JavaBackports: A Dataset for Benchmarking Automated Backporting in Java

Kaushal Kahapola
kaushal.21@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Sharada Galappaththi
sharadag.21@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Dinith Ranasinghe
dinith.21@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Ridwan Shariffdeen
shariffdeenr@acm.org
SonarSource
Singapore

Nisansa de Silva
NisansaDdS@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Srinath Perera
srinath@wso2.com
WSO2 Lanka (Pvt) Ltd
Sri Lanka

Sandareka Wickramanayake*
sandarekaw@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Abstract

Manually backporting critical patches to long-term support versions is both error-prone and often overlooked, resulting in substantial security risks. Progress in this area is constrained by the absence of datasets that capture the semantic complexities across versions, inherent to backporting in large Java ecosystems. To address this gap, we present `JavaBackports`, a curated dataset of 491 real-world backport instances, systematically selected and manually validated from more than 11,000 candidate patches in fifteen widely used open-source Java projects: *Druid, Elasticsearch, Hadoop, Kafka,* among others and four major *JDK* versions (*jdk11, jdk17, jdk21, jdk25*). To assess the utility of `JavaBackports`, we conduct preliminary experiments to evaluate the effectiveness of the state-of-the-art Large Language Models (LLMs) in zero-shot automatic patch backporting. The results indicate that current LLMs struggle with backporting tasks, particularly when the required changes involve non-trivial logical or structural modifications. These findings demonstrate both the difficulty of the problem and the potential of `JavaBackports` to stimulate new research directions in automated software maintenance and repair.

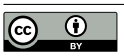
CCS Concepts

• **Software and its engineering → Software defect analysis**; **Software evolution**.

Keywords

software maintenance, patch backporting, datasets, mining software repositories

*corresponding author



MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793331

ACM Reference Format:
Kaushal Kahapola, Sharada Galappaththi, Dinith Ranasinghe, Ridwan Shariffdeen, Nisansa de Silva, Srinath Perera, and Sandareka Wickramanayake. 2026. `JavaBackports`: A Dataset for Benchmarking Automated Backporting in Java. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793331

1 Introduction

The maintenance of long-term support (LTS) software versions is a cornerstone of enterprise software engineering, ensuring stability and security for mission-critical systems [2, 11]. A central activity in this process is *patch backporting*, where security fixes and critical bug corrections are adapted from the main development branch to older, actively supported versions [11, 17]. Despite its importance, manual backporting remains highly challenging: it is time-intensive, cognitively demanding, and frequently neglected in practice [15, 19]. Empirical studies report that nearly 80% of security patches associated with Common Vulnerabilities and Exposures (CVEs) never reach all vulnerable branches, and the patches that are eventually backported exhibit an average delay of 40.46 days [15]. These delays create prolonged vulnerability exposure windows, underscoring the need for more reliable and efficient backporting mechanisms.

The Java ecosystem further amplifies these challenges due to its large dependency hierarchies, evolving APIs, and frequent refactoring, which introduce significant structural and semantic divergence across versions [2, 25]. As a result, backporting in Java becomes a complex semantic translation task that is insufficiently supported by existing program repair datasets, which generally do not capture cross-version transformations. Recent progress in Large Language Models (LLMs) for code comprehension and generation offers an opportunity to automate backporting [4, 7, 9, 23] as an Automated Program Repair (APR) task [10], but effective development of such tools requires high-quality, domain-specific datasets that reflect real-world backporting scenarios.

To address this need, we introduce **JavaBackports**, a curated benchmark dataset designed to support systematic research on automated backporting and cross-version semantic reasoning in Java. JavaBackports comprises of 491 manually validated backport instances drawn from 11,382 candidate commits across fifteen widely used open-source projects, including Elasticsearch, Hibernate, Druid, Kafka, Hadoop, Crate, HBase, Graylog, gRPC-Java, SQL, Logstash, Spring Framework, and four major JDK maintenance branches (jdk11, jdk17, jdk21, and jdk25). Each instance is annotated with metadata capturing its intent and complexity. Preliminary experiments with state-of-the-art LLMs, Gemini and GPT-4, reveal that automated backport generation remains a challenging task, particularly when patches require non-trivial logical or structural changes, highlighting the importance of JavaBackports in enabling future research. This paper makes the following contributions:

- A Java backport dataset named JavaBackports consisting of real-world Java backports across fifteen major OSS projects.
- A systematic, high-precision pipeline for identifying and validating backport instances.
- An analysis of types of backports in Java enterprise applications captured in JavaBackports.
- Preliminary results of the effectiveness of state-of-the-art LLMs to automated Java backporting.

2 Motivation

Maintaining LTS versions is critical for software stability and security, with patch backporting being a core maintenance activity [3, 11]. In large Java ecosystems such as Elasticsearch, Apache Kafka, and OpenJDK, delayed or incorrect backports can introduce significant operational and security risks [16]. Backporting is labour-intensive, requiring developers to reason about semantic and structural divergence between branches [3, 12], which slows patch propagation, extends vulnerability exposure [16], and leads to missed or faulty fixes [20]. Although LLMs have shown promise for automated software maintenance [1, 10], backporting remains challenging as a cross-version code translation task involving API drift, structural evolution, and behavioral compatibility [3, 18].

To illustrate these challenges, we evaluated two state-of-the-art LLMs, GPT-4.1 Mini and Gemini 2.0 Flash, in a zero-shot setting on a Type-II backport from OpenJDK 17[1]. Type-II backports require only positional adjustments without API or structural changes [11]. Both models were provided with the original patch and all target-branch context files. Despite this idealized setup, both generated non-compiling patches, omitting a required SET_BOUNDS import and a 16-line method declaration (Figure 1); GPT-4.1 Mini additionally misplaced new logic and failed to remove deprecated code. These results indicate that even minimal backporting cases exceed the reliability of current zero-shot LLM approaches and highlight the lack of suitable Java backporting benchmarks.

To address this gap, we introduce JavaBackports, a benchmark of **491** real-world Java backports annotated by intent and complexity. The dataset enables systematic evaluation of LLM-based cross-version translation, retrieval-augmented generation methods [21], and automated backporting tools.

[1] f00607a50acecb3bfd6dd961e4378b4e1697e1e7

Original Backport

```
47  +import static java...SET_BOUNDS;
48  import ...
445  + /**
446  + * Reapplies the size of th...
450  + */
451  + private static void
       resizeFSWindow(...) {
452  + ...
458  + }
459  +
460       @Override
461       public boolean
       isDisplayChangeSupported() {
462           return (isFullScr...

                  ...

472  - // resize the...
473  - // display mode
474  - Rectangle screen...
475  - w.setBounds(...);
477  - // Note:...
479  + // Note: th...
482           } else {
483               throw ...
531  @@-531,6 +544,10 @@public void
       displayChanged() { }
  ...
545  + Rectangle screen...
546  + resizeFSWindow(...);
547  +
548           // pass on to all ...
549           topLevels.notify...()
```

Gemini 2.0 Flash Output

```
467 +++.../Win32GraphicsDevice.java
468 @@ -467,13 +467,9 @@
469   WWindowPeer peer = AWTAccessor
        .getComponentAccessor().
        getPeer(w);
470   configDisplayMode(screen, peer
        , dm.getWidth(), dm.
        getHeight(),
471   dm.getBitDepth(), dm.
        getRefreshRate());
472 - // resize the fullscreen...
473 - // display mode
474 - Rectangle screenBounds =
        get...Bounds();
475 - w.setBounds(screenBounds.x...);
```

GPT-4.1 Mini Output

```
443 + private static void
        resizeFSWindow(final Window w,
        final Rectangle b) {
444 + ...
459 + }
460 +
461       @Override
462       public boolean
        isDisplayChangeSupported() {
463           return ...;
529 @@-529,6 +542,10 @@public void
        displayChanged() {
543           // pass on to all ...
544           topLevels.notify...();
```

Figure 1: Critical failure points in LLM-generated backports vs. ground truth patch for OpenJDK 17 commit[2].

3 Related Work

Automated patch porting and backporting have attracted increasing attention due to the need to maintain long-term branches and rapidly propagate security fixes. Existing datasets largely target C, C++, or PHP ecosystems. FixMorph [11] curates 350 Linux kernel backport pairs and 30 Common Vulnerabilities and Exposure (CVE)-related cases from a large-scale analysis of the Linux kernel.. SKYPORT [14] provides 155 security patches for PHP applications across ten CMS frameworks. PatchWeave [13] includes 24 cross-project patch transplantation tasks in C/C++ libraries. PPatHF [8] identifies 6,487 ported patches between Vim and Neovim and releases a validated subset of 310 pairs. Mystiqe [22] presents 5,977 patch pairs linked to 3,536 CVEs across 33 repositories, evaluated primarily at the function level. TSBPORT [24] introduces 1,815 Linux kernel security backports, while PORTGPT [6] consolidates prior datasets into 146 patch pairs across 34 programs in C, C++, and Go for LLM evaluation.

Despite this progress, existing datasets emphasize vulnerability fixes or cross-project transplantation and largely exclude Java, despite its widespread enterprise use and reliance on long-term support branches. JavaBackports addresses this gap by providing a manually validated dataset of Java backports annotated with intent and complexity to support fine-grained analysis and model development.

4 JavaBackports Dataset

This section presents the methodology we followed to curate a dataset of Java backports and provides a statistical overview of the dataset. JavaBackports contains 491 real-world backport instances, systematically selected and manually validated from fifteen widely used open-source Java projects. The dataset curation process is shown in Figure 2.

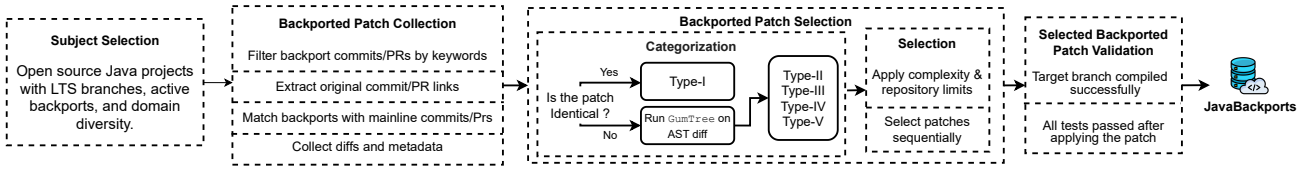


Figure 2: Process Diagram: Overview of the multi-stage pipeline for data extraction, stratification, and validation.

4.1 Methodology

Subject Selection: To ensure realism and relevance, we curated data from large-scale, open-source Java projects widely used in enterprise settings. Project selection followed three criteria: (i) enterprise relevance, targeting mature systems maintained by major organizations or foundations (e.g., Apache, Elastic, OpenJDK); (ii) the presence of multiple concurrently maintained long-term support (LTS) branches with established patch workflows; and (iii) diversity in application domain and maintenance context [3]. Overall, `JavaBackports` spans nine functional domains, including the Java Development Kit, distributed systems, search and analytics, and big data.

Candidate Backported Patch Collection: We mined the selected repositories using GitHub API between September and October 2025. To enable reproducible dataset construction, we developed a keyword-driven curation pipeline that identifies backporting instances. The pipeline leverages developer-provided metadata (commit messages and pull-request descriptions), which reliably captures developer intent [16]. To prioritize precision, we retain only commits explicitly containing backport-related terms (e.g., "backport", "cherry-pick"; Table 1), yielding high-precision candidates. The process consists of four steps:

(1) *Filtering:* Commit histories are scanned using predefined keywords and regex patterns.
(2) *Linking:* References to original commits are extracted from message text.
(3) *Pairing:* Each backport is matched to its corresponding mainline commit.
(4) *Extraction:* Code diffs, file paths, and branch metadata are collected for each extracted pair.

This step produced 11,382 candidate backported patches.

Table 1: Representative keyword patterns used for backported patch curation (analogous to bug-fix commit mining).

Pattern Category	Keyword / Regular Expression
Backport Terminology	`backport`, `port fix`, `down-port`
Version Control Actions	`cherry-pick`, `cherry pick`
Automated Tool Signatures	`(cherry picked from commit (\w+))`
Issue Tracking References	`backport for CVE-`, `fix for JIRA-`

Backported Patch Selection: To capture the spectrum of backporting difficulty, each candidate patch was automatically assigned to one of five types (Type I–V) following the taxonomy of Shariffdeen et al. [11]. Classification employed code-hunk analysis and Abstract Syntax Tree (AST) differencing using `GumTree` [5]; candidates with AST parsing failures were excluded. For each valid instance, we computed syntactic (line-level) and structural (AST-level) divergence to quantify adaptation effort. Candidates were ordered chronologically, prioritizing recent patches. The final 491 instances were selected using a per-repository, per-type cap of 50 to ensure balanced coverage and prevent dominance by high-volume projects. For higher-complexity types, all qualifying instances within the collection window were retained.

Selected Backported Patch Validation. All selected instances underwent rigorous verification using the latest build environment. A backport was retained only if it satisfied both of the following conditions:

(1) *Build Stability:* The target-branch codebase compiled successfully and passed all relevant test suites in both pre-patch and post-patch states.
(2) *Test-Level Correctness:* Patch-relevant tests either (i) passed when newly introduced by the backport, or (ii) transitioned from failing in the pre-patch state to passing in the post-patch state. For modified tests, changes were applied to the pre-patch version to ensure failures reflected the buggy state rather than test drift.

This validation ensures that all instances in `JavaBackports` are both syntactically valid and functionally correct within the target-version environment.

Intent Labeling: To improve dataset utility, we labelled each backport with the intent of the patch in addition to the type which represents the required adaptation effort. Adopting a closed coding scheme, each instance is assigned to one of five categories, Security Fix, Bug Fix, Feature Backport, Performance Improvement, or Refactoring, using direct keyword matching and semantic similarity analysis via Sentence Transformers on commit messages.

5 Overview of the `JavaBackports` dataset

Table 2 summarizes repository characteristics and the distribution of backport instances by patch type and intent in `JavaBackports`. The dataset contains 491 verified backport pairs collected from fifteen large-scale Java projects maintained by organizations such as Apache, Elastic, and OpenJDK, spanning domains including distributed systems, search and analytics, and multiple JDK release lines. All projects were analyzed over a recent two-year period (January 2024–January 2026) to capture contemporary backporting practices in the Java ecosystem.

Type I (47.5%) and Type II (30.8%) patches constitute the majority of instances, while Type V patches account for 15.7%. In terms of intent, Bug Fixes dominate (54.6%), followed by Security Fixes (30.1%)

Table 2: Overview of the JavaBackports dataset.

Repository Demographics				Patch Type Distribution					Intent Category					Total
Repository	Organization	Domain	Date Span	Type-I	Type-II	Type-III	Type-IV	Type-V	Security Fix	Bug Fix	Performance	Feature	Refactoring	
crate	crate	Distributed Data Store	2024/01–2026/01	50	50	5	1	21	30	84	4	8	1	127
druid	apache	Distributed Data Store	2024/02–2024/10	5	1	0	0	1	1	5	0	1	0	7
elastic	elastic	Search & Analytics	2024/08–2025/07	50	26	2	0	16	32	41	4	16	1	94
graylog	Graylog2	Log Management	2024/01–2025/11	12	5	1	0	3	1	11	0	7	2	21
grpc-java	grpc	Networking	2024/03–2025/12	23	3	0	1	1	7	13	0	8	0	28
hadoop	apache	Big Data	2024/01–2025/05	3	2	0	0	1	2	3	0	1	0	6
hbase	apache	Distributed Data Store	2024/02–2025/12	20	0	0	0	1	7	11	1	2	0	21
hibernate-orm	hibernate	Data Access	2025/05–2026/01	1	4	4	1	3	5	7	0	1	0	13
jdk11u-dev	openjdk	Java Development Kit	2024/01–2025/05	0	5	0	0	5	3	5	0	2	0	10
jdk17u-dev	openjdk	Java Development Kit	2024/01–2025/07	13	28	9	1	8	25	28	1	5	0	59
jdk21u-dev	openjdk	Java Development Kit	2024/01–2025/05	28	14	3	0	9	20	30	1	3	0	54
jdk25u-dev	openjdk	Java Development Kit	2025/06–2025/12	21	5	0	0	0	11	14	0	0	1	26
logstash	elastic	Data Ingestion	2025/01–2025/04	3	0	0	0	0	2	0	0	1	0	3
spring-framework	spring-projects	App Framework	2024/01–2025/06	4	4	0	0	1	2	7	0	0	0	9
sql	opensearch-project	Search & Analytics	2025/04–2025/11	0	4	2	0	7	0	9	0	4	0	13
Total				**233**	**151**	**26**	**4**	**77**	**148**	**268**	**11**	**59**	**5**	**491**
% of Total				47.5%	30.8%	5.3%	0.8%	15.7%	30.1%	54.6%	2.2%	12.0%	1.0%	100.0%

and Feature backports (12.0%). Overall, more than half of the dataset (52.5%, Types II–V) involves non-trivial adaptation, highlighting the prevalence of complex cross-version maintenance scenarios.

The predominance of Type I and Type II backports indicates that many maintenance tasks involve either direct patch reuse or minor positional adaptation. However, as shown in Section 6, current Large Language Models (LLMs) struggle even with these cases under naive prompting, suggesting that effective automation requires robust context alignment across versions. Moreover, the presence of higher-complexity patches underscores the need for advanced semantic reasoning to handle structural divergence in automated backporting systems.

6 Baseline Evaluation

To establish baseline performance on JavaBackports, we conducted a zero-shot patch generation study using Gemini 2.0 Flash and GPT-4.1 Mini. For each of the 491 verified instances, models were provided with the mainline patch and the corresponding target-branch source code and tasked with generating an equivalent backport. Performance was evaluated using three metrics:

(1) **Compilation Success (Comp.):** The number of generated patches that compile when applied to the target branch codebase.
(2) **Test Pass (Test):** The number of instances where the patched code passes the relevant test suite (Section 4.1).
(3) **Exact Match (EM):** The number of instances where the generated patch is byte-for-byte identical to the developer-authored ground truth backport.

Table 3 reports results stratified by backport difficulty. Both models achieve higher compilation rates for Type I and Type II patches, but test pass rates are consistently lower, indicating frequent semantic errors despite syntactic validity. GPT-4.1 Mini outperforms Gemini 2.0 Flash across all patch types and metrics. Performance degrades sharply for Type IV and Type V instances, which require structural or logical adaptation, and EM scores remain near zero across all types. These results highlight the difficulty of automated backporting and the limitations of current LLMs in zero-shot settings. It is worthy noticing that the input length was not a confounding factor: only two instances exceeded 300K tokens, well within the supported context windows of both models.

Table 3: Zero-Shot Performance on JavaBackports, stratified by Backport Difficulty Type. All metrics report the number of successful instances over the total count per type.

Type	Gemini 2.0 Flash			GPT-4.1 Mini		
	Compile	Tests Pass	Exact Match	Compile	Tests Pass	Exact Match
Type-I	153/233	136/233	33/233	188/233	174/233	39/233
Type-II	77/151	71/151	17/151	83/151	74/151	22/151
Type-III	11/26	8/26	0/26	14/26	11/26	2/26
Type-IV	1/4	1/4	0/4	2/4	1/4	0/4
Type-V	18/77	5/77	0/77	16/77	9/77	1/77

7 Conclusion

For decades, manual backporting has been a major bottleneck in Java enterprise software maintenance, yet the community has lacked a rigorously curated dataset to study and automate this task. We address this gap by introducing JavaBackports, a dataset of 491 manually verified, real-world Java backport instances from widely used enterprise projects. Using this dataset, we evaluated zero-shot automatic backport generation with two state-of-the-art LLMs, Gemini 2.0 Flash and GPT-4.1 Mini. Both models perform reliably only on trivial (Type I) cases, with performance degrading sharply for complex (Type IV and Type V) backports. These results demonstrate that JavaBackports captures challenging cross-version adaptation scenarios and exposes the limitations of current LLMs without task-specific training. As a publicly available benchmark, JavaBackports provides a foundation for advancing research on automated Java patch backporting.

Data Set: Our dataset can be accessed via GitHub from the following repository: https://github.com/Javabackports/javabackports

8 Disclaimer

The views and conclusions expressed in this paper are those of the authors alone and do not represent the official policies or endorsements of SonarSource or WSO2. Furthermore, the findings presented herein are independent and should not be interpreted as an evaluation of the quality of products at SonarSource or WSO2.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ekin D. Cubuk, Elad Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. (08 2021).

[2] Dipannita Chakroborti, K. A. Schneider, and Chanchal K. Roy. 2022. Backports: Change Types, Challenges and Strategies. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 636–647.

[3] Debasish Chakroborti, Kevin A. Schneider, and Chanchal K. Roy. 2022. Backports: Change Types, Challenges and Strategies. In *Proceedings of the 2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. IEEE, 636–647. doi:10.1145/3524610.3527920

[4] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*.

[6] Zhaoyang Li, Zheng Yu, Jingyi Song, Meng Xu, Yuxuan Luo, and Dongliang Mu. 2025. PortGPT: Towards Automated Backporting Using Large Language Models. *arXiv preprint arXiv:2510.22396* (2025).

[7] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint arXiv:2203.13474* (2022). doi:10.48550/arXiv.2203.13474

[8] 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.

[9] Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, Arman Szlam, Yossi Bakkal, Kevin Zhang, Lyanne Martinet, Machel Reid, Myle Ott, Edouard Grave, Thomas Scialom, and Allan Joulin. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023). doi:10.48550/arXiv.2308.12950

[10] Jayanath Senevirathna, Ayesh Vininda, Prasad Sandaruwan, Ridwan Shariffdeen, Sandareka Wickramanayake, and Nisansa de Silva. 2025. FusionRepair: Iterative Multi-Line APR via Fusion . In *2025 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, Los Alamitos, CA, USA, 27–34. doi:10.1109/APR66717.2025.00009

[11] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Sudipta Chattopadhyay, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, 633–645. doi:10.1145/3460319.3464821

[12] Ridwan Salihin Shariffdeen, Sudipta Chattopadhyay, Ming Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Trans. Softw. Eng. Methodol.* 30, 1 (Dec 2021). doi:10.1145/3412376

[13] 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.

[14] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches. In *31st USENIX Security Symposium (USENIX Security 22)*. 1993–2010.

[15] Xinchen Tan, Yulun Zhang, Jingwu Cao, Kangjie Sun, Mingwei Zhang, and Min Yang. 2022. Understanding the Practice of Security Patch Management Across Multiple Branches in OSS Projects. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*. Association for Computing Machinery, 767–777. doi:10.1145/3485447.3512236

[16] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the Practice of Security Patch Management across Multiple Branches in OSS Projects. In *Proceedings of the ACM Web Conference 2022*. Association for Computing Machinery, New York, NY, USA, 767–777. doi:10.1145/3485447.3512236

[17] Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 222–232.

[18] Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 222–232. doi:10.1109/ICSME.2016.71

[19] Yuan Tian. 2017. *Mining Software Repositories for Automatic Software Bug Management from Bug Triaging to Patch Backporting*. Ph. D. Dissertation. Singapore Management University. https://ink.library.smu.edu.sg/etd_coll_all/26

[20] Yuan Tian. 2017. *Mining Software Repositories for Automatic Software Bug Management from Bug Triaging to Patch Backporting*. PhD Thesis. Singapore Management University. https://ink.library.smu.edu.sg/etd_coll_all/26

[21] Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 146–158. doi:10.1145/3611643.3616256

[22] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Chen, Zhuotong Zhou, Yiheng Huang, JunPeng Zhao, and Xin Peng. 2025. Mystique: Automated Vulnerability Patch Porting with Semantic and Syntactic-Enhanced LLM. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 130–152.

[23] Cuiyun S. Xia, Yang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494.

[24] 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.

[25] Hao Zhong, Li Zhang, Tao Xie, and Hong Mei. 2020. An Empirical Study on API Parameter Rules. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE, 899–911. doi:10.1145/3377811.3380391