IEEE, 293–300. https://doi.org/10.1109/TrustCom50675.2020.00049

[34] The state of open source security. 2024. https://snyk.io/reports/open-source-security/. Accessed: 2024-04.

[35] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the Practice of Security Patch Management across Multiple Branches in OSS Projects. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 767–777. https://doi.org/10.1145/3485447.3512236

[36] Zhushou Tang, Minhui Xue, Guozhu Meng, Chengguo Ying, Yugeng Liu, Jianan He, Haojin Zhu, and Yang Liu. 2019. Securing android applications via edge assistant third-party library detection. *Comput. Secur.* 80 (2019), 257–272. https://doi.org/10.1016/j.cose.2018.07.024

[37] tcpdump. 2024. https://github.com/the-tcpdump-group/tcpdump. Accessed: 2024-04.

[38] Clang Tool. 2024. https://clang.llvm.org/get_started.html. Accessed: 2024-04.

[39] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: jump-aware transformer for binary code similarity detection. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 1–13. https://doi.org/10.1145/3533767.3534367

[40] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2409–2426. https://doi.org/10.1109/SP46215.2023.10179479

[41] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 149–160. https://doi.org/10.1109/DSN48987.2021.00030

[42] Xinda Wang, Shu Wang, Kun Sun, Archer L. Batcheller, and Sushil Jajodia. 2020. A Machine Learning Approach to Classify Security Patches into Vulnerability Types. In *8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020*. IEEE, 1–9. https://doi.org/10.1109/CNS48642.2020.9162237

[43] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 326–337. https://doi.org/10.1145/3338906.3338962

[44] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures.. In *USENIX Security Symposium*. 1165–1182.

[45] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 347–359. https://doi.org/10.1145/3597926.3598061

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 363–376. https://doi.org/10.1145/3133956.3134018

[47] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 376–387. https://doi.org/10.1145/3395363.3397361

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 590–604. https://doi.org/10.1109/SP.2014.44

[49] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 867–879. https://doi.org/10.1145/3597926.3598102

[50] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing OSS Patch Backporting with Semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2366–2380. https://doi.org/10.1145/3576915.3623188

[51] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHUNTER: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150

[52] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 887–902. https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang

[53] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3649–3666. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng

[54] Deqing Zou, Hanchao Qi, Zhen Li, Song Wu, Hai Jin, Guozhong Sun, Sujuan Wang, and Yuyi Zhong. 2017. SCVD: A New Semantics-Based Approach

for Cloned Vulnerable Code Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10327)*, Michalis Polychronakis and Michael Meier (Eds.). Springer, 325–344. https://doi.org/10.1007/978-3-319-60876-1_15