Toward Efficient Package Maintenance: An Empirical Study of Patch Sharing across Four Linux Distributions

Jian Peng[‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
pengjian23@otcaix.iscas.ac.cn

Jiaxin Zhu[*†‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhujiaxin@otcaix.iscas.ac.cn

Yuwei Zhang[†]
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangyuwei@iscas.ac.cn

Wei Chen[†‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
wchen@otcaix.iscas.ac.cn

Guoquan Wu[†‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
gqwu@otcaix.iscas.ac.cn

Wei Wang[†‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
wangwei@otcaix.iscas.ac.cn

Jun Wei[†‡§]
Institute of Software, Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

Abstract

As a cornerstone of cloud computing and artificial intelligence, the Linux operating system has given rise to many popular distributions (distros), such as Fedora and Debian. A distro includes a rich collection of software packages derived from third-party open-source projects (upstream). Due to the nature of reusing upstream, some packages originate from the same upstream projects, noted as homologous packages. Maintaining packages is a core but tedious task for distros, which relies on continuously developing code patches. The maintenance of homologous packages could benefit from cross-distro collaboration. However, their shared origin may also lead to redundant efforts and delayed reuse of valuable maintenance work performed by other distros. To measure potential inefficiencies and explore possible solutions, this paper conducts an empirical study of patch sharing among homologous packages across four representative distros spanning three distinct families: Debian, Ubuntu, Fedora, and openEuler. We find that homologous packages are prevalent across distros, even across different distro families. Specifically, there are 13,873 homologous packages shared between Debian and Fedora, accounting for 40.3% and 56.1% of all packages in Debian and Fedora, respectively. For homologous packages with the same upstream version, we found that 19.5% of the patches are shared between Fedora and Debian, and 46.7% between Fedora and openEuler. However, these shared patches often exhibit remarkable propagation latency, with a median lag of 530.0 days between Fedora and Debian and 169.0 days between Fedora and openEuler. These results highlight the importance and necessity of cross-distro-family patch recommendations. We identified 17 shared Fedora-exclusive patches and contributed them to openEuler, among which 15 patches were successfully merged, demonstrating the potential of cross-distro-family patch recommendation and inspiring further research in this area.

CCS Concepts

• **Software and its engineering → Reusability**.

Keywords

Patch, Linux distribution, Maintenance, Software package

ACM Reference Format:
Jian Peng, Jiaxin Zhu, Yuwei Zhang, Wei Chen, Guoquan Wu, Wei Wang, and Jun Wei. 2026. Toward Efficient Package Maintenance: An Empirical Study of Patch Sharing across Four Linux Distributions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787836

1 Introduction

The Linux distributions have become the essential software infrastructure for cloud computing and artificial intelligence. Over the years, numerous open-source Linux distributions have emerged, including prominent examples such as Debian, Ubuntu, and Fedora. In addition to the Linux kernel, a Linux distribution typically includes a wide range of software components and applications, such

[*]Jiaxin Zhu is the corresponding author.
[†]Affiliated with University of Chinese Academy of Sciences (CAS) and Key Laboratory of System Software CAS.
[‡]Affiliated with University of CAS, Nanjing.
[§]Affiliated with Nanjing Institute of Software Technology.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3787836

as compilers, libraries, frameworks, and end-user programs, many of which originate from third-party open-source projects. These third-party open-source projects serve as the foundational sources for the software integrated into the distribution and are commonly referred to as ***upstream*** projects. The source code released by upstream projects is typically bundled with patches and configuration scripts into ***packages*** maintained by distribution maintainers [1]. Packages across different distributions may be derived from the same upstream project. We refer to such packages, which originate from the same upstream project but are included in different distributions, as ***homologous packages***. For brevity, we refer to a Linux distribution as a ***distro*** throughout this paper.

Although each Linux distro carries out its own development and maintenance, many homologous packages encounter similar technical challenges and maintenance efforts. However, existing approaches to systematically identify and analyze these commonalities remain inadequate, limiting opportunities for collaboration, knowledge sharing, and the dissemination of best practices. Consequently, effectively uncovering and leveraging cross-distro homogeneity has become a key challenge in enhancing the efficiency of package maintenance and reducing overall costs [2].

The inherent diversity of the Linux distro ecosystem further exacerbates collaboration challenges. Major distros are typically grouped into several distinct distro ***families*** [3], among which the Red Hat and Debian families are the most prominent, adopting the RPM and DEB package management systems [4], respectively. These two systems differ substantially in package structure, patch management, and tool chains, introducing technical barriers to patch migration and effort reuse. To enable a comprehensive investigation of package maintenance practices across these paradigms, this study selects Fedora, a long-standing and widely used representative of the Red Hat family, and Debian and Ubuntu from the Debian family. We also include openEuler, an RPM-based distro from a different family, in which we have participated.

We study patch sharing across distros from three perspectives: (1) the pervasiveness of homologous packages across distros, (2) the characteristics of shared and non-shared patches within the homologous packages, and (3) the timing of patch introduction across distros. We also explore the potential for cross-distro patch recommendations. We identified homologous packages across Fedora 41, openEuler 24.03, Debian 12.7, and Ubuntu 24.04. Our results show that Debian and Ubuntu share the largest number of homologous packages, accounting for 96.6% and 89.1% of their package sets, respectively, which aligns with their derivative relationship. Subsequently, we identified homologous packages that share identical upstream versions and conducted cross-family comparisons, specifically between Fedora and Debian, and between Fedora and openEuler. Our results show that 50.29% and 85.34% of the analyzed packages in the Fedora–Debian and Fedora–openEuler comparisons exhibit divergent patch sets. Even among the packages that share at least one patch, the overall patch set overlap remains limited, averaging 46.73% for Fedora–openEuler and 19.45% for Fedora–Debian. Moreover, shared patches experience introduction delays between distro families, with median values of 530.0 days for Fedora–Debian and 169.0 days for Fedora–openEuler, highlighting that cross-distro-family patch sharing and propagation remain considerably delayed.

Beyond the quantitative results of patch sharing, we demonstrate the practical potential of cross-distro-family patch recommendation. From homologous packages with identical upstream versions, we manually selected 17 patches that were present in Fedora for prolonged periods of time but absent in openEuler, based on their technical relevance and potential value. All 17 patches were submitted to the openEuler community, successfully applied to the corresponding packages, and passed the CI checks. As of this writing, 15 patches have been reviewed and accepted by openEuler package maintainers. This demonstration sheds light on the potential of cross-distro-family patch recommendations to reduce redundant maintenance efforts and improve package quality in an effective and efficient manner. The data and scripts used in this study are publicly available on Zenodo [5].

The main contributions of this paper are as follows:

- We present a comprehensive quantitative empirical study of package patch sharing among Linux distributions from distinct families, highlighting evidence of shortcomings in package maintenance.
- We identified and contributed 17 Fedora-exclusive patches from homologous packages to openEuler, demonstrating the potential of cross-distro-family patch recommendation and motivating further research in this area.
- We curate a dataset of homologous packages and their corresponding patch sets across four mainstream Linux distributions.

2 Preliminaries and Motivation

2.1 Linux Distros and Their Software Packages

A *Linux distribution (distro)* is an operating system based on the Linux kernel. Apart from the Linux kernel, it also incorporates a rich variety of open-source applications and components. For example, popular distros, such as Debian, Ubuntu, and Fedora, include a wide range of software such as desktop environments (e.g., GNOME or Unity), office suites (e.g., LibreOffice), and terminal emulators (e.g., GNOME Terminal or Konsole) [6]. Each distro manages its own software repositories, release cycles, and policies, offering customized solutions to meet diverse user needs and use cases [7]. The early distros introduced two widely known package management systems, i.e., *RPM* and *DEB*, to manage the software applications and components. These packaged applications and components are referred to as ***packages***, In particular, source packages are used during development, whereas binary packages are used for release. For brevity, we often use the term package to refer to ***source packages*** in this paper.

2.2 Development and Maintenance of Linux Distro Packages

In the Linux ecosystem, ***upstream*** typically refers to the original open-source projects maintained by their respective communities (e.g., LibreOffice and MySQL), which continuously release new versions of code that include bug fixes, new features, and other improvements. Distros (such as Debian, Fedora, and openEuler) integrate, package, and adapt the software from upstream projects to construct their own tailored systems. At the same time, distros

are related through lineage, giving rise to several major families[1]. Distros within the same family share a number of similarities. As shown in Figure 1, Debian is the root of the Debian family, while Ubuntu is derived from Debian and reuses Debian's code, package management system (DEB), repository structure, etc. In turn, Linux Mint[2] is a derivative of Ubuntu and, indirectly, of Debian. This derivative model enables rapid propagation of innovation and improvements throughout the Linux ecosystem, while also fostering collaborative software maintenance and adaptation [8].

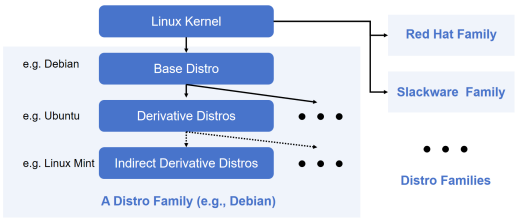


Figure 1: Linux distribution family tree.

Figure 2 shows an illustrative example, where package development and maintenance in Linux distros involve monitoring upstream releases, integrating new versions, and adapting them through patch management and configuration. In addition, derivative distros often import and update term packages from their parent distros rather than directly from the original upstream projects. Each package has a ***Git repository*** maintained by distribution developers. A package's Git repository serves as the source shared by all releases of a distro that include the package. It mainly contains patches and configuration files. Code from upstream projects is kept separate and remains unmodified. Packages are built and tested in controlled environments, with rigorous patch review and quality assurance processes ensuring compliance with distro policies [9]. Coordinated release management delivers tailored packages to end users, while continuous feedback from operational environments drives iterative improvement and sustained maintenance [10].

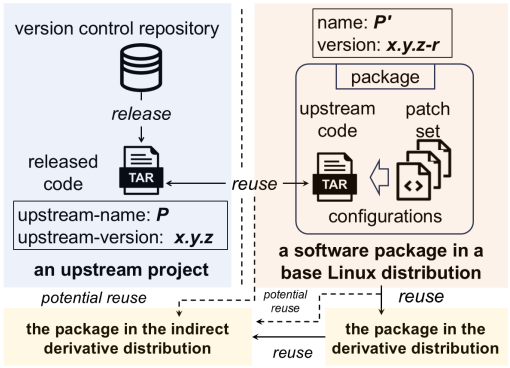


Figure 2: An illustrative example of a software package in a Linux distro and its upstream.

[1]https://distrowatch.com/dwres.php?resource=family-tree
[2]https://www.linuxmint.com/

It should be noted that the names and versions involved in the above process differ between the upstream and distro perspectives. As illustrated in Figure 2, the upstream project *P* releases the **upstream version** *x.y.z*, and the distro reuses the released code to build a package named *P'*. The package name *P'* follows the name of the upstream project, but may be a variant with changes in letter case or other minor modifications. The package also has its own version, *x.y.z-r*, where *x.y.z* denotes the upstream version, and *r* indicates changes in the patch set or configurations of the package. Examples of actual package names and versions can be found in Section 4.2.

There are several sources of patches within a package. Some patches are independently developed by distro package maintainers or contributors, either because the upstream has not yet addressed the issues of bug/vulnerability fixes and new features, or due to the specific requirements of the distro. There are also patches derived from third parties, such as upstream projects, parent distros and other distros. For example, Ubuntu launched the Patch Pilot program[3] to support and coordinate patch contributions. These patches help ensure the inclusion of essential improvements and fixes. This diversity in patch sources enhances the flexibility and responsiveness of distro package maintenance. However, recent empirical studies [1] have found that many high-priority bugs are fixed only within distros and are not promptly submitted upstream, resulting in redundant maintenance efforts and increasing technical debt for distro maintainers. These challenges underscore the urgent need for systematic mechanisms to track, manage, and recommend reusable patches across distros and upstream projects [11, 12].

3 Research Questions

We investigate the commonalities, differences, and sharing of the patches in homologous packages across major Linux distros. Given the numerous collaborative efforts within the same family (i.e., a base distro and its direct/indirect derivatives) and the explicit reuse relationships among them, our study mainly focuses on collaboration across different families. We define the following four research questions (RQs), which focus on the extent of homologous packages, the sharing of patches among them, and the delay in patch propagation.

RQ1: What is the distribution of homologous packages across the studied Linux distributions?

Our study stems from the widespread practice of reusing open-source projects to build Linux distro packages. Many homologous packages likely exist across various Linux distros, including those belonging to both the same and different families. As a first step, we aim to quantify the proportion of homologous packages among Linux distros to reveal the importance of investigating their development and maintenance.

RQ2: What are the shared and non-shared patches in homologous packages across the studied distributions from different families?

We observed notable differences in the patch sets of many homologous packages. To better understand these differences, we analyze the characteristics of both shared and non-shared patches to investigate whether non-shared patches could also be reused

[3]https://documentation.ubuntu.com/project/community/contribute/patch-pilots/

across distro families to improve package maintenance efficiency. The answer to this question may help measure both the existing and the additional efforts needed to maintain homologous packages across distro families.

RQ3: When were the shared patches introduced in the development of the studied distributions from different families?

Existing shared patches indicate the presence of common issues that need to be addressed. However, the timing of their introduction into the package Git repositories varies across distros. Patches that address critical issues should be adopted as early as possible. Therefore, we aim to investigate potential inefficiencies in patch reuse and quantify the delay of reuse.

RQ4: Can patches be recommended across the studied distributions from different families?

The first three questions aim to reveal the necessity of and inefficiencies in patch sharing across distro families. In this RQ, we seek to explore whether there are opportunities to address these inefficiencies. We assess the feasibility of cross-distro-family patch recommendation by examining whether patches developed and maintained in one distro can be effectively recommended and accepted by by distros from other families. The answer to this question can serve as evidence supporting the development of automated tools that promote patch sharing.

4 Methodology

To answer the research questions, we follow a multi-step approach shown in Figure 3: collecting packages from four mainstream Linux distros, identifying homologous ones, and analyzing and recommending patches across distro families.

4.1 Dataset

We selected three representative long-standing distros, Fedora, Debian, and Ubuntu. As described in Section 2, Debian and Ubuntu belong to the same family, with Ubuntu being a derivative of Debian, whereas Fedora belongs to a different lineage. In addition, we included a prominent emerging distro, openEuler, from another lineage, in whose development we have participated. In November 2024, when this study began, the latest releases of the selected distros were Fedora 41, Debian 12.7, Ubuntu 24.04, and openEuler 24.03. The packages from these releases were used in this study. We collected the packages from their respective official mirrors. The number of packages in Fedora 41, Debian 12.7, Ubuntu 24.04, and openEuler 24.03 is 24,725, 34,467, 37,383, and 6,155, respectively.

The organization and application of patches in source packages differ between RPM and DEB package management systems. In Fedora and openEuler (RPM), each source package includes a `.spec` file that defines build logic and a set of patch files, which are applied sequentially during the build process [13]. In Debian (DEB), source packages consist of a `.dsc` descriptor, an upstream source archive (`.orig.tar.gz`), and a Debian-specific archive (`.debian.tar.gz`) containing packaging metadata and patches. The `debian/patches/` directory and its `series` file explicitly list and order the patches to be applied. Figures 4 and 5 illustrate these structures, based on which we extract patches from the source packages.

4.2 Homologous Packages Identification

As defined in Section 2.2, homologous packages are those that originate from the same upstream project. Different Linux distros follow distinct naming conventions. Package names may vary in letter case across distros. In addition, distros often add different prefixes or suffixes to indicate language bindings or package roles, such as *python-*, *php-*, *golang-*, *lib*, *nodejs-*, *-dev*, *-utils*, or *-devel*. Based on these observations, we identify homologous packages for RQ1 through a process that ranges from package name comparison to content comparison. We first compare package names in a case-insensitive manner and directly classify those with identical names as ***Exact_Match***. For the remaining unmatched packages, we first apply name normalization, removing distro-specific prefixes, suffixes, or delimiters, and identifying additional matches based on standardized names. To avoid false positives, we draw on prior work that leverages project metadata [14], such as consistent project names in homepage or repository URLs for aligning upstream projects, and employs textual similarity measures to assess package description similarity. Using Python's scikit-learn library, we preprocess the package descriptions by removing punctuation, converting English text to lowercase, and applying language-aware tokenization, followed by constructing TF–IDF representations. Since textual similarity serves only as a supporting signal, we adopt a relatively loose similarity threshold of 0.6 to identify more homologous package candidates, above which packages are labeled as ***Std_Match***. In some cases, packages may have entirely different names across distros, even after removing prefixes and suffixes. For instance, *libfits-java* in Debian and *nom-tam-fits* in Fedora form a pair of homologous packages, illustrating that Std_Match may still fail to capture such relationships. Repology[4], a public cross-distro package monitoring platform, along with its Repology-Rules[5] framework, helps associate upstream projects with packages across multiple distros. The associations are maintained in continuously updated database dumps, and we downloaded the latest dump and imported it into our local database. It is straightforward to obtain homologous package candidates within the same distro family, such as Debian and Ubuntu, by querying the database for *projects*. For homologous package candidates across different families, we also take into account the *related projects* identified by Repology. This step also helps eliminate packages that share identical names across distros but do not originate from the same upstream project.

Finally, three of the authors manually reviewed all Std_Match candidates as well as additional candidates obtained from Repology. They did not have any disagreements. We paid particular attention to packages with significantly different version numbers to ensure the accuracy of our identification.

The version-aligned homologous packages are analyzed to address RQ2 to RQ4. This selection eliminates upstream-side differences, allowing the comparison to focus solely on distro-specific variations. Furthermore, we selected the Fedora–Debian and Fedora–openEuler pairs for comparisons. As elaborated in the preliminaries and research questions, the base distros and their direct and indirect derivatives have established mechanisms for inherited

[4]https://repology.org/
[5]https://github.com/repology/repology-rules

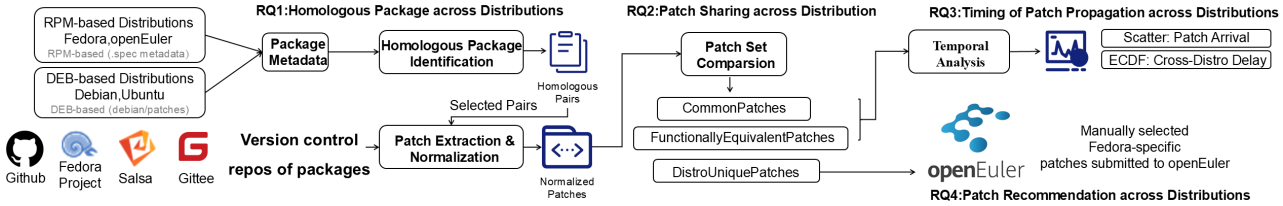


Figure 3: Approach overview of investigating patch sharing in homologous packages across Linux distros.

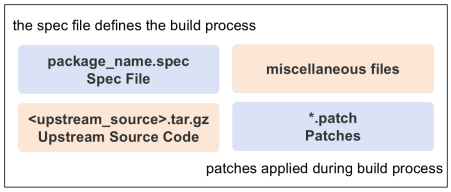


Figure 4: File structure of an RPM source package (SRPM).

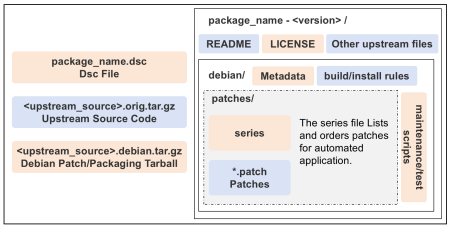


Figure 5: File structure of a DEB source package.

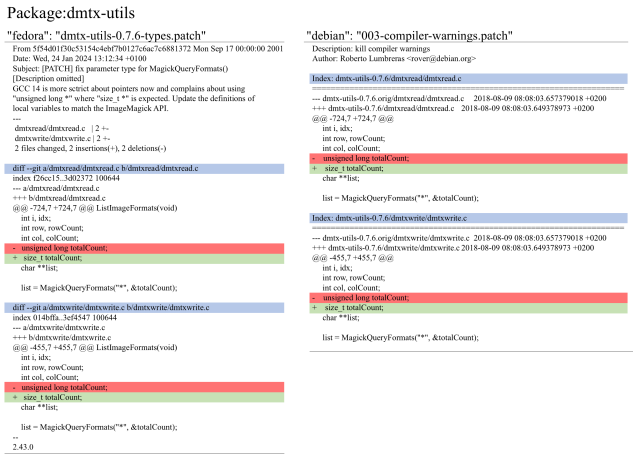


Figure 6: Functionally equivalent patches for `dmtx-utils` in Fedora and Debian.

collaboration in package and patch reuse, whereas this paper focuses on distros from different families. The Fedora–Debian pair allows us to examine the challenges and opportunities of migrating patches across different packaging paradigms (i.e., DEB vs. RPM), while the Fedora–openEuler pair is used to examine patch reuse across families that nonetheless share the same packaging system (RPM).

4.3 Comparison of Patches in Homologous Packages

We abstract the differences between the DEB and RPM formats to enable consistent patch comparison.

4.3.1 Patch Extraction. For RPM packages, patches are extracted by parsing patch declarations in the `.spec` file, as well as handling associated commands such as `%patch`, `%autopatch`, or `%autoset up`, which may affect patch file path levels. For Debian packages, we locate and parse the `series` file under the `debian/patches/` directory, which specifies the application order of all patch files. Each listed patch is then located relative to the repository root or the unpacked source tree. We also enabled detailed logging to monitor package downloads and parsing, thereby ensuring complete patch extraction.

4.3.2 Patch Normalization. Automated comparison requires a clear understanding of the standard structure of patch files as used in Linux distros. As shown in Figure 6, a patch file consists of metadata (such as filename, author, or description), followed by one or more blocks specifying file paths and the corresponding code changes, commonly referred to as *hunks*. Each hunk contains the context of the modification and highlights added (green) or deleted (red) lines. The example in Figure 6 shows patches for two homologous `dmtx-utils` packages from Fedora and Debian. Although the patch filenames, metadata, and paths of the modified code differ, both patches implement the same code change: updating the type of `totalCount` from `unsigned long` to `size_t` in two source files. Such cases highlight the challenges in automated comparison, namely, that functionally equivalent patches may differ substantially across distros, making normalization and structural analysis indispensable.

We normalize the patches by retaining only the essential information, such as the actual code changes, the modified files, and the affected line numbers. The normalization process removes irrelevant formatting differences, path prefixes, file rename suffixes, and unifies line endings and encodings. This helps ensure that structurally equivalent patches are not overlooked due to superficial differences, thereby enabling accurate automated comparison and equivalence detection [15].

4.3.3 Shared Patch Identification. A normalized patch is represented by three components: the set of modified files, the number

of changed lines, and the hunks. If all three components are identical between two patches, we label them as shared. For the remaining unmatched patches, we calculate a composite similarity score based on modified files and code changes to identify patches with highly similar core content.

Comparison of Changed File Set. To account for package directory structure differences across distros, such as directory renaming, variations in source tree layout, or the use of commands like `%patch -P1` that strip leading path components, we normalize patch file paths by comparing only filenames and, when available, including up to the last two directory components for matching. Specifically, we define the normalized path of a file p as:

$$\text{NormPath}(p) = \begin{cases} \text{last two components of } p, & \text{if Depth}(p) \geq 2 \\ \text{FileName}(p), & \text{otherwise} \end{cases} \tag{1}$$

This approach helps to align semantically identical files that may differ in superficial directory prefixes across distros. For each file in the set modified by patch P_1, we consider it matched with a file in patch P_2 if their normalized paths are equal. The similarity score is then computed as:

$$S_{\text{path}} = \frac{|F_m|}{|F_1 \cup F_2|} \tag{2}$$

where F_1 and F_2 denote the sets of modified files (after normalization) in each patch, and F_m denotes the number of matched files with the same normalized paths.

Comparison of Code Changes. We extract all added and deleted lines from each patch, tokenize them into n-grams, and compute the Jaccard similarity for both additions and deletions:

$$S_{\text{code}} = 0.5 \cdot J(A_1, A_2) + 0.5 \cdot J(R_1, R_2) \tag{3}$$

where A_1 and A_2 are the n-gram sets of added lines, R_1 and R_2 are the n-gram sets of deleted lines in each patch, and $J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$ denotes the Jaccard similarity.

Composite Patch Similarity. The patch similarity score is computed as a weighted sum:

$$S_{\text{composite}} = x \cdot S_{\text{path}} + y \cdot S_{\text{code}} \tag{4}$$

S_{code} and S_{path} contribute differently to patch identification. S_{code} captures the semantics of patches but may lead to false positives caused by shared template patterns and false negatives due to contextual drift or formatting differences. S_{path} helps alleviate these issues. We evaluated several weighting pairs (0.6/0.4, 0.7/0.3, 0.8/0.2, 0.9/0.1) and found that assigning weights of 0.8 to S_{code} and 0.2 to S_{path} yields the most reliable identification performance. Analysis of the sample set further shows that most shared patches naturally exceed a composite similarity of 0.7, motivating the use of 0.7 as our identification threshold. When $S_{\text{path}} = 1$, this threshold requires $S_{\text{code}} > 0.625$ under the 0.8/0.2 configuration. However, some true shared patches fall below this value due to differences in comments, whitespace, or coding style. To account for this, we compute an enhanced code similarity S'_{code} using Python's *difflib.SequenceMatcher* on comment-stripped code. If a patch pair satisfies $S_{\text{code}} < 0.625$ but $S'_{\text{code}} > 0.8$, we apply a correction:

$$S'_{\text{composite}} = \frac{S_{\text{composite}} + S'_{\text{code}}}{2} \tag{5}$$

This adjustment effectively covers borderline cases without impacting the remaining ones. Finally, patch pairs are considered shared if their (corrected) composite score exceeds 0.70.

4.4 Temporal Analysis of Shared Patch Propagation during Development

To answer **RQ3**, we conduct a temporal analysis of the introduction times of shared patches in the package Git repositories. First, based on previously collected package metadata, we identify and clone the corresponding Git repositories from multiple hosting platforms, including Fedora's `src.fedoraproject.org`, openEuler's `gitee.com/src-openeuler`, and Debian's `Salsa.debian.org`. Packages whose Git repositories cannot be located are excluded from our analysis. Second, we analyze the modification history of patch files by applying `git log --follow`, which maintains historical continuity in the presence of file moves or renames [16], and identify the date when each patch file was added to the repository. The dates are normalized to ISO 8601-formatted strings. We further visualize the temporal distro of patch introductions between distros using scatter plots, and employ empirical cumulative distro function plots to statistically analyze patch propagation delays, thereby revealing the order and latency of patch introduction across distros.

4.5 Cross-distro-family Patch Recommendation

For RQ4, we systematically evaluate the practical potential for cross-distro-family patch recommendation. Specifically, for homologous package pairs with identical upstream versions, we manually select patches exclusive to Fedora (i.e., present only in Fedora) that address security vulnerabilities, general feature enhancements, or compatibility improvements. These candidate patches are then validated by cloning the corresponding openEuler package repositories, applying the patch via `git apply --check`, and ensuring successful integration [17, 18]. Patches that pass this validation are submitted as pull requests to the openEuler community for peer review. To eliminate potential subjective bias, we submit the pull requests using an account that has no prior contributions to openEuler. This manual yet reproducible workflow mirrors the real-world process of patch porting and provides both practical and empirical evidence for the feasibility of automated cross-distro patch recommendation.

5 Results

5.1 Homologous Packages across Distributions

To address RQ1, we identified the homologous packages across openEuler 24.03, Fedora 41, Debian 12.7, and Ubuntu 24.04. Figure 7 presents the number of packages in each Linux distro, along with the number of homologous packages shared across the four distros. The dot plot at the bottom indicates the participating distros (highlighted as dark dots) for each group of homologous packages. The most prominent intersection occurs between Debian and Ubuntu, which share 33,293 homologous packages, representing 96.6% of all Debian packages and 89.1% of all Ubuntu packages. This substantial overlap reflects their direct derivative relationship: Ubuntu initially imported most of its package set from Debian and has since continued to update and expand the set through security updates, feature

enhancements, and distro-specific adaptations, resulting in an even larger package collection.

Beyond Debian–Ubuntu, we also observe a sizable set of homologous packages shared between Debian and Fedora. Specifically, the two distros share 13,873 homologous packages, accounting for 40.3% of all Debian packages and 56.1% of all Fedora packages. Despite the absence of a lineage relationship and their use of heterogeneous packaging systems (DEB vs. RPM), this level of overlap reflects the widespread reuse of popular upstream projects across the Linux ecosystem. It also suggests that substantial potential exists for cross-distro-family patch collaboration, even between technically divergent distros.

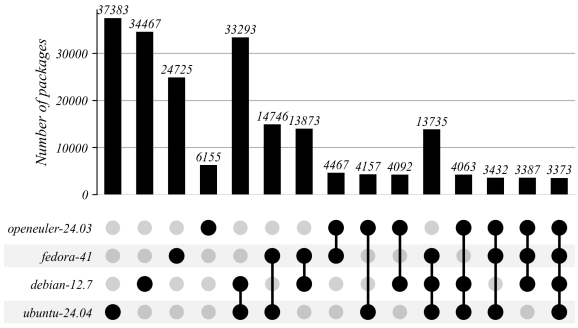


Figure 7: Number of homologous packages across Fedora, Debian, Ubuntu and openEuler.

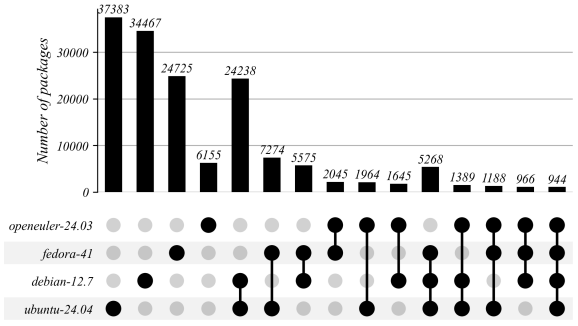


Figure 8: Number of homologous packages with the same upstream versions across Fedora, Debian, Ubuntu, and openEuler.

When examining openEuler, the largest set of homologous packages is found in its overlap with Fedora, comprising 4,467 packages, representing 18.1% of all Fedora packages and 72.6% of all openEuler packages. This reflects the close technological relationship between openEuler and Fedora: both rely on the RPM build toolchain, SRPM format, and `spec` scripting system, and share common upstream sources for the core packages. Such structural consistency reduces barriers to understanding and reusing patches across distros and provides a solid technical foundation and promising feasibility for future cross-distro collaboration. We also identified 3,373 packages that are universally shared across the four distros, highlighting the need for general cross-distro collaboration.

Among the homologous packages, we further looked into those of the same upstream versions. Figure 8 shows the remaining homologous packages. The number of version-aligned packages is relatively small: 5,575 between Fedora and Debian and 944 across all four distros. This indicates that the selected distro releases are not fully aligned in the upstream versions of homologous packages, despite being the latest releases at the start of our study.

Answer to RQ1. We identified 13,873 homologous packages between Fedora and Debian (40.3% of all Debian packages and 56.1% of all Fedora packages) and 4,467 between Fedora and openEuler (18.1% of all Fedora packages and 72.6% of all openEuler packages), highlighting both the need and potential for cross-distro-family collaboration.

5.2 Patch Sharing across Distribution Families

To understand the rationale behind patch sharing and answer RQ2, we classify them into three categories:

- `CommonPatches`: Patches that are exactly identical in content.
- `FunctionallyEquivalentPatches`: Patches that achieve the same maintenance purpose but are not identical after normalization (Section 4.3.2) due to differences in formatting, comments, or minor code variations.
- `DistroExclusivePatches`: Patches that exist only in a single distro (exclusive to a distro).

We treat both `CommonPatches` and `FunctionallyEquivalentPatches` as shared patches. For `DistroExclusivePatches`, we manually inspect a representative subset to assess their potential for cross-distro reuse and recommendation.

Table 1 presents key statistics on patch set overlap for version-aligned homologous packages between Fedora 41 and openEuler 24.03, and between Debian 12.7 and Fedora 41. **Unless otherwise specified, all homologous packages referenced in the following results are version-aligned.**

Some homologous packages were found to have no patches and are not considered in the following analysis. Among the 5,575 homologous packages between Fedora and Debian, 2,872 (51.5%) have at least one patch in either distro. In the case of Fedora and openEuler, 859 out of 2,045 packages (42.0%) contain at least one patch in either distribution. The degree of patch sharing between distros is markedly different across the two pairs. Only 421 packages (14.7% of those with patches) have shared patches between Fedora and Debian, whereas for Fedora and openEuler, 427 packages (49.7%) with patches exhibit overlap. Distinguishing `CommonPatches` from `FunctionallyEquivalentPatches` further reveals that `CommonPatches` constitute the vast majority of shared patches in both pairs.

The mean overlap ratio, defined as the average proportion of shared patches across all patches in homologous packages, differs substantially between the two distro pairs: it is 19.45% for Fedora–Debian and much higher at 46.73% for Fedora–openEuler. This result is expected since Fedora and openEuler both use the RPM package management system and maintain similar workflows, leading to greater patch sharing. In contrast, the distinct package management systems and conventions of Debian and Fedora naturally constrain direct patch reuse. Still, it is noteworthy

Table 1: Patch Set Overlap Statistics for Homologous Packages across Distros

Comparison	Total Packages	Packages with Patches	Packages with Shared Patches	Shared Patches (Exact / Functional)	Overlap Ratio (%)	Mean Overlap Ratio (%) (Overlapping Packages)
Fedora–Debian	5,575	2,872	421	634 / 120	14.66	19.45
Fedora–openEuler	2,045	859	427	1500 / 60	49.71	46.73

that nearly 20% overlap occurs even between these heterogeneous distros, demonstrating that patch exchange and reuse are possible across different package management systems. These results highlight the substantial practical value and impact of cross-distro-family patch recommendation and collaboration.

While many patches are shared across distros, a significant proportion of packages contain exclusive patches. In the case of Fedora–Debian, 2,451 packages (85.34%) with patches contain exclusive patches. For Fedora–openEuler, this number is 432 packages (50.29%). These results indicate that, although cross-distro-family patch sharing exists, a significant amount of maintenance effort remains isolated within individual distros.

However, it would be an oversimplification to conclude that packages with entirely distinct patch sets lack any reusable patches. Our in-depth review of these non-shared patches reveals that some patches are strictly tailored for the needs of a specific distro. Meanwhile, many of these patches address general issues such as upstream bug fixes, security vulnerabilities, or feature enhancements that could potentially benefit other distros as well. For instance, in our dataset, approximately 75% of Debian–Fedora and 70% of Fedora–openEuler CVE-fixing patches were found to remain non-shared. Although such patches have not been adopted by peer distros, possibly due to differences in maintenance policies, stability requirements, or risk assessments, they nevertheless represent a valuable pool of candidates for cross-distro-family patch recommendation.

Answer to RQ2: Patch sharing is observed to be common across distributions from different families. Specifically, 49.7% of homologous packages in Fedora–openEuler and 14.7% in Fedora–Debian contain shared patches, highlighting the presence and underexplored potential of patch reuse.

5.3 Timing of Patch Propagation during the Development

To answer RQ3, we performed a systematic investigation into the introduction times of shared patches in Fedora–Debian and Fedora–openEuler. Figures 9 and 10 display scatter plots comparing the introduction dates of shared patches in Fedora (x-axis) and the counterpart distro (y-axis). Each point on the plot corresponds to a pair of shared patches between the two distros, and the dashed diagonal line ($y = x$) represents ideal synchronization, i.e., patches introduced at exactly the same time in both distros. In an ideal scenario where patch propagation is timely and systematic, we would expect all points to closely follow the diagonal, indicating minimal lag in patch adoption across distros.

In the case of Fedora–Debian (Figure 9), although a subset of points lies close to the diagonal, most points are clearly scattered

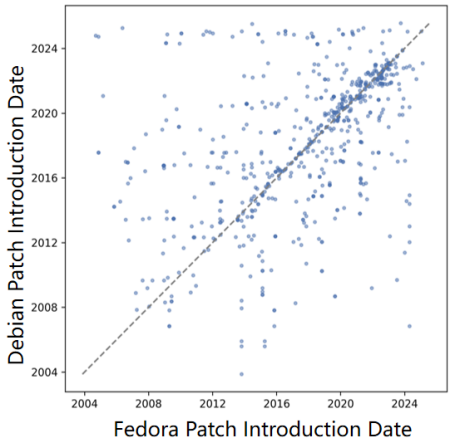


Figure 9: Introduction time of shared patches between Fedora and Debian.

above and below it, quantitatively indicating substantial and bidirectional temporal divergence in how Fedora and Debian introduce the same patches: points above the diagonal represent patches introduced earlier in Debian, whereas points below represent patches introduced earlier in Fedora. The symmetry of the plot suggests that Fedora and Debian alternately take the lead in introducing shared patches, indicating a bidirectional flow and a degree of operational independence in patch propagation. The wide horizontal spread of points reveals that while some patches are introduced only a few days apart, or even within the same 24-hour window, many others experience substantial delays, sometimes lasting several years, between their adoption in one distro and their eventual integration into the other. The delays are measured from the Git repositories, reflecting the development perspective. These delays indicate that inefficiencies exist in patch sharing across Linux distros during development. Moreover, the delays in the Git repositories can influence the timing of when fixes and enhancements are made available to end users, as patches must first be integrated into the Git repositories before being included in official releases. It should also be noted that each distribution follows its own release policy, which could potentially influence when patches are introduced into Git repositories and, in turn, affect when these fixes and enhancements become visible to end users, e.g., rolling-release[6] policy tends to integrate patches into releases rapidly.

For the Fedora–openEuler pair (Figure 10), a distinct pattern emerges. Initially, many patches shared identical introduction dates, corresponding to openEuler's mass repository seeding during its initial launch (2019–2021) [19]. To better reflect real maintenance

[6]https://www.webopedia.com/definitions/rolling-release/

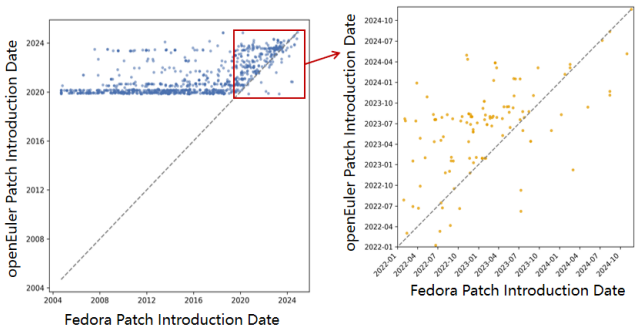
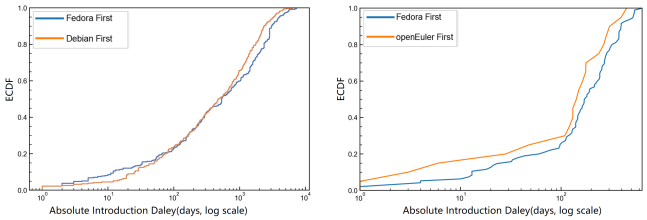


Figure 10: Introduction time of shared patches between Fedora and openEuler.

activity, we filtered out all such patches. When focusing only on patches from the most recent three years, there remains a clear and persistent delay in openEuler's patch adoption relative to Fedora.

To provide a complementary perspective on the temporal dynamics of patch propagation, we further employ empirical cumulative distro function (ECDF) plots [20] to quantify the pairwise introduction delay of shared patches between distro pairs (Figures 11a and 11b). The pairwise introduction delay for a given patch is defined as the absolute time interval (in days) between its initial inclusion in the first distro and its subsequent inclusion into the counterpart. These ECDF plots reveal considerable heterogeneity in patch propagation speeds across distros, ranging from rapid synchronization to significant delays.



(a) Fedora and Debian (all shared patches)

(b) Fedora and openEuler (excluding mass migrations)

Figure 11: ECDF of patch introduction delays across Fedora, Debian, and openEuler.

Figure 11a presents the ECDF of patch introduction delays between Fedora and Debian, distinguishing whether Fedora or Debian introduced the patch first. The median introduction delay is 530.0 days, and approximately 50% of the patches are included by both distros within a few hundred days. Both curves indicate that patches experience extended delays, sometimes spanning several years, before being adopted by the other distros.

Figure 11b illustrates the ECDF of patch introduction delays between Fedora and openEuler, based on patches adopted between 2022 and 2024. In this case, the median introduction delay is 169.0 days. Combining insights from Figure 10, the vast majority of shared patches are first included in Fedora, with openEuler integrating them after substantial delays. Specifically, for over half of these patches, openEuler's adoption time is several hundred days later than Fedora's. Moreover, the 'openEuler First' curve is positioned to the left of the 'Fedora First' curve, indicating shorter delays when patches appear earlier in openEuler, whereas longer delays are observed when Fedora is the earlier adopter.

Patch introduction may be influenced by development practices tied to distro release cycles. However, we find that patches can be introduced at any time within a release cycle. Moreover, the observed delays are substantially longer than typical release cycles, particularly minor release cycles for issue fixes, which typically span only a few months. This suggests that the delays are not inherently constrained by release-cycle–based development practices, and package maintainers may either be unaware of reusable patches from other distro families or encounter obstacles in applying them in a timely manner.

Answer to RQ3: The propagation of shared patches across distro families is highly variable and often asymmetric: while 20% are adopted by all distros within approximately 30 days, 40% require over 1000 days to be fully included. Delays occur in both directions between Fedora and Debian, but patch adoption between Fedora and openEuler is mostly one-sided, with most patches introduced first in Fedora.

5.4 Patch Recommendation across Families

To evaluate the feasibility and practical impact of cross-distro-family patch recommendation (RQ4), we targeted Fedora 41 and openEuler 24.03, and attempted to contribute exclusive patches with broad applicability from Fedora 41 to openEuler 24.03. Our findings for RQ3 reveal a highly asymmetric propagation pattern: most shared patches originate in Fedora, whereas openEuler typically incorporates them. This observation highlights the value of contributing Fedora patches into openEuler, as it may help determine whether such delays are deliberate or stem from genuine inefficiencies.

We selected 17 candidate patches through manually examining their technical purpose and applicability. These patches span different categories and packages, and each can be cleanly adapted, typically through minor hunk or context adjustments, to the counterpart package in openEuler, ensuring that our evaluation reflects practically transferable, real-world patches. In addition, approximately half of these patches had been present in Fedora for more than two years at the time of our study. We submitted the 17 patches via pull requests to openEuler's official Git repositories[7] for the corresponding packages. As of this writing, 15 (88.23%) patches have been successfully accepted.

Table 2 lists a representative subset of these pull requests. A clear trend emerges: security-related patches are reviewed and merged quickly, sometimes within just one day. For example, the enscript-CVE-vasnprintf.patch and 0005-Fix-for-CVE-2017-16516.patch were both accepted and integrated within three days. In contrast, compatibility or code modernization patches (e.g., "c99" series for code

[7]https://gitee.com/organizations/src-openeuler/projects

Table 2: Representative Patches Submitted to openEuler

PackageName	PatchName	Patch Type	Status	Review Time (days)
aenscript	aenscript-CVE-vasnprintf.patch	Security Fix	PR merged	1
yajl	0005-Fix-for-CVE-2017-16516.patch	Security Fix	PR merged	3
jdom	CVE-2021-33813.patch	Security Fix	PR merged	3
dwz	dwz-0.15-index9.patch	Feature-add	PR merged	5
compface	compface-c99.patch	Compatibility	PR merged	30
execstack	execstack-configure-c99.patch	Compatibility	PR merged	69
zssh	zssh-c99.patch	Compatibility	PR passed CI	/

standard conformance) experienced much longer review times, often stretching to several weeks or even months, such as compface-c99.patch(30 days) and execstack-configure-c99.patch (69 days). Two compatibility patches remained open.

This pattern highlights a common phenomenon in open-source community maintenance: patches with significant security impact are prioritized for rapid review and integration, while non-critical compatibility or enhancement patches often face longer evaluation cycles due to limited reviewer bandwidth, perceived urgency, or downstream testing requirements.

Overall, the successful integration of the majority of recommended patches demonstrates the practical utility of systematic cross-distro patch recommendation, which fosters collaboration, reduces redundant effort, and accelerates the dissemination of important fixes across the Linux ecosystem. Moreover, the short merge times of most pull requests suggest that reusable patches can be integrated soon after becoming known, making deliberate policies unlikely contribute to the delays observed in RQ3.

Answer to RQ4: Cross-distro-family patch recommendation is practically feasible, enabling distributions to notice and adopt beneficial patches that might otherwise be overlooked, thereby improving maintenance efficiency across the Linux ecosystem.

6 Implications

We have explored the practice of patch sharing across four main Linux distros and examined the potential for making this process more efficient. Our findings have several implications for both practitioners and researchers.

Collaborative Package Maintenance. Homologous packages are prevalent across Linux distros, but their patch sets often diverge, resulting in duplicated maintenance efforts and missed opportunities for timely patch reuse. While some patches serve distro-specific purposes, our pull requests merged in openEuler indicate that a subset of non-shared patches could be applicable to other distros, even across different distro families. Package maintainers may track their counterparts in other distros to discover valuable updates, particularly reusable patches that address bugs and vulnerabilities. To promote such collaboration, existing platforms such as the Ultimate Debian Database [21] and programs such as Patch Pilot could be extended to encompass distros from other families.

Automated Patch Propagation. Beyond governance and management approaches to improve the maintenance of homologous packages, researchers can also develop techniques to automate the propagation of patches across them. Our shared patch identification approach lays the groundwork for developing automated tools, while we have demonstrated the feasibility of patch recommendation. To automate this process, two types of approaches are required. The first is to identify reusable patches, and the second is to adapt them to the distro-specific code context of each package. Recent Large Language Models (LLMs) have shown impressive performance on software maintenance tasks. CodeX [22] and CodeT5+ [23] have significantly advanced the state of the art in code understanding and summarization, while unified pre-training methods [24] leverage both code and natural language data, resulting in models with improved generalization abilities across a wide range of software development and maintenance scenarios. These techniques could facilitate more effective automated patch propagation.

7 Threats to Validity

7.1 Internal Validity

The automated steps in identifying homologous packages and shared patches may introduce some inaccuracies. To mitigate this, we employed Repology to identify homologous packages and conducted a thorough manual review to eliminate false positives introduced by the broad selection. The manual review reveals that we mined 5,289 additional homologous packages from Repology, of which 351 are false positives. We also identified 8 homologous packages that cannot be obtained solely through Repology.

Exhaustive manual validation of shared patch identification was infeasible due to the large number of patches. We therefore designed an automated identification process with enhanced accuracy and randomly sampled 100 identified patches for manual review. Only three were false positives, yielding an accuracy of 97% and indicating the robustness of our method. At a minimum, the reported numbers and proportions of homologous packages and shared patches represent a lower bound.

For patch propagation delay analysis, we take a development perspective by examining Git repositories. Thus, the results do not reflect the perspective of end-user releases. The time when a patch is released to end users is influenced by each distro's release timing and policies. Nevertheless, our pull requests to openEuler confirm that reusable patches can be merged into the Git repositories shortly after submission (most of them were merged within one month, while a few lower-priority patches took slightly longer), indicating that the delays are not shaped by deliberate policies. Additionally,

we were unable to locate publicly available source code repositories for some Debian packages, and thus excluded their associated shared patches from the analysis. However, such cases account for less than *6%* of the shared patches and thus have negligible impact.

7.2 External Validity

In this case study, we examined four popular Linux distros, Debian, Ubuntu, Fedora, and openEuler. The results may not generalize to other distros. To investigate patch sharing and patch introduction delay, we focus on distros from different families, and the results may not apply to those that have direct or indirect derivative relationships. Given that our study involves extensive manual review of packages and patches, we aimed to strike a balance between representativeness and manual effort. Therefore, we selected these four distros based on the latest releases available at the beginning of our study.

Moreover, our analysis primarily focuses on homologous packages with identical upstream versions. However, packages with differing upstream versions may also benefit from patch sharing, which warrants further investigation in future research on package patch recommendation. The pull request process involves participation from the open-source community, which is inherently uncertain. Therefore, we submitted only a representative subset of patches covering a variety of packages and patch types.

8 Related Work

Patch maintenance is a long-standing challenge in large-scale, open-source software engineering. Linux distro communities have developed several programs and systems to facilitate package maintenance. Meanwhile, prior research has mainly focused on patch porting, security patch propagation, and automated patch analysis.

Package Tracking. Repology is a public platform that monitors software packages across many Linux distros, providing information on open-source projects and their corresponding packages in each distro. It helps developers track upstream releases and compare versions adopted by different distributions. We employed Repology for homologous package identification, obtaining a more comprehensive result. However, our manual inspection revealed that some inaccurate data were also included. There are also some intra-distro and intra-family tracking platforms used for package maintenance. The Ultimate Debian Database is a representative and well-established example [21], aggregating extensive Debian-related data, including package and source information from Debian and Ubuntu, as well as bug reports from the Debian Bug Tracking System, to support data mining and quality assurance.

Patch Porting and Ecosystem Fragmentation. Early work by Ray and Kim [25] systematically analyzed the BSD family, revealing recurring patch reuse and propagation. Later studies extended this thread to the Linux and Android kernels, with Li et al. [26] and Zhang et al. [27] documenting delays and fragmentation in patch flow across mainline, stable, and downstream branches. Further, collaborative patch review and pull-based workflows introduce their own challenges [28]. Fragmentation is also apparent in broader ecosystem evolution and API stability [29], and can hinder dependency upgrade efforts [30].

Security Patch Lifecycle and Vulnerability Propagation. Another focus has been on the latency and coverage of security patches. Li and Paxson [31] measured patch application and exposure times in open-source projects. Frei et al. [32] and Shahzad et al. [33] analyzed vulnerability patching lifecycles. More recent work, Farhang et al. [34] and Zhou et al. [19], examined security patch response in Android and openEuler, revealing gaps in timeliness and coordination. Patch uplift practices, such as in Firefox, have been studied for quality and scheduling impacts [35]. Incidents like Heartbleed [36, 37], Shellshock [38], Dirty COW [39], Sudo Baron Samedit [40], and Spectre/Meltdown [41, 42] highlight risks from delayed or inconsistent propagation. Xie et al. [43] further showed that 81% of CVE patches evolve post-release, undermining static vulnerability detection. Studies of open-source bug and patch characteristics confirm the diversity and complexity of real-world maintenance [15].

Automated Patch Classification and Analysis. To address the growing scale of patches, researchers have proposed automated techniques. PatchNet [44, 45] combines commit messages and code changes to identify stable Linux kernel patches, while PTracer [46] provides an end-to-end monitoring and recommendation system. Dang et al. [47] leveraged leveraged LLMs in MIGGPT to automate out-of-tree kernel patch migration using modular fingerprints. Additionally, automated pull requests and integration have been studied for dependency management [30]. Beyond classification, a series of tools have advanced patch generation and porting. Patch generation based on learning from human-written patches [48] and correct code [49], code transformation inference [50], and semantic learning [51, 52] have improved the accuracy of patch creation. Automated patch transplantation [53], backporting in Linux [54], and zero-shot patch porting [55] further reduces manual effort and enables automated repair to be applied at scale across projects.

Despite extensive research on patches, patch sharing in homologous packages across Linux distributions, a promising avenue for improving package maintenance, remains largely unexplored. To the best of our knowledge, this work presents the first empirical study on this topic. We are encouraged to see ongoing efforts that promote collaboration on package maintenance in practice. This paper also benefits from one of these efforts, Repology. However, these efforts either focus on distros within the same family or remain at a relatively coarse package-level granularity. Our results provide a more detailed and comprehensive view of patch sharing across distro families.

9 Conclusions

As the foundation of modern information technology, the Linux family of operating systems relies on collaborative maintenance by a global community of developers. Our analysis shows that while major Linux distros share thousands of homologous packages, patch reuse remains limited and often delayed across distro families. Notably, many non-shared patches could also be valuable for sharing, as demonstrated by our 15 accepted contributions to openEuler. These findings highlight the urgent need for systematic tools to identify and accelerate patch reuse, as the current inefficiencies lead to redundant maintenance efforts, slower security response, and fragmented collaboration cross distro families.

Acknowledgments

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320102, Major Project of ISCAS (ISCAS-ZD-202302), Youth Innovation Promotion Association of the Chinese Academy of Sciences (2023121).

References

[1] Jiahuei Lin, Haoxiang Zhang, Bram Adams, and Ahmed E. Hassan. 2022. Upstream Bug Management in Linux distros – An Empirical Study of Debian and Fedora Practices. *Empirical Software Engineering* 27 (2022), 134.

[2] Linux Foundation. 2024. *Linux Foundation Annual Report 2024.* Technical Report. Linux Foundation. https://www.linuxfoundation.org/resources/publications/linux-foundation-annual-report-2024

[3] D. Jin, N. Li, K. Yang, et al. 2025. A First Look at Package-to-Group Mechanism: An Empirical Study of the Linux distros. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 299–309.

[4] Diomidis Spinellis. 2012. Package Management Systems. *IEEE Software*, 29, 2 (2012), 84–86.

[5] Jian Peng. 2026. Filpped/LDPatch: LDPatch_v1.0.0. Zenodo. Version v1.0.0. Available at: https://doi.org/10.5281/zenodo.18150449.

[6] 2024. Debian Documentation. https://www.debian.org/doc.

[7] 2024. Linux distribution. https://en.wikipedia.org/wiki/Linux_distribution.

[8] Alexandre Decan, Tom Mens, and Maël Claes. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[9] Bram Adams, Kris De Schutter, Henk Tromp, and et al. 2008. The Evolution of the Linux Build System. *Electronic Communications of the EASST* 8 (2008).

[10] Luyao Ren. 2019. Automated Patch Porting Across Forked Projects. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* ACM, Tallinn, Estonia, 3. doi:10.1145/3338906.3342488

[11] Xin Tan, Minghui Zhou, and Brian Fitzgerald. 2020. Scaling Open Source Communities: An Empirical Study of the Linux Kernel. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* 1222–1234.

[12] Roberto Di Cosmo. 2018. Software Heritage: Why and How We Collect, Preserve and Share All the Software Source Code. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS).* IEEE, 2–2.

[13] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How has forking changed in the last 20 years? A study of hard forks on GitHub. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE).* IEEE, 445–456.

[14] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 404–414.

[15] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.

[16] Danielle Gonzalez, Joanna CS Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 391–401.

[17] Panyawut Sri-Iesaranusorn, Raula Gaikovina Kula, and Takashi Ishio. 2021. Does code review promote conformance? A study of OpenStack patches. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* IEEE, 444–448.

[18] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International conference on mining software repositories (MSR).* IEEE, 356–367.

[19] Minghui Zhou, Xiaohu Hu, and Wei Xiong. 2022. openEuler: Advancing a Hardware and Software Application Ecosystem. *IEEE Software* 39, 2 (March–April 2022), 101–105. doi:10.1109/MS.2021.3132138

[20] 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.* 2201–2215.

[21] Lucas Nussbaum and Stefano Zacchiroli. 2010. The Ultimate Debian Database: Consolidating Bazaar Metadata for Quality Assurance and Data Mining. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010).* IEEE, 52–61.

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[23] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven Hoi. [n. d.]. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing.*

[24] WU Ahmad, S Chakraborty, B Ray, and KW Chang. 2021. Unified pre-training for program understanding and generation.. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.*

[25] Baishakhi Ray and Miryung Kim. 2012. A Case Study of Cross-System Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* ACM, 53:1–53:11.

[26] Xingyu Li, Zheng Zhang, Zhiyun Qian, Trent Jaeger, and Chengyu Song. 2024. An Investigation of Patch Porting Practices of the Linux Kernel Ecosystem. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR).* ACM, 63–74.

[27] Zhiyun Zhang, Haoyu Zhang, Zhiyun Qian, et al. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *30th USENIX Security Symposium (USENIX Security 21).* 3649–3666.

[28] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th international conference on software engineering.* 285–296.

[29] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 109–120.

[30] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE).* IEEE, 84–94.

[31] 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.* ACM, 2201–2215.

[32] Stefan Frei, Matthias May, Ulrich Fiedler, and Bernhard Plattner. 2006. Large-Scale Vulnerability Analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense.* ACM, 131–138.

[33] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* IEEE, 771–781.

[34] Seyed Mohammadjavad Farhang, Mohammad Bilal Kirdan, Aron Laszka, and Jens Grossklags. 2019. Hey Google, What Exactly Do Your Security Patches Tell Us? A Large-Scale Empirical Study on Android Patched Vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium.* USENIX.

[35] Marco Castelluccio, Le An, and Foutse Khomh. 2019. An empirical study of patch uplift in rapid release development pipelines. *Empirical Software Engineering* 24, 5 (2019), 3008–3044.

[36] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, and Frank Li. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference.* ACM, 475–488. doi:10.1145/2663716.2663755

[37] I. D. Barbu and I. Bacivarov. 2014. Heartbleed—the Vulnerability that Changed the Internet. *International Journal of Information Security and Cybercrime* 3 (2014), 49–57.

[38] CERT Coordination Center. 2014. *Vulnerability Note VU#252743: GNU Bash Shellshock Vulnerability.* Technical Report. CERT/CC. https://www.kb.cert.org/vuls/id/252743/ 2024-06-06.

[39] Red Hat. 2016. *Dirty COW (CVE-2016-5195) - Linux Privilege Escalation Vulnerability.* Technical Report. Red Hat, Inc. https://access.redhat.com/security/vulnerabilities/DirtyCow 2024-06-06.

[40] Inc. Qualys. 2021. *Qualys Security Advisory: CVE-2021-3156 - Heap-Based Buffer Overflow in Sudo (Baron Samedit).* Technical Report. https://www.qualys.com/2021/01/26/cve-2021-3156/cve-2021-3156.txt 2024-06-06.

[41] Red Hat. 2018. *Spectre and Meltdown: Kernel Side-Channel Attacks.* Technical Report. Red Hat, Inc. https://access.redhat.com/security/vulnerabilities/speculativeexecution 2024-06-06.

[42] Google Project Zero. 2018. Reading Privileged Memory with a Side-Channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html Project Zero Blog, 2024-06-06.

[43] Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. 2024. Unveiling the Characteristics and Impact of Security Patch Evolution. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1094–1106.

[44] Thong Hoang, Julia Lawall, Richard J. Oentaryo, Yuan Tian, and David Lo. 2019. PatchNet: A Tool for Deep Patch Classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* IEEE, 83–86.

[45] Tung T. T. Hoang, Julia Lawall, Yuanyuan Tian, Gilles Muller, and David Lo. 2019. PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2471–2486. doi:10.1109/TSE.2019.2903183

[46] Yang Wen, Jicheng Cao, and Shengyu Cheng. 2019. PTracer: A Linux Kernel Patch Trace Bot. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1182–1185. doi:10.1109/ASE.2019.00131

[47] Pucheng Dang, Di Huang, Dong Li, Kang Chen, Yuanbo Wen, Qi Guo, Xing Hu, and Ninghui Sun. 2025. MigGPT: Harnessing Large Language Models for Automated Migration of Out-of-Tree Linux Kernel Patches Across Versions. *arXiv preprint arXiv:2504.09474* (2025).

[48] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 802–811.

[49] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 298–312.

[50] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

[51] 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*. 2366–2380.

[52] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-Granularity Patch Generation for Better Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1554–1566.

[53] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2020. Automated Patch Transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–36.

[54] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 633–645.

[55] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2024. Automating Zero-Shot Patch Porting for Hard Forks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 363–375.