Refactoring-Aware Patch Integration Across Structurally Divergent Java Forks

Daniel Ogenrwot
Department of Computer Science
Howard R. Hughes College of Engineering
University of Nevada Las Vegas
Las Vegas, USA
ogenrwot@unlv.nevada.edu

John Businge
Department of Computer Science
Howard R. Hughes College of Engineering
University of Nevada Las Vegas
Las Vegas, USA
john.businge@unlv.edu

Abstract—While most forks on platforms like GitHub are short-lived and used for social collaboration, a smaller but impactful subset evolve into long-lived forks, referred to here as variants, that maintain independent development trajectories. Integrating bug-fix patches across such divergent variants poses challenges due to structural drift, including refactorings that rename, relocate, or reorganize code elements and obscure semantic correspondence. This paper presents an empirical study of patch integration failures in 14 divergent pair of variants and introduces `RePatch`, a refactoring-aware integration system for Java repositories. `RePatch` extends the RefMerge framework, originally designed for symmetric merges, by supporting asymmetric patch transfer. `RePatch` inverts refactorings in both the source and target to realign the patch context, applies the patch, and replays the transformations to preserve the intent of the variant. In our evaluation of 478 bug-fix pull requests, Git `cherry-pick` fails in 64.4% of cases due to structural misalignments, while `RePatch` successfully integrates 52.8% of the previously failing patches. These results highlight the limitations of syntax-based tools and the need for semantic reasoning in variant-aware patch propagation.

Index Terms—Patch Integration, Refactoring-aware Tools, Software Variants, Cherry-pick Failure, Structural Divergence, Semantic Conflict Resolution

I. INTRODUCTION

Many modern software systems are derived by copying and adapting existing codebases, a practice known as forking [1]–[9]. Although many of such forks are short-lived or used for temporary collaboration, a smaller but impactful subset develops into long-lived forks, referred to in this work as *variants* [2]–[5], [10]–[12], that maintain independent development trajectories. These variants enable customization and organizational control [3], [5], [13], but the structural and semantic divergence over time complicates the reuse of upstream patches such as bug fixes or enhancements.

Structural and semantic divergence poses a major barrier to patch propagation. As variants undergo refactoring, such as method renaming, class movement, or interface modification [14]–[16], syntactic similarity with the source deteriorates. Tools like Git `cherry-pick`, which rely on textual alignment, often fail when entities have been reorganized or renamed [15], [16]. Even minor structural edits can cause merge conflicts or incorrect applications, requiring developers

to manually identify equivalent code regions, which is a tedious and error-prone task [5], [17]. Although LLM-based techniques like Pan et al. [16] offer a promising direction for patch adaptation, they provide limited transparency in how structural edits are resolved. Likewise, ML-based merge tools such as `DeepMerge` [18] and `MergeBERT` [19] frame merge resolution as token classification or sequence generation tasks over textual representations of code. Although they achieve strong results on benchmark datasets, their core models do not explicitly reason about structural drift or semantic refactorings, instead relying on learned token-level patterns. In addition, these models offer limited interpretability as their predictions lack insight into how structural changes influence merge outcomes.

Empirical studies highlight the practical challenges of patch integration. Businge et al. [3] analyzed over 9,000 *software families*[1] and found that integration between forked variants was rare, revealing the lack of effective patch propagation mechanisms. Ramkisoen et al. [5] reported over 1,000 cases each of "missed opportunities" (patches present in the source but missing in the target variant), and "effort duplication" (manual re-implementation) across 364 variant pairs. Many missed patches lagged behind by a median of 52 weeks, showing the fragility of patch reuse in clone-and-own ecosystems. Refactoring-aware tools like `MolhadoRef` [14], `RefMerge` [15], and `IntelliMerge` [20] have reduced merge conflicts in symmetric divergence scenarios where branches share recent history. However, they are not designed for asymmetric divergence, where forks evolve independently and accumulate unsynchronized structural changes. Therefore, these tools do not address the challenge of patch transfer across divergent repositories.

To address these limitations, we investigate patch integration across structurally divergent software variants. These variants share a common origin, but have evolved independently, introducing structural changes that hinder patch reuse. Rather than merging entire branches, we focus on applying individual bug-fix commits previously identified as relevant but not

---

[1]A software family refers to a group of variants that originate from a common codebase and evolve independently over time.

integrated into target variants [5]. We examine how method-level refactorings affect patch transferability and present a refactoring-aware strategy that enhances Git's `cherry-pick` with semantic reasoning. Our approach builds on the invert-and-replay strategy of Ellis et al. [15], extending it to support integration between long-diverged repositories. Using `RefactoringMiner` [21], we detect and invert refactorings in both source and target variants to enable patch application, then replay them to preserve structural intent. This supports a systematic assessment of how refactorings affect patch integration success across real-world divergent variants.

We evaluate our approach on 478 bug-fixing patches from the `PaReco` dataset [5], which catalogs missed integration opportunities across divergent variant pairs. Each case uses the merge commit from the original pull request as the integration unit. Our evaluation spans 14 variant pairs. While Git `cherry-pick` fails on 64.6% of these due to structural conflicts, our method successfully integrates 52.8% of the previously failing patches, substantially reducing manual effort. By explicitly modeling the structural changes that cause cherry-pick failures, our approach enables interpretable and reliable patch propagation across variant-rich ecosystems. To our knowledge, this is the first empirical study to isolate refactorings as a root cause of integration failure and assess their effect on patch reuse.

Contributions. This paper makes the following contributions:

- We frame patch integration between long-diverged forks as a refactoring-aware cherry-pick problem and analyze the structural barriers to reuse across independent repositories.
- We present `RePatch`, a rule-based integration system that extends `RefMerge` to asymmetric settings by performing dual-sided refactoring detection, inversion, semantic patch application, and transformation replay.
- Our study of 478 bug-fix pull requests across 14 Java variant pairs reveals that 91.6% of Git cherry-pick failures are due to target-side refactorings. `RePatch` resolves 52.8% of previously failed cases and reduces conflicts in over half of them, while introducing new conflicts in less than 4%.
- We release all tools, datasets, patch traces, and integration logs to support replication and future work on variant-aware patch propagation [22].

II. TERMINOLOGY, PROBLEM AND CONCRETE EXAMPLES

In this section, we formally define the terminology used throughout our study and illustrate the practical challenges of patch integration in divergent forks through a concrete example.

A. Terminology for Variant-Based Patch Integration

- **source variant.** The repository where a patch originates, typically where the bug fix was authored and merged.
- **target variant.** The repository receiving the patch. Structural divergence may prevent direct application.
- **hunk.** A contiguous block of line-level changes in a diff, marked by `@@ -s,n +s,n @@` [23]–[25].

- **patch.** A set of hunks spanning one or more files, representing all code changes in a pull request for a bug fix.
- **refactoring.** A behavior-preserving structural transformation (e.g., Rename Method, Move Class) that may obscure traceability between variants.
- **symmetric divergence.** Two branches evolve from a recent common ancestor with aligned histories, allowing bidirectional reasoning using standard merge tools [15].
- **asymmetric divergence.** Variants evolve independently from a shared base, accumulating unsynchronized commits. Lacks a recent common ancestor, requiring semantic reasoning for patch transfer.

B. Motivating Problem Scenario

Figure 1 shows patch integration between two variants that initially share a synchronized history but diverge at a specific point (`divergence_date`). After this point, both variants evolve independently and no longer share synchronized commits. Suppose a developer identifies and fixes a bug in `Foo.java`, a file present in both source and target variants, by submitting a pull request in the source repository. Although the same bug likely exists in the target variant, there is no direct mechanism to transfer the patch. Developers often use Git's `cherry-pick` to apply the source commit to the target's `git_head`, with two possible outcomes:

(i) The patch applies cleanly if the change involves simple textual edits.

(ii) The patch fails if structural changes in the target, such as refactorings or reorganizations, conflict with the assumptions of the patch.

Even small independent edits can cause patch failures due to mismatches in method signatures, class structures, or identifier names. Success often depends on the structural consistency between the merge commit in the source and the current state of the target. When alignment exists, patch transfer may succeed with minimal effort. Otherwise, developers must manually resolve discrepancies. Addressing this requires detecting and aligning structural edits such as refactorings. The integration in Figure 1 is bidirectional; either variant may produce reusable patches.

C. Patch Integration Examples: Success and Failure

To illustrate these challenges, we present examples of successful and failed patch integrations from `apache/kafka` to its industrial variant `linkedin/kafka`. We begin with a success case that demonstrates how structural consistency enables Git `cherry-pick` to apply a patch cleanly. This is followed by a failure case where divergence obstructs integration.

Example 1: Success Case. We begin with a scenario where a source patch integrates cleanly into the target variant using Git's native `cherry-pick`, without requiring semantic reasoning. In pull request KAFKA-14540 [26], the source repository fixes a bug in the `writeByteBuffer` method of the `DataOutputStreamWritable` class. The bug involved incorrectly passing `buf.limit()` instead
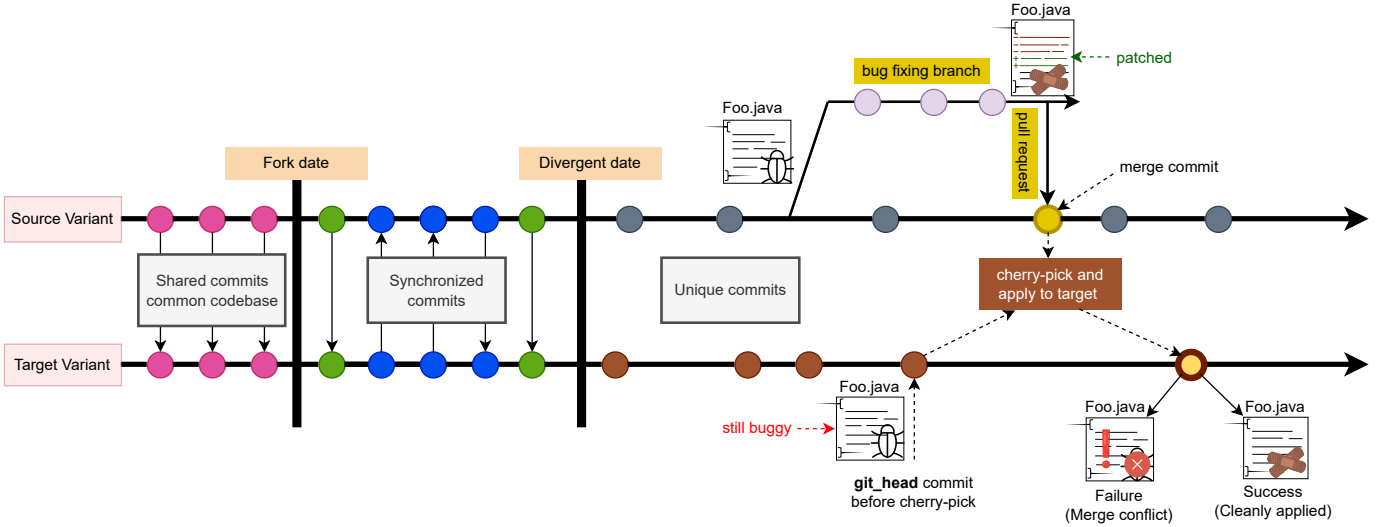
Fig. 1: Illustration of patch integration from source to target variant.

of `buf.remaining()` as the length argument in the `out.write(...)` call, which could lead to data corruption when writing `ByteBuffer` content to an output stream.

Listing 1: Diff file for PR–13032 from the source repository

```
1  @@ -99,7 +99,7 @@ public void writeUnsignedVarint(int
       i) {
2      public void writeByteBuffer(ByteBuffer buf) {
3          try {
4              if (buf.hasArray()) {
5  -                out.write(buf.array(), buf.position()
     , buf.limit());
6  +                out.write(buf.array(), buf.
     arrayOffset() + buf.position(), buf.remaining());
7              } else {
8                  byte[] bytes = Utils.toArray(buf);
9                  out.write(bytes);
```

The fix updated the call to correctly compute the offset and remaining length using `buf.arrayOffset() + buf.position()` and `buf.remaining()`, and was applied cleanly to the target variant. This successful cherry-pick was enabled by the structural consistency of the file and method across both source (Listing 1) and the `git_head` of the target variant (file: DataOutputStreamWritable.java, lines 99–110)[2]. The patch was confined to a single method and introduced no conflicting edits, allowing Git to apply it without developer intervention (Listing 1, line 6). This example shows that when source and target variants remain structurally aligned, `cherry-pick` can be a viable patch integration strategy. The next example illustrates a failure case, where semantically meaningful but structurally disruptive edits prevent clean application.

Example 2: Failure case.

Pull request #12363 [27] in `apache/kafka` addresses a parameter misordering bug and improves metric naming and log reporting.[3] Although the changes are functionally straightforward, Git's `cherry-pick` fails

to apply the patch cleanly to the `linkedin/kafka` variant due to structural divergence. The patch fails with 12 merge conflicts across 79 lines in five files: `KafkaStreams.java`, `PartitionGroup.java`, `PartitionGroupTest.java`, `StreamThread.java`, and `StreamTask.java`.[4] For example, a semantic renaming of the metric field `totalBytesSensor` to `totalInputBufferBytesSensor` in `PartitionGroup.java` and `StreamTask.java` causes Git to fail to match equivalent logic. Additionally, the patch performs an inlining refactoring in `StreamTask.java`, replacing the separately declared variable `enforcedProcessingSensor` with a direct call to `TaskMetrics.enforcedProcessingSensor(...)` within the constructor's argument list (see Line 186 in the patch[2]). The modifications are shown below in Listing 2.

Listing 2: Refactoring mismatch in `StreamTask.java`

```
// source:
partitionGroup = new PartitionGroup(...,
    TaskMetrics.enforcedProcessingSensor(...), ...);
// target:
final Sensor enforcedProcessingSensor;
enforcedProcessingSensor = TaskMetrics.
    enforcedProcessingSensor(...);
partitionGroup = new PartitionGroup(...,
    enforcedProcessingSensor, ...);
```

This example highlights how semantically meaningful edits (such as renaming and inlining) can break textual alignment and lead to patch integration failure across variants. A manual reconstruction suggests that a refactoring-aware tool could reconcile such conflicts in at least 4 of 5 affected files (80%), reducing conflicting lines from 79 to approximately 20. However, manual resolution does not scale for large or complex patches, which underscores the need for automated support.

---

[2]File: `DataOutputStreamWritable.java`, lines 99–110
[3]Patch details: https://github.com/apache/kafka/pull/12363/files

[4]These represent content-level conflicts, not file-level.

III. STUDY DESIGN

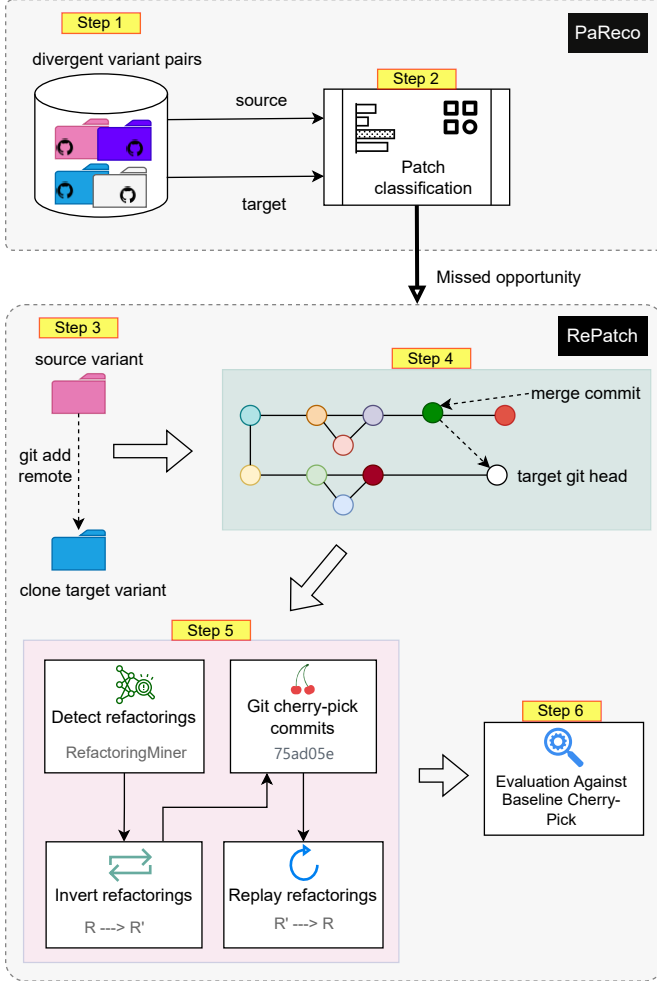This section presents the research method from data collection, tool design, and experiment setup.

Fig. 2: Overview of the research method pipeline.

A. Research Questions

- **RQ1:** *How often do source variant bug-fix patches fail to apply cleanly to target variants using Git's cherry-pick?* This question establishes a baseline by quantifying the limitations of Git's patch integration in the presence of structural divergence between variants. It highlights the practical challenges of cross-variant patch transfer and motivates the need for refactoring-aware support.

- **RQ2:** *What proportion of cherry-pick failures are attributable to refactoring operations (e.g., method / class renaming or moving)?* This question quantifies how often semantic changes, such as method or class renaming and movement, are responsible for patch integration failures. It distinguishes refactoring-related failures from those caused by deletions, interface drift, or unrelated edits, and motivates the need for refactoring-aware integration techniques.

- **RQ3:** *Can a refactoring-aware integration approach reduce merge conflicts and increase the success rate of applying patches across divergent variants?* This question assesses whether our method outperforms Git's cherry-pick by reducing conflicts and increasing success rates. We investigate if incorporating refactoring awareness enables better integration outcomes, regardless of whether the patch originates from the source or target variant.

B. Data Collection

This section describes the steps we followed to collect the data used in our study. A visual summary of the process is presented in Figure 2.

Step 1: Selection of divergent variant pairs: We base our study on the dataset created by Ramkisoen et al. [5], which includes 364 GitHub variant pairs identified as structurally divergent. In the original study, projects were considered active if they had commit activity within the six months leading up to August 6, 2021, which marked the end of their data collection period. In our study, we reuse this dataset but apply a new activity filter to ensure that the projects are still relevant. Specifically, we retain only the variant pairs that show ongoing development activity on May 15, 2025. From this filtered set, we further restrict our selection to Java-based pairs since our refactoring-aware integration approach is implemented for Java projects. This results in a curated set of active and structurally divergent Java variant pairs for evaluation.

Table I presents the metadata of the 14 divergent variant pairs selected for this study. Each pair consists of a source repository and a structurally divergent target variant. For each pair, we report the number bug fixing pull requests, *Missed Opportunity* patches identified using the PaReco tool, as well as the number of commits the target is ahead and behind relative to the source. We also include popularity metrics (stargazers and forks) for each source repository to highlight their relevance and community engagement. The *Divergent Date* indicates the estimated point at which the two repositories began to evolve independently, calculated based on the last shared commit in their histories. Since patch integration can be attempted in both directions (i.e., source→target and vice versa), we treat each directional pair independently, resulting in a total of 14 divergent variant pairs for analysis. This subset provides realistic cases of structural divergence and refactoring evolution suitable for evaluating semantic patch integration across forks.

Step 2: Patch Identification. We use the PaReco tool [5] to identify bug-fix patches suitable for integration across structurally divergent Java-based variant pairs. PaReco is a token-based clone detection tool extended to analyze both modified and unmodified code regions between divergent variants. Specifically, it focuses on pull requests that contain bug-fix commits and classifies each patch based on whether its changes have been fully, partially, or not at all propagated to the target variant. This classification is determined by examining whether the modified code fragments, known as *hunks*, from the source are present in the corresponding files

in the target. Among these, we focus exclusively on *Missed Opportunities*, defined as patches where at least one modified hunk is absent from the target variant. This includes both entirely unpropagated patches and partially applied ones that still exhibit semantic divergence. These cases offer a realistic and challenging context for evaluating the effectiveness of refactoring-aware integration techniques.

To ensure that selected patches reflect current and actionable integration scenarios, we rerun `PaReco` on each divergent variant pair using commits made from the established divergence point through May 15, 2025. For each missed opportunity identified, we extract the corresponding `merge commit` from the source repository bugfix pull request as the integration unit. The `PaReco` classification has been empirically validated with a precision of 91%, recall of 80%, accuracy of 88%, and an F1 score of 85%, confirming its reliability in identifying integration gaps where structural differences and refactorings can prevent direct reuse of upstream patches.

C. Patch Integration

We employ `RePatch`, a refactoring-aware patch integration system that extends `RefMerge` [15]. While `RefMerge` operates at the branch level and performs full-tree merges, `RePatch` supports commit-level cherry-pick workflows suited for structurally divergent variants. `RePatch` adapts `RefMerge`'s core mechanisms of refactoring inversion and replay for localized, commit-scoped integration. It detects and inverts refactorings on both source and target variants, generates a clean diff from the base of the source commit, and applies it to the inverted target. Refactorings from both sides are then replayed to restore the intended structure. To ensure semantic correctness, `RePatch` reuses `RefMerge`'s conflict matrix, a data structure that captures ordering dependencies and replay constraints, but reconfigures it for commit-level patching. Unlike `RefMerge`, which reasons over entire branch histories, `RePatch` applies the matrix at finer granularity, allowing it to detect and resolve structural clashes that arise during cherry-pick. This design preserves patch semantics and variant integrity while extending refactoring-aware integration to real-world cross-repository workflows.

Step 3: Link Target and Source Repository. To facilitate patch transfer, `RePatch` begins by cloning the target repository. It then configures the source repository as a remote within the target's Git environment. This linkage is established using the `git remote add` and `git fetch` commands, which import the source's commit history into the local Git namespace. This setup allows us to reference and operate on source commits directly within the integration context, thereby enabling cherry-pick-based patch transfer between structurally divergent repositories.

```
git clone https://github.com/linkedin/kafka # target
git remote add kafka https://github.com/apache/kafka #
    ↪ source
git fetch kafka
```

Step 4: Apply Bug-Fix Commits via Cherry-Pick. Once the repositories are linked, we identify the relevant bug-fix

commits corresponding to *Missed Opportunities*, previously extracted as `merge commits` from source variants' pull requests. These commits encapsulate fixes that are present in the source variant but missing in the target. To enable fine-grained, commit-level integration, we adopt Git's `cherry-pick` operation as the core mechanism of our pipeline. Unlike `git merge`, which combines entire branch histories and assumes shared ancestry, `cherry-pick` allows isolated commits to be transferred across divergent variants. This makes it more appropriate for integrating individual patches without relying on symmetric version histories. Other tools like `git apply`, which operate directly on patch files, lack support for commit metadata, and do not integrate well with Git's conflict resolution mechanisms.

In contrast, `cherry-pick` offers a more reproducible and semantically meaningful integration process. For each *Missed Opportunity* case, we apply the corresponding merge commit to the target variant using Git's `cherry-pick --no-commit` flag to allow for conflict handling and further transformation if needed.

```
git cherry-pick <commit-hash> --no-commit
```

For each `cherry-pick` attempt, we automatically record whether the operation succeeds or fails. Failures are defined by the presence of Git-reported merge conflicts that prevent the patch from being applied cleanly. To ensure reproducibility, each `cherry-pick` is performed on a fresh checkout of the target variant at the same commit immediately prior to the attempted integration (i.e., the `git_head` of the target variant). If the `cherry-pick` completes without conflicts, the bug fix is integrated seamlessly into the target repository. However, due to structural drift often caused by independent refactorings, class reorganizations, or identifier renaming, many such attempts result in merge conflicts. These failures signal the inadequacy of traditional syntax-based patch application in variant-aware development settings and motivate the need for semantic-aware resolution strategies, as explored in the subsequent steps of our methodology.

Step 5: Refactoring-Aware Conflict Resolution. For commits that fail to apply cleanly due to merge conflicts, we employ a refactoring-aware resolution strategy. In many cases, integration failures stem from semantic discrepancies such as method renaming, relocation, or parameter reordering that introduce structural misalignment between the source and target repositories. To address these conflicts, we use `RefactoringMiner` [21] to detect structural refactorings in both the `cherry-picked` commit and the current `git_head` of the target repository (See Figure 1). The detected refactorings are classified and processed by a rule-based algorithm implemented in `RePatch`. The system automatically determines whether a failed integration is refactoring-related by applying deterministic matching rules (e.g., identifier correspondence, method relocation mapping) without subjective human judgment. These refactorings are then temporarily inverted to align the two repositories structurally, approximating the state they would have shared at the point

TABLE I: Metadata of source and target variant Pairs used in this study.

Source	Stars	Forks	Target	Ahead[1]	Behind[2]	Diverged On	PRs	MOs
apache/kafka [28]	30.1k	14.4k	linkedin/kafka	470	6,332	2022-06-02	1,247	393
DSpace/DSpace [29]	968	1.3k	ufal/clarin-dspace	1,902	13,418	2020-08-07	465	53
bitcoinj/bitcoinj [30]	5.1k	2.5k	bisq-network/bitcoinj	229	2,126	2019-03-01	13	6
bitcoinj/bitcoinj [30]	5.1k	2.5k	langerhans/dogecoinj-new	143	1,841	2019-03-01	13	6
javacc/javacc [31]	1.2k	251	tulipcc/ParserGeneratorCC	414	592	2018-11-20	40	2
xerial/sqlite-jdbc [32]	3.0k	641	Willena/sqlite-jdbc-crypt	362	16	2020-12-30	35	1
typetools/checker-framework [33]	373	1.1k	eisop/checker-framework	1,634	2,566	2024-09-05	40	5
Source → Target Interchanged								
linkedin/kafka [34]	147	58	apache/kafka	6,332	470	2022-06-02	118	7
ufal/clarin-dspace [35]	28	19	DSpace/DSpace	13,418	1,902	2020-08-07	12	1
bisq-network/bitcoinj [36]	14	28	bitcoinj/bitcoinj	2,126	229	2019-03-01	2	0
langerhans/dogecoinj-new [37]	58	41	bitcoinj/bitcoinj	1,841	143	2019-03-01	0	0
tulipcc/ParserGeneratorCC [38]	10	7	javacc/javacc	592	414	2018-11-20	1	0
Willena/sqlite-jdbc-crypt [39]	190	36	xerial/sqlite-jdbc	16	362	2020-12-30	7	1
eisop/checker-framework [40]	22	24	typetools/checker-framework	2,566	1,634	2024-09-05	40	3
						Total	**2,033**	**478**

[1] Commits present in the target repository but absent in the source.
[2] Commits present in the source repository but absent in the target.

of divergence. Simple cases such as *Rename Method* or *Move Method* are handled by direct identifier substitution and class relocation. For example, if a method has been renamed in the target repository from `m1` to `m2 → {RenameMethod (m1, m2)}`, we temporarily invert this transformation by applying `{RenameMethod (m2, m1)}`. This reverts the method to its original identifier, thereby aligning the naming convention with the source patch to facilitate successful integration. For more complex cases, such as *Extract Method*, RePatch leverages IntelliJ IDEA's PSI-based refactoring APIs to perform semantic inlining. Specifically, the system identifies the extracted method in the PSI tree, locates all invocation sites in the original calling method, determines the surrounding statement context, and then programmatically applies IntelliJ's `InlineMethodProcessor` to revert the extraction (more details are povided in the `RefMerge` study [15]). This ensures that the original code structure matches the source patch context before reattempting integration. Once the repositories are structurally aligned, we reapply the patch using the procedure described in **Step 4**. If the patch is successfully integrated without manual intervention, we then replay the previously inverted refactorings to restore the intended evolution history of the target repository. This two-phase process enables the integration of semantic patches even in the presence of substantial structural divergence.

Step 6: Evaluation Against Baseline Cherry-Pick. To assess the effectiveness of our refactoring-aware integration strategy, we compare each `Repatch` attempt to a baseline `Git cherry-pick` performed without any structural alignment. This comparative analysis enables us to quantify (i) the proportion of cherry-pick failures attributable to refactorings, and (ii) the extent to which our method successfully recovers failed integrations via temporary inversion and replay. These insights are critical for evaluating the practical utility of semantic awareness in automated patch transfer.

IV. RESULTS & DISCUSSION

A. *RQ1: How often do source variant bug-fix patches fail to apply cleanly to target variants using Git's cherry-pick?*

To answer RQ1, we quantify the baseline failure rate of Git's native `cherry-pick` operation when applied to divergent variants. As detailed in Section III, each patch for the source variant is cherry picked into the target variant using Git's default `cherry-pick` command without any semantic augmentation. When Git reports a merge conflict, we classify the attempt as "Failed"; otherwise, it is recorded as "Passed." Of the 478 cherry-pick attempts, 309 patches failed due to merge conflicts, resulting in an overall failure rate of 64.4%. Table II presents the detailed success and failure counts across all target variants. Notably, the failure rates exhibit substantial variation across projects. For instance, `clarin-dspace` experienced a failure rate of 94.3%, and `linkedin/kafka` failed to apply 59.0% of upstream patches. In contrast, other projects such as `dogecoinj-new`, `ParserGeneratorCC`, and `sqlite-jdbc-crypt` showed no cherry-pick failures. To summarize the distribution across all projects, the **median failure rate is 94.3%**, indicating that for most variants, Git's default `cherry-pick` mechanism struggles to apply source patches cleanly. This variability reflects differing levels of structural and semantic divergence. High failure rates, particularly in `clarin-dspace` and `linkedin/kafka`, suggest significant architectural drift or refactoring activity, which disrupts the assumptions of textual equivalence required by line-based patching tools. These observations highlight the fragility of syntax-based patch propagation techniques such as cherry-pick and diff-based merging, which assume structural

consistency between repositories. In fork-based development where semantic and architectural drift is common, these traditional approaches often fail, underscoring the need for refactoring-aware integration strategies.

TABLE II: Cherry-Pick Success and Failure Rates Across Target Variants

Target Variant	MOs	Passed	Failed
linkedin/kafka	393	161 (41.0%)	232 (59.0%)
apache/kafka	7	1 (14.3%)	6 (85.7%)
ufal/clarin-dspace	53	3 (5.7%)	50 (94.3%)
DSpace/DSpace	1	0 (0.0%)	1 (100%)
bisq-network/bitcoinj	6	0 (0.0%)	6 (100%)
bitcoinj/bitcoinj	–	–	–
langerhans/dogecoinj-new	6	0 (0.0%)	6 (100%)
bitcoinj/bitcoinj	–	–	–
eisop/checker-framework	5	0 (0.0%)	5 (100%)
typetools/checker-framework	3	1 (33.3%)	2 (66.7%)
tulipcc/ParserGeneratorCC	2	2 (100%)	0 (0.0%)
javacc/javacc	–	–	–
Willena/sqlite-jdbc-crypt	1	1 (100%)	0 (0.0%)
xerial/sqlite-jdbc	1	0 (0.0%)	1 (100%)
TOTAL	**478**	**169**	**309**

Discussion of Findings. The cherry-pick failure rate observed in our study underscores the structural fragility of patch propagation in divergent software variant ecosystems. Although the overall failure rate is 64.4%, the median failure rate across target variants is even higher at 85.7%, with several projects including `clarin-dspace`, `checker-framework`, and `bitcoinj` variants exhibiting failure rates between 94% and 100%. These results signal a fundamental barrier to automated patch reuse when repositories evolve independently through uncoordinated changes such as refactorings, code reorganizations, and identifier renamings. Even when the functional intent of a patch remains valid, Git's syntax-driven `cherry-pick` often fails to map semantically equivalent constructs, leading to spurious conflicts or rejection of the patch.

This contrasts with symmetric development settings. For example, the `RefMerge` study [15] reports a median merge conflict rate of 16% across 20 open-source projects using Git `merge`. While `merge` and `cherry-pick` serve different roles, the comparison remains informative. Git `merges` typically operate between branches with a recent common ancestor and a largely aligned structure, allowing line-based heuristics to function with relatively low conflict rates. In contrast, cherry-pick operations across asymmetrically diverged variants must contend with semantic and architectural drift, including refactorings, identifier renamings, and code reorganizations. Even when a patch's intent remains valid, structural misalignments can prevent Git from applying it cleanly. These findings highlight the limitations of syntax-based tools in

variant-to-variant patch propagation and motivate the need for refactoring-aware integration strategies that reason beyond line-level edits. In the following sections (RQ2 and RQ3), we investigate the root causes of patch failures and introduce `RePatch`, a system designed to improve patch transfer accuracy by reconciling structural divergence and semantic intent across independently evolving variants.

> ⚲ **Summary of RQ1 Results**: *The results of RQ1 demonstrate that traditional syntax-based patch integration mechanisms, such as Git's `cherry-pick`, are inadequate for reliably propagating bug-fix patches across structurally diverged software variants. High failure rates across multiple projects reflect the compounding effects of semantic drift, architectural changes, and uncoordinated evolution. These findings establish a clear need for integration techniques that move beyond textual similarity and account for structural and semantic alignment. RQ2 and RQ3 build on this foundation by examining the nature of these failures and evaluating a refactoring-aware system designed to improve patch integration in divergent variant ecosystems.*

B. RQ2: What proportion of cherry-pick failures are attributable to refactoring operations (e.g., method / class renaming or moving)?

To investigate the influence of refactorings on patch integration failures, we analyze all cherry-pick failures identified in RQ1 to determine whether these failures are semantically linked to structural changes in the target repository. For each failing patch, we apply `RefactoringMiner` [21] to detect refactorings in both the pull request merge commit from the source and the `git_head` of the target variant. While `RefactoringMiner` supports over 100 refactoring types, we focus our analysis on 17 structurally impactful refactorings that are supported by `RefMerge` [15]. These include: *RenameMethod*, *MoveMethod*, *Move&RenameMethod*, *RenameClass*, *MoveClass*, *Move&RenameClass*, *InlineMethod*, *ExtractMethod*, *PullUpMethod*, *PushDownMethod*, *RenameField*, *MoveField*, *Move&RenameField*, *PullUpField*, *PushDownField*, *RenamePackage*, and *RenameParameter*. A cherry-pick failure is labeled as *refactoring-related* if the conflicting patch semantically overlaps with any refactored entity in the target codebase. This classification is based on syntactic references (e.g., method or class names) and structural transformations in the affected region.

Figure 3 presents the five main types of refactoring that most frequently contributed to cherrypick failures between divergent variants. To ensure a meaningful interpretation, we restricted the analysis to target repositories that exhibit at least one instance of supported refactoring. The figure aggregates the number of refactoring-related conflicts per type, color-coded by project. While the figure shows the raw number of implicated refactorings, we also computed relative frequencies to identify which types are most associated with failure. The results indicate that *RenameMethod*

(53.1%), *RenameParameter (41.1%)*, and *MoveClass (3.5%)* are the most dominant contributors to conflict, especially in `linkedin/kafka`, `apache/kafka`, `clarin-dspace`, and `typetools/checker-framework` target variants. These findings emphasize how even conceptually simple but semantically disruptive refactorings can critically undermine the applicability of patches, reinforcing the necessity for refactoring-aware strategies in modern patch management workflows.
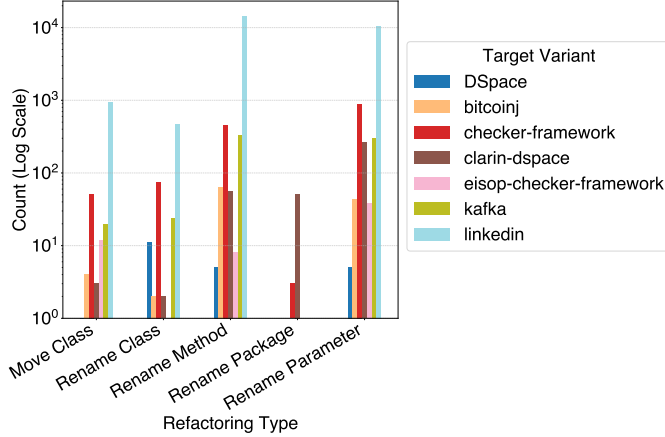

Fig. 3: Top 5 refactoring types associated with cherry-pick failures, grouped by refactoring type (X-axis) and target variant (legend). Y-axis uses a log scale to normalize project size variability and improve comparability across targets.

Among the 309 cherry-pick failures identified in RQ1, 283 (**91.6%**) were attributable to structural refactorings in the target variant. This aligns with our expectations for divergent variants, where independent evolution introduces widespread renamings, relocations, and method restructuring. In particular, projects with high refactoring activity exhibited the highest proportion of refactoring-induced conflicts. For instance, `linkedin/kafka` exhibited an 88.5% refactoring-induced failure rate. These findings reinforce the importance of semantic traceability in maintaining long-lived forks.

Discussion and Implications: These findings confirm that refactorings are a dominant factor behind patch integration failures in divergent forks. In most cases, the source patch references methods, classes, or identifiers that have been renamed, moved, or structurally transformed in the target, making them unresolvable through Git's syntax-based `cherry-pick`. This limitation underscores the broader need for integration tools that incorporate semantic reasoning, especially when structural correspondence between variants has eroded over time. Our analysis shows that even seemingly simple transformations, such as method renaming or parameter reordering, can derail patch transfer when semantic alignment is not maintained.

Prior work such as `RefMerge` [15] demonstrated that refactoring-aware merging can significantly reduce conflicts in symmetric development scenarios, successfully resolving or reducing conflicts in 25% of merge cases. Our study builds on this insight by demonstrating the heightened importance of refactoring-awareness in asymmetric settings, where patches from a stable source must be applied to independently evolving targets. In this context, 91.6% of patch integration failures were attributable to structural refactorings that disrupt identifier and context alignment across source and target. This sharp contrast highlights the structural fragility of fork-based development and the limitations of syntax-driven heuristics in the presence of long-term architectural drift. These results motivate the elevation of refactoring detection and traceability as first-class capabilities in cross-variant patch integration workflows, where accurate semantic mapping is essential for bridging the gap between source intent and target structure.

♀ Summary of RQ2 Results: *We find that **91.6%** of cherry-pick failures across variant pairs are attributable to refactorings in the target repositories. The most prevalent types include RenameMethod (53.1%), RenameParameter (41.1%), and MoveClass (3.5%), which break identifier traceability despite semantic equivalence. Like* `RefMerge`*, which reverts and replays refactorings to reduce merge conflicts in symmetric settings, our approach applies a similar strategy in an asymmetric context, where structural drift has accumulated. These results underscore the need for refactoring-aware integration tools to support reliable patch propagation across structurally divergent forks.*

---

*C. **RQ3: Can a refactoring-aware integration approach reduce merge conflicts and increase the success rate of applying patches across divergent variants?***

To assess the effectiveness of our refactoring-aware integration, we applied `RePatch` to all 309 cherry-pick failures identified in RQ1. `RePatch` extends Git's `cherry-pick` with semantic awareness via automated refactoring detection and transformation (see Section III). A patch is considered successfully integrated if it applies without textual conflicts, and resolution rate is measured as the proportion of previously failing patches that `RePatch` integrates successfully, compared to Git's baseline.

`RePatch` failed to complete execution for 17 of the 309 patches due to timeouts, which we conservatively treat as failures. Manual inspection showed that these typically involved many conflicting files or extensive edits. To manage resources and ensure experimental completion, we imposed a 15-minute timeout per patch. We hypothesize that lifting this constraint could yield more successful cases. Our reported analysis thus focuses on the remaining 292 patches, excluding timeouts from resolution statistics.

Table III shows the effectiveness of our refactoring-aware tool, `RePatch`, across 292 previously failing previously Git's `cherry-pick` patch integration scenarios. Each row reports the number of patches where Git cherry-pick failed, the number of patches where `RePatch` reduced, increase or did not change the amount of conflicting files and lines of code

TABLE III: Patch integration outcomes when applying `RePatch` to previously failing Git `cherry-pick` scenarios. The table reports the number of cases where `RePatch` reduced (↓), increased (↑), or unchanged (−) the number of conflicting files and lines of code (LOC), relative to the baseline Git cherry-pick conflict state. Only target variants with at least one cherry-pick failure are included in this analysis.

Target Variant	Total Failures	Conflicting Files			Conflicting LOC		
		↓ Reduced	↑ Increased	− Unchanged	↓ Reduced	↑ Increased	− Unchanged
linkedin/kafka	232	111 (47.8%)	4 (1.7%)	117 (50.4%)	114 (49.1%)	9 (3.9%)	109 (46.9%)
ufal/clarin-dspace	50	37 (74.0%)	0 (0.0%)	13 (26.0%)	37 (74.0%)	0 (0.0%)	13 (26.0%)
apache/kafka	3	3 (100%)	0 (0.0%)	0 (0.0%)	3 (100%)	0 (0.0%)	0 (0.0%)
typetools/checker-framework	2	2 (100%)	0 (0.0%)	0 (0.0%)	2 (100%)	0 (0.0%)	0 (0.0%)
bisq-network/bitcoinj	4	1 (25.0%)	0 (0.0%)	3 (75.0%)	0 (0.0%)	1 (25.0%)	3 (75.0%)
DSpace/DSpace	1	1 (100%)	0 (0.0%)	0 (0.0%)	1 (100%)	0 (0.0%)	0 (0.0%)
Total	**292**	**155 (53.1%)**	**4 (1.4%)**	**133 (45.5%)**	**157 (53.8%)**	**10 (3.4%)**	**125 (42.8%)**

(LOC) in a given target variant. For clarity, we use ↑, ↓, and − to indicates instances where `RePatch` increase, reduce or preserved the number of conflicts, respectively.

`RePatch` reduced file-level conflicts in 155 of 292 previously failing patch applications (53.1%) and line-level conflicts in 157 cases (53.8%), while increasing conflicts in only 1.4–3.4% of cases. These improvements were especially notable in high-conflict variants like `linkedin/kafka` and `clarin-dspace`, where `RePatch` resolved 47.8% and 74.0% of failures respectively. Even in variants with fewer structural differences, `RePatch` consistently outperformed Git. Overall, it successfully integrated 163 out of the original 309 failed patches, achieving a resolution rate of 52.8%.

To contextualize these results, we revisit the failure case in Example 2 (Listing 2). In that instance, Git's `cherry pick` produced 12 merge conflicts across 5 files, totaling 79 conflicting lines, mainly due to renaming and inlining. When processed with `RePatch`, 4 of the 5 files were resolved cleanly, and conflicts were reduced by approximately 75% (from 79 to 20 lines). However, `RePatch` also introduced a new conflict in `StreamThread.java`, a file that Git handled cleanly, resulting in 6 additional conflicting lines. Manual inspection attributes this to overgeneralized inlining heuristics misaligned with the target's constructor signature. This case mirrors broader trends in Table III, where `RePatch` reduced file-level conflicts in 53.1% of cases (155/292) and line-level conflicts in 53.8%, while introducing new conflicts in only 1.4–3.4%. The example illustrates both the strengths and trade-offs of refactoring-aware integration: `RePatch` can substantially improve patch portability by reconciling semantic drift in structurally evolved forks like `linkedin/kafka`, but also highlights the need for precise transformation heuristics to avoid regressions in otherwise conflict-free files. Together with RQ2, these findings show that refactoring-induced drift is not only common (91.6% of failures) but can be effectively mitigated through semantically informed integration.

Discussion and Implications:

Our results show that refactoring-aware integration reduces developer effort when maintaining divergent variants. `RePatch` performs well when semantic intent is preserved

(e.g., via renaming or reordering), enabling low-effort patch application. These findings support temporary structural alignment and refactoring replay as practical strategies. Compared to `RefMerge` [15], which resolves 25% of merge conflicts in symmetric settings, `RePatch` targets asymmetric integration and recovers 52.8% of failed cherry-picks. While the settings are not directly comparable, this improvement reflects `RePatch`'s tighter scope, commit-level granularity, and targeted integration workflow.

Failure cases expose limitations in semantic mapping and refactoring detection. Integration tends to fail when target-side changes delete or reorganize context-critical code, or when transformations are too subtle for current tools like `RefactoringMiner` [21]. This reflects earlier observations on the brittleness of automated merging under structural divergency [41], [42]. Although ML-based tools such as `DeepMerge` [18], `MergeBERT` [19], and `PPAtHF.` [16] show promise, they often lack transparency. In contrast, `RePatch` offers an interpretable rule-based pipeline that supports traceability and debugging. Our approach aligns with previous work on semantic program transformation [43] and reinforces the case for treating refactoring tracking as a first-class concern in integration workflows.

🔖 **Summary of RQ3 Results**: *Out of 292 previously failing patch applications, `RePatch` reduced file-level merge conflicts in **53.1%** of cases and line-level conflicts in **53.8%**. In contrast, it introduced new conflicts in only **1.4%** and **3.4%**, respectively. Overall, `RePatch` successfully integrated **155** of the **292** failed patches, achieving a resolution rate of **52.8%**. These findings show that refactoring-aware integration can substantially improve patch applicability while reducing manual effort and semantic misalignment associated with structural divergence.*

V. RELATED WORK

Fork Variants and Patch Reuse. Forking is common in open-source and industrial development to support experimentation, customization, or business-specific needs [2]. While

some forks are short-lived, long-lived variants often evolve independently with little integration of upstream changes [44], [45]. Ramkisoen et al. [5] used the `PaReco` tool to identify two major challenges: *missed opportunities*, where patches applied in one variant are absent or only partially reused in another, and *effort duplication*, where similar changes are reimplemented manually. However, prior work did not provide actionable mechanisms for patch transfer, especially when structural misalignments hinder reuse. Our work fills this gap by enabling systematic recovery of unpropagated patches through a refactoring-aware integration strategy that addresses variant-specific structural drift.

Learning-Based Patch Porting. Traditional tools like Git's `cherry-pick` assume syntactic similarity and often fail when code structure diverges across forks. Early machine learning systems such as `LASE` [43], [46] learn recurring edit patterns to apply patches across similar contexts but struggle when structural alignment is lost. Deep learning-based merge tools like `PatchNet` [47], `DeepMerge` [18], and `MergeBERT` [19] frame conflict resolution as token-level prediction or sequence modeling, achieving high accuracy on benchmarks. However, they typically assume symmetric divergence and lack interpretability, limiting their reliability in practice. Recent LLM-based approaches like `PPatHF` [16] use large models for zero-shot patch porting. While they reduce reliance on historical data, their performance degrades under significant refactoring and offers little transparency into how transformations are applied. This reflects the broader challenge identified by Barr et al. [48], who showed that code is often reused under renamed or structurally altered forms, making syntactic alignment unreliable.

Refactoring-Aware Merge and Detection Tools. Early work on operation-based merging [41], [49] emphasized modeling changes as semantic operations rather than textual diffs. Building on this vision, several rule-based systems have been developed to model code structure and refactorings explicitly. `MolhadoRef` [14], `IntelliMerge` [20], `CatchUp!` [50], `ReBA` [51], and `RefMerge` [15] implement semantic merge strategies that leverage AST differencing or transformation logs to resolve conflicts more accurately than line-based tools. However, these systems assume symmetric divergence within a shared repository history and are not designed for one-sided patch integration across forks. They often rely on precise refactoring detection, such as that provided by `RefactoringMiner` [21], [52], which identifies behavior-preserving changes like method renaming or class relocation [53]–[55]. Empirical studies have shown that refactorings can increase the likelihood of fault-prone commits [56]–[58], yet few systems operationalize this knowledge for cross-repository patch reuse. Our approach, `RePatch`, addresses this gap by extending `RefMerge` to operate at the granularity of individual patches and support asymmetric integration through patch-scoped refactoring inversion and replay.

Positioning Our Contribution. While prior work has diagnosed the challenges of patch propagation across forks, including integration failures due to structural drift [5], few tools provide actionable mechanisms to recover and apply unpropagated patches. Our contribution lies in bridging this gap through a refactoring-aware rule-based system that enables reproducible and interpretable integration of missed patches, even across repositories with no recent shared history.

VI. THREATS TO VALIDITY

Our findings are subject to a number of threats to validity.

The reliability of our semantic patch integration depends on accurate refactoring detection. We use `RefactoringMiner` [21], a widely adopted Java refactoring tool with high reported precision and recall. However, like all static analyzers, it may miss or misclassify fine-grained or intertwined refactorings, which can affect the correctness of inversion and replay. Our current study focuses on 14 Java-based variant pairs and does not yet include multilingual or proprietary systems. To broaden applicability, we are extending `RePatch` to support additional languages using parser-agnostic tools such as `Tree-sitter` and AST differencing.

Third, each patch integration attempt was subject to a 15-minute timeout to ensure experimental feasibility. 17 patches (5.5%) exceeded this limit, typically due to high file churn or extensive structural edits, and were conservatively treated as failures. Future work will investigate adaptive timeout strategies and analyze computational tradeoffs. Fourth, in rare cases, `RePatch` introduced conflicts not detected by Git `cherry-pick`, likely due to overgeneralized heuristics or incomplete refactoring mappings. We are improving our matching logic and introducing validation checks to increase precision and reduce false positives. Fifth, our evaluation focused on syntactic conflict resolution and did not perform functional validation of integrated patches (e.g., via test suites). Future work will assess behavioral correctness. Finally, our classification of refactoring-induced failures is based on static analysis and transformation logs. While scalable, this approach may not fully capture developer intent in ambiguous cases. To promote transparency and reproducibility, we publicly release our dataset, scripts, and logs [22].

VII. CONCLUSION

This study presents the first empirical analysis of patch integration failures across long-lived, structurally divergent forks, isolating refactorings as a key source of conflict. By modeling integration as a refactoring-aware process, we demonstrate that semantic alignment significantly improves patch applicability over Git's syntactic `cherry-pick`, resolving 52.8% of previously failing cases across 14 real-world variant pairs.

To enable this analysis, we extend `RefMerge` with commit-level, bidirectional refactoring tracking and replay, implemented in our prototype tool `RePatch`. These findings highlight the need for semantic reasoning in variant-aware development and suggest directions for future work, including multi-language support and integration into patch management pipelines.

References

[1] J. Krüger and T. Berger, "An empirical analysis of the costs of clone- and platform-oriented software reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 432–444.

[2] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018.* IEEE Computer Society, 2018, pp. 625–634.

[3] J. Businge, M. Openja, S. Nadi, and T. Berger, "Reuse and maintenance practices among divergent forks in three software ecosystems," *Empirical Software Engineering*, vol. 27, no. 2, p. 54, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10078-2

[4] J. Businge, A. Decan, A. Zerouali, T. Mens, and C. D. R. Serge Demeyer, "Variant forks – motivations and impediments," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[5] P. K. Ramkisoen, J. Businge, B. van Bladel, A. Decan, S. Demeyer, C. De Roover, and F. Khomh, "Pareco: patched clones and missed patches among the divergent variants of a software family," ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 646–658. [Online]. Available: https://doi.org/10.1145/3540250.3549112

[6] H. Rocha and J. Businge, "Blockchain-oriented software variant forks: A preliminary study," in *5th International Workshop on Blockchain Oriented Software Engineering*, 2022.

[7] S. Stanciulescu, T. Berger, E. Walkingshaw, and A. Wasowski, "Concepts, operations, and feasibility of a projection-based variation control system," in *International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 323–333.

[8] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product line," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 316–326.

[9] S. Zhou, "Improving collaboration efficiency in fork-based development," Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2020.

[10] J. Businge, A. Decan, A. Zerouali, T. Mens, and S. Demeyer, "An empirical investigation of forks as variants in the npm package distribution," in *Proceedings of the 19th Belgium-Netherlands Software Evolution Workshop, BENEVOL 2020, Luxembourg, December 3-4, 2020*, ser. CEUR Workshop Proceedings, M. Papadakis and M. Cordy, Eds., vol. 2912. CEUR-WS.org, 2020. [Online]. Available: http://ceur-ws.org/Vol-2912/paper1.pdf

[11] J. Businge, M. Abdi, and S. Demeyer, *Analyzing Variant Forks of Software Repositories from Social Coding Platforms*. Cham: Springer International Publishing, 2023, pp. 131–152. [Online]. Available: https://doi.org/10.1007/978-3-031-36060-2_6

[12] M. Njima, J. Businge, and S. Demeyer, "An empirical study of technical debt management as a motivation for forking," 2020.

[13] M. Lillack, S. Stanciulescu, W. Hedman, T. Berger, and A. Wasowski, "Intention-based integration of software variants," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 831–842.

[14] D. Dig, T. N. Nguyen, K. Manzoor, and R. Johnson, "Molhadoref: a refactoring-aware software configuration management tool," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 732–733. [Online]. Available: https://doi.org/10.1145/1176617.1176698

[15] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2698–2721, 2023.

[16] S. Pan, Y. Wang, Z. Liu, X. Hu, X. Xia, and S. Li, "Automating zero-shot patch porting for hard forks," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 363–375. [Online]. Available: https://doi.org/10.1145/3650212.3652134

[17] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 329–332. [Online]. Available: https://doi.org/10.1145/2934466.2962726

[18] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, "Program merge conflict resolution via neural transformers," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 822–833. [Online]. Available: https://doi.org/10.1145/3540250.3549163

[19] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri, "Deepmerge: Learning to merge programs," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1599–1614, 2023.

[20] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: a refactoring-aware software merging technique," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360596

[21] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 483–494. [Online]. Available: https://doi.org/10.1145/3180155.3180206

[22] D. Ogenrwot and J. Businge, "Refactoring-aware patch integration across structurally divergent java forks," Aug. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16756017

[23] GitHub. (2021, apr) Online appendix. GitHub. [Online]. Available: https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html

[24] D. Ogenrwot and J. Businge, "Patchtrack: Analyzing chatgpt's impact on software patch decision-making in pull requests," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2480–2481. [Online]. Available: https://doi.org/10.1145/3691620.3695338

[25] ——, "Patchtrack: A comprehensive analysis of chatgpt's influence on pull request outcomes," 2025. [Online]. Available: https://arxiv.org/abs/2505.07700

[26] 2025. [Online]. Available: https://github.com/apache/kafka/pull/13032

[27] 2025. [Online]. Available: https://github.com/apache/kafka/pull/12363

[28] 2025. [Online]. Available: https://github.com/apache/kafka

[29] 2025. [Online]. Available: https://github.com/DSpace/DSpace

[30] 2025. [Online]. Available: https://github.com/bitcoinj/bitcoinj

[31] 2025. [Online]. Available: https://github.com/javacc/javacc

[32] 2025. [Online]. Available: https://github.com/xerial/sqlite-jdbc

[33] 2025. [Online]. Available: https://github.com/typetools/checker-framework

[34] 2025. [Online]. Available: https://github.com/linkedin/kafka

[35] 2025. [Online]. Available: https://github.com/ufal/clarin-dspace

[36] 2025. [Online]. Available: https://github.com/bisq-network/bitcoinj

[37] 2025. [Online]. Available: https://github.com/langerhans/dogecoinj-new

[38] 2025. [Online]. Available: https://github.com/tulipcc/ParserGeneratorCC

[39] 2025. [Online]. Available: https://github.com/Willena/sqlite-jdbc-crypt

[40] 2025. [Online]. Available: https://github.com/eisop/checker-framework

[41] E. Lippe and N. van Oosterom, "Operation-based merging," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 5, p. 78–87, Nov. 1992. [Online]. Available: https://doi.org/10.1145/142882.143753

[42] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 190–200. [Online]. Available: https://doi.org/10.1145/2025113.2025141

[43] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 502–511.

[44] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, p. 131–183, Jun. 1992. [Online]. Available: https://doi.org/10.1145/130844.130856

[45] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma, "An empirical investigation into merge conflicts and their effect on software quality," *Empirical Software Engineering*, vol. 25, no. 1, pp. 562–590, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09735-4

[46] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," vol. 46, no. 6. New York, NY, USA: Association for Computing Machinery, Jun. 2011, p. 329–342. [Online]. Available: https://doi.org/10.1145/1993316.1993537

[47] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2021.

[48] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–317. [Online]. Available: https://doi.org/10.1145/2635868.2635898

[49] T. Mens and F. Van Rysselberghe, "Analyzing software evolution," in *IWPC*, 2004.

[50] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 274–283. [Online]. Available: https://doi.org/10.1145/1062455.1062512

[51] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "Reba: refactoring-aware binary adaptation of evolving libraries," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 441–450. [Online]. Available: https://doi.org/10.1145/1368088.1368148

[52] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[53] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[54] ——, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[55] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[56] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 104–113.

[57] I. Ferreira, E. Fernandes, D. Cedrim, A. Uchôa, A. C. Bibiano, A. Garcia, J. a. L. Correia, F. Santos, G. Nunes, C. Barbosa, B. Fonseca, and R. de Mello, "The buggy side of code refactoring: understanding the relationship between refactorings and bugs," ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 406–407. [Online]. Available: https://doi.org/10.1145/3183440.3195030

[58] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 556–567. [Online]. Available: https://doi.org/10.1145/3368089.3409695