Pushing the Boundaries of Patch Automation

***Abstract*—Patching is a fundamental software maintenance and evolution task enabling the (semi-)automated propagation of changes across different software versions. Established and widely used general-purpose patchers, such as `GNU-patch`, work on textual artifact representations (i.e., files) and typically rely on line numbers and contexts (i.e., surrounding unchanged text) to apply changes. This strategy often fails if source and target of a patch differ: Some required changes may be rejected, others may be applied to the wrong location; provoking cumbersome manual effort. In this paper, we study the effectiveness of current patchers, and propose a novel technique that pushes the boundaries of patch automation. First, we curate and analyze a large dataset of more than 400,000 patch scenarios (i.e., cherry picks) from 5,000 GitHub projects. Next, we examine the effectiveness of established patchers on the gathered patch scenarios, observing that patchers often fail to apply changes correctly. Third, we develop a novel general-purpose patch technique, `mpatch`, that utilizes a source-target matching to determine suitable change locations. By comparing `mpatch` to existing patchers, we find that it significantly improves recall while precision remains stable. Thus, `mpatch` can minimize the burden of manually fixing failed patches, specifically in projects with frequent patch applications.**

***Index Terms*—patching, cherry-picking, variant synchronization, change propagation, software maintenance, git**

I. Introduction

Patching is a core software maintenance activity which allows for propagating fine-grained software updates among software versions automatically. Patches are applied by patching tools or *patchers*, such as `GNU-patch` which is potentially the most well-known implementation of document patching, being at the heart of many software maintenance tasks for decades, such as contributing to the Linux kernel [1]. Patches summarize changes to a file that can be reapplied to copies of the file. These copies may reside, for instance, on different development branches in a version control system. Contrary to merging development branches [2], patching usually does not transfer all changes that occurred on a branch but a desired subset of changes, which is also known as cherry picking [3]. Therefore, patching is a crucial activity in projects that maintain co-evolving software versions simultaneously [4, 5, 6, 7], a common practice known as clone-and-own [8, 9, 10, 11]. Most patchers expect a list of changes performed on a source version as input, and apply these changes to one or more target versions in form of a *patch*. They try to identify the most suitable locations in the target and which changes should be applied. Figure 1 presents an example of such a *patch scenario*. In the beginning, a developer changes a file committed at (A1), representing the source version of the patch scenario, and commits the changes in (A2). Later, the developer derives a patch from these changes, and applies the patch to commit (B1), the target version, located on a different branch. The patch application (i.e., the patched target) results in commit (B2).

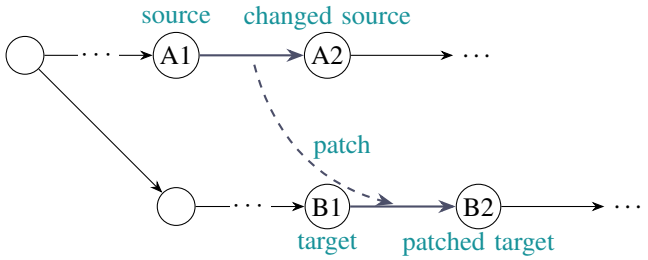

Fig. 1. Re-applying changes from a source on a target version via a patch.

While patching is trivial in cases where the source and target version of the patch are identical, it becomes challenging the more the source and target version differ [12, 13]. In such cases, patchers often apply the changes at wrong locations or even fail to apply them at all. For these complex patch scenarios, prior work found that current general-purpose patchers provoke a particularly high rate of rejected changes for patches of the Linux kernel [12, 13] requiring costly manual fixes.

In this work, we decrease the rejection rate of current patchers significantly by developing a novel general-purpose patcher, `mpatch`. During patching, `mpatch` computes a matching first, yielding a global alignment (i.e., matching) of the source and target file of a patching scenario. Thus, it detects the correct location for applying a change in the target more frequently than a local, context-based patcher.

First, we demonstrate empirically that current general-purpose patchers perform unsatisfactory in various patch scenarios, including domains other than the Linux kernel. We mine a dataset comprising 423,717 patch scenarios from 5,000 popular projects on GitHub, covering the 10 most used project languages. We evaluate the effectiveness of state-of-the-art patchers `GNU-patch`, `git apply`, and `git cherry-pick` by measuring precision, recall, and the number of required manual fixes. Overall, the patchers exhibit high precision but rather poor recall, provoking great manual effort.

Next, we compare the quality of `mpatch` with the established patchers. Compared to all current general-purpose patchers, `mpatch` significantly improves recall, increases the amount of patches that can be applied fully automatically, and reduces the number of patches that require manual fixes. Regarding the total number of required fixes, either `mpatch` or `git cherry-pick` perform best, depending on the project language. Lastly, `mpatch` and the other patchers perform equally for precision and execution time.

To gauge the potential impact of improving general-purpose patching, we investigate projects from our dataset that heavily use patching. We compute the impact `mpatch` would have, if the resulting projects adopted it. As a result, we find that `mpatch` has the potential for substantially reducing the manual efforts in such projects.

Overall, we gain strong empirical evidence that `mpatch` considerably improves existing general-purpose patchers while sharing their applicability to a broad spectrum of patch scenarios in various projects. In summary, we contribute

- a dataset comprising 423,717 patch scenarios (i.e., cherry-picks) mined from 5,000 popular projects on GitHub;
- a study of the prominence of patching in practice and the frequency of complex patch scenarios;
- `mpatch`: a novel general-purpose patcher that utilizes a source-target matching to push the boundaries of current patch automation;
- an extensive empirical evaluation and comparison of the effectiveness of `mpatch` and other general-purpose patchers in a mutlitude of complex patch scenarios;
- an online reproduction package [14], comprising our novel dataset, our implementation of `mpatch`, the experimental setup, and all results.

II. Motivation

In this section, we first explain the concept of patching using a motivational example. We then present the state-of-the-art general-purpose patching tools and explain their approaches.

A. General-purpose Patching

General-purpose patching, from here on referred to as 'patching', reduces the effort of applying the same changes to multiple versions of a file. Typically, such changes are first performed manually on one version and should be repeated on other versions that may benefit from integrating the same changes, e.g., because the changes fix a crucial bug. Changes can be automatically applied to a target version as a *patch*.

A patch typically is represented as a *diff* that documents changes to a source version, including unchanged text that surrounds the changes (i.e., their *context*) in a unified format, aka. (asymmetric) difference [15], (directed) delta [16], or edit script [17]. A *unified* diff aggregates changes with overlapping context in a *hunk* that defines the location of the changes in the source file. Figure 2 presents an example of such a unified diff, which we adapted from the commit ba66f3b in Apache hadoop. It comprises several changes that add new code for a Transport Layer Security verification in the Java file `WebHdfsFileSystem.java`, grouped into two hunks (starting at lines L181 and L242). Lines to be added by the patch are shown in green, surrounded by three context lines.

Over time, source and target of a patch may diverge – this is the case for the source and target of our example. If a divergence affects files to be patched, a patch created from the diff between the source and changed source may no longer suit the target. The changes in the two hunks of the patch shown in Figure 2 cannot be easily applied to the target because they need to be placed in a different location. Figure 3 shows excerpts of the source and target file, focusing on the location where the Line `private boolean isTLSKrb;` from the first hunk should be added. This location is indicated by a green arrow; common lines are highlighted in blue. In the source file, the line is added below L183, while in the target file it should be added below L171.

```
--- WebHdfsFileSystem.java
+++ WebHdfsFileSystem.java
@@ -181,6 +182,7 @@
[...]
  private DFSOpsCountStatistics storageStatistics;
  private KeyProvider testProvider;
+ private boolean isTLSKrb;

  /**
   * Return the protocol scheme for the FileSystem.
@@ -242,6 +244,7 @@
   .newDefaultURLConnectionFactory(connectTimeout,
       readTimeout, conf);
  }

+ this.isTLSKrb = "HTTPS_ONLY".equals(conf.get(
    DFS_HTTP_POLICY_KEY));

  ugi = UserGroupInformation.getCurrentUser();
  this.uri = URI.create(uri.getScheme() + "://" +
      uri.getAuthority());
[...]
```

Fig. 2. Adapted patch that was created from commit ba66f3b of hadoop. We omitted parts of the patch indicated by "`[...]`".

TABLE I
Overview of Evaluated Patchers

Tool Name	Type	Input
`GNU-patch` [18]	context-based	unified diff, target
`git apply` [19]	context-based	unified diff, target
`git cherry-pick` [20]	merge-based	git repository, changed source ID, target ID
`mpatch` (ours)	match-based	unified diff, source, target

For the second hunk of our example, we observe a similar alteration in location and context. In general, depending on how much the source and target files diverge from each other, some changes cannot be applied: the location of some changes may not be found, or it is questionable whether a change is actually required in the target (cf. Section IV-B4 for more details). Patchers typically reject changes if it is uncertain whether or where to apply them, instead they may be inspected and applied manually.

B. Current General-Purpose Patchers

Table I shows an overview of the patchers which we consider in this paper: `GNU-patch` [18], `git apply` [19], and `git cherry-pick` [20]. Naturally, this list is incomplete as other version control systems (VCS) (e.g., Mercurial or Subversion), editors, and IDEs contain their own patch utilities. Additionally, search and replace utilities, such as the stream editor `GNU sed`, could be appropriated for patching. In this work, we focus on patchers used by Git and Unix-based operating systems, cf. Table I. Git, with its patchers `git apply` and `git cherry-pick`, is the most popular VCS [21], and `GNU-patch` has great impact in open-source development (e.g., Linux kernel [1]).

1) `GNU-patch`: The first release of `GNU-patch` [18] was in 1985. Despite its age, it is still actively maintained, with the latest commit in March 2024 (as of July 2024), and remains a core development package of many Unix-based distributions.

```
179  private String restCsrfCustomHeader;
180  private Set<String> restCsrfMethodsToIgnore;
181
182  private DFSOpsCountStatistics statistics;
183  private KeyProvider testProvider;
184
185  /**
186  * Return the protocol for the FileSystem
187   *
188   * @return <code>webhdfs</code>
```

```
167  private Set<String> restCsrfMethodsToIgnore;
168  private static final ObjectReader READER =
169      new ObjectMapper().reader(Map.class);
170
171  private DFSOpsCountStatistics statistics;
172
173  /**
174  * Return the protocol for the FileSystem
175  * <p/>
176  *
```

Fig. 3. Excerpt from the source file (left) and target file (right) of our exemplary patch scenario from commit ba66f3b in Apache hadoop. Common lines are highlighted in blue. The green arrows indicate where the change of the first hunk in Figure 2 was added in the source file and should be added in the target.

`GNU-patch` expects a unified diff as input, and then applies these changes based on their line number and context. For each hunk in a patch, `GNU-patch` searches the hunk's context, starting at the line number specified in the hunk. Searching above and below that line number, `GNU-patch` applies the changes at the first location with a matching context. If it cannot find a suitable context in the file, it relaxes the context by removing the outermost line of the leading and trailing context and repeats the search. In its default configuration, `GNU-patch` repeats this context relaxation at most twice. Thus, `GNU-patch` allows for differences between the source and the target of a patch, except for the most direct neighboring context. If `GNU-patch` finds no suitable context, it reports the hunk as rejected.

Following this approach, `GNU-patch` might apply changes incorrectly or reject required changes if the source and target versions diverged substantially. When applying the patch of Figure 2 to the target in Figure 3 (right), `GNU-patch` rejects the first hunk containing the addition of `private boolean isTLSKrb;` because the context line before is different. In general, `GNU-patch` may reject changes if the first leading or trailing context line diverges. Besides rejecting the first hunk of our example, `GNU-patch` applies the change in the second hunk to a wrong location. This error occurs because `GNU-patch` applies changes to the first suitable location, which may not be the correct one, especially in cases with multiple context occurrences.

2) `git apply`: Git comprises two patch utilities, `git apply` [19] and `git cherry-pick` [20]. The former is git's counterpart to `GNU-patch` and uses a similar, context-based approach. In contrast to `GNU-patch`, however, `git apply`'s default configuration is much more conservative regarding differences between the source and target version. The complete context of a hunk must fit and if a single hunk of a patch is rejected, `git apply` rejects the entire patch. To override this behavior, `git apply` provides a `--reject` option. This option applies all applicable hunks and reports rejected hunks in a manner similar to `GNU-patch`.

In both default or non-default mode, `git apply` rejects the two hunks of our motivating example (cf. Figure 2), as the contexts of the hunks do not exactly fit the target version. In general, we expect that `git apply` will reject required changes more frequently than `GNU-patch`.

3) `git cherry-pick`: Git's second patch utility is `git cherry-pick` [20] which can be used to transfer one or more commits between branches without a full merge of these branches. It is tightly integrated with git's version control and requires a commit history; specifically, it can only be applied if the source and target version have a common ancestor in the commit history. Given a list of commits (i.e., patches), `git cherry-pick` re-applies the changes of these commits to the current working tree one-by-one, creating a new commit for each applied patch.

If source and target of a cherry pick have diverged, `git cherry-pick` may encounter conflicts. `git cherry-pick` tries to *merge* these files using the same merge strategy as `git merge`. Using git's default configuration, `git cherry-pick` applies changes if they are not part of a conflicting hunk (i.e., hunks whose context diverged in source and target). For conflicting hunks, `git cherry-pick` writes the conflict directly into the file by concatenating the hunk of the source and the hunk of the target, highlighting it with conflict markers. Such conflicts must be resolved manually.

In our experience, `git cherry-pick` can reliably identify the correct location for a change in most cases by considering the common ancestor of source of target version. However, for diverged versions, `git cherry-pick` may report merge conflicts even though the conflicting changes could simply be applied, and each conflict requires manual effort to resolve. For the patch of our motivating example, `git cherry-pick` would report merge conflicts for both hunks, provoking manual intervention.

III. Match-based patching with `mpatch`

We hypothesize that the shortcomings of current general-purpose patchers for diverged versions can be mitigated by utilizing a simple source-target matching. Current patchers either rely on a context which only contains information about the source version, but not the target, or on merging changes with respect to a common ancestor version, which may produce tedious merge conflicts. In contrast, a matching identifies where common text is located in the source and target versions, which makes it possible to more reliably identify the correct locations for changes in a patch. To this end, we developed `mpatch`, a new match-based patcher.

A. Overview

Figure 4 provides an overview of the three-phase process of `mpatch`. Its input is the source version, its unified diff to the

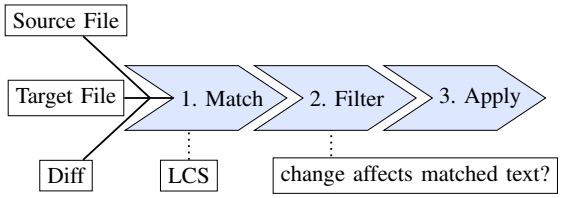

Fig. 4. Overview of the phases of `mpatch`.

changed source version, and the target version. First, the Match phase is responsible for determining a matching between the source and the target versions. Second, the Filter phase filters changes from the unified diff that are undesired in the target version. Lastly, the Apply phase applies the remaining changes to the target version according to the source-target matching.

1) Match: `mpatch` determines a source-target matching (for each to-be-patched file) based on the largest common subsequence (LCS) [22] of lines between a file and its counterpart in the target version. We selected LCS for its broad applicability to all kinds of textual software artifacts, and because it represents a proven basis for popular diff tools such as `diff` and for version control systems such as git.

Like other patchers, `mpatch` first has to determine which files should be compared to identify the right patch targets. To this end, `mpatch` considers all files that were changed in the given diff and locates their counterpart in the target version, similarly to `GNU-patch`. If a file has no counterpart, its content is treated as unmatched. If a counterpart is found, the source and target files are compared using LCS to match their common content. The result of this first phase is a matching of the files to be patched. For the source and target files of our motivating example, LCS correctly matches the common lines of both files as highlighted in Figure 3.

2) Filter: In this phase, `mpatch` filters all changes from the unified diff that are not valid for the target version. We chose a filter that uses the matching to decide whether a change should be filtered or not. `mpatch` considers a change as undesired if the change affects unmatched content of the source version (e.g., changes to a function in the source version that the target version does not have). Here, `mpatch` differentiates between added and deleted lines. Added lines cannot have a match themselves, thus, `mpatch` considers the lines above and below the added line in the source version. If the neighboring lines directly above *or* below have a match in the target, the added line is kept for the patch; thereby `mpatch` accounts for prepending or appending of lines to matched content. Without this filter, `mpatch` would still be able to find appropriate locations for required changes, but it would not be able to determine whether a line that was added to the source version should also be added to the target version. In our motivating example, `mpatch` keeps the changes because they add lines next to matched lines. Specifically, the addition of the first hunk happens directly above several matched lines as shown in Figure 3. In contrast, deleted lines must have a match in the target for the filter to keep them, because they could not be deleted otherwise. The result of the filtering phase is a filtered patch and a list of changes that were filtered, i.e., rejected – similar to rejects being reported by `GNU-patch`. These filtered changes may then be reviewed by a user. A filtered patch may be empty, in this case the patcher only reports filtered changes and quits.

3) Apply: The changes in the patch are applied to the target version according to the source-target matching. `mpatch` identifies the lines that should be deleted directly from the matching. They must have a match in the target, otherwise they would have been filtered out. For lines to be added, however, there is no direct match that determines the location of the addition. Therefore, `mpatch` considers the lines adjacent to the added line in the source variant from which the patch was created: Due to the filter phase, at least one of the neighboring lines must have a match in the target version, and `mpatch` adds the line below, above, or in between the matched neighboring line(s).

There may be cases in which an added line has two neighbors with a match in the source version, but these neighbors got separated in the target version. In such cases, `mpatch` adds the line next to the neighbor that has moved the least from its location in the source version w.r.t. its line number, preferring the preceeding neighbor in case of a tie; this design decision is based on the intution that changes should be applied to the most similar location. For the first change of our motivating example, `mpatch` adds the line directly below L171 in Figure 3, as this location is surrounded by the matches of the lines where the change was originally applied in the source.

B. Implementation

We implemented a prototype of `mpatch` [14] in Rust to evaluate its performance and compare it to the other patchers. `mpatch` offers a command line interface, similar to `GNU-patch`: the prototype accepts diffs calculated with `diff` as input. Therefore, `mpatch` could already be used in most cases in which `GNU-patch` is applied today and could also be integrated in a VCS such as git, cf. Table I.

IV. Evaluation Methodology

The central goal of our paper is to determine the boundaries of general-purpose patching in scenarios where the target of a patch differs from the source. To this end, we now present our study that aims to understand the effectiveness of current patchers in such scenarios (**RQ1**), to which degree our new patcher `mpatch`, presented in the prior section, can improve patch automation (**RQ2**), as well as the potential impact of improving patching in software maintenance (**RQ3**).

RQ1: *How effective are existing general-purpose patchers in complex patch scenarios?*

Previous work on patching in the Linux kernel found that `GNU-patch` may fail to patch C source code correctly, once patches become more complex, i.e., if the location for changes differs in the target [12]. However, it is unclear whether these observations generalize to patch scenarios in other projects, and whether other patchers exhibit the same limitations or automation potential. To close this knowledge gap, we evaluate

the effectiveness of three well-known patchers on a broad range of patch scenarios in public repositories.

RQ2: *How effective is* `mpatch` *in comparison to existing patchers?*

As a result of RQ1, we find that existing patchers frequently fail to identify the correct location to apply a change. By manually inspecting failing cases, we find that matching the source and target of a patch application might solve most of the encountered problems. Consequently, we developed `mpatch` (cf. Section III), a novel general-purpose patcher that relies on an matching. With RQ2, we measure the effectiveness of `mpatch` and compare it with the other patchers.

RQ3: *What is the potential impact of improving general-purpose patching with* `mpatch`*?*

To assess the significance of patching in practice and to gauge the impact of `mpatch`'s potential improvement over existing patchers, we study to which extent patching is a relevant software maintenance activity in practice. Specifically, patching may be more prevalent in certain types of communities or repositories (e.g., depending on the main programming language), which then would also be more heavily burdened by failing patches. To answer RQ3, we quantify the usage of `git cherry-pick` in public repositories on GitHub. Lastly, we compare the effectiveness of `mpatch` with the best current patcher, identified in RQ1, on projects where patching has the greatest impact.

A. Data Collection

To answer our research questions, we created a dataset of patching scenarios, specifically cherry picks (i.e., patches applied with `git cherry-pick`), mined from public repositories. First, we curated a set of 5,000 public projects from GitHub. We selected the ten most popular project languages on GitHub in the first quarter of 2024 by means of their number of stars. We identified the most popular languages based on the language overview of the GitHut project [23]. The most popular languages on GitHub were: Python, JavaScript, Go, C++, Java, TypeScript, C, C#, PHP, and Rust. Next, we collected 500 projects from the most popular projects for each language using GitHub's REST API: We filtered projects by language and sorted them by their number of stars in descending order, then selecting the top results. To identify a project's (programming) language, GitHub selects the language of the majority of artifacts in the repository, though some project artifacts are often written in another language (e.g., shell scripts or Docker files).

Within our dataset, we identified cherry picks among all commits and branches of the projects. While git provides a dedicated `git cherry` command that is meant to find cherry picks by comparing the diffs of commits, this command does not work for cases in which the patch performed by a cherry pick differs in the source and target – which are exactly the cases that are interesting to us. When users perform a cherry pick with `git cherry-pick` they can use an optional feature that appends a line to the commit message that states "*(cherry picked from commit <id>)*". To identify cherry picks, we parse the commit messages of all commits in a repository, searching for instances of such a line. Using the specified *<id>* of the commit, we can then retrieve the cherry. In a few repositories that rewrote their git histories (using git 'rebase'), we could not find some cherry commits and thus could not include them into our evaluation.

By mining cherry picks using the approach outlined above, we are confident that the identified cherry picks are indeed cherry picks; however, we cannot determine how many cherry picks actually were performed without the optional commit message appendix. Thus, there is potentially a large number of hidden cherry picks that we miss. We are not aware though of *reliable* alternatives to identify such hidden cherry picks.

B. Evaluation of Patcher Effectiveness

To evaluate the effectiveness of patchers, we use the cherry picks from our dataset as patch scenarios.

1) Considered Patchers: We consider `GNU-patch`, `git apply`, and `git cherry-pick` as state-of-the-art for general-purpose patching (cf. Section II-B). We use the default configurations for `GNU-patch`, and `git cherry-pick` because they are likely the most-used configurations in practice. To not discriminate against `git apply` in our evaluation, we invoke it with the `reject` flag (cf. Section II-B2), which simulates a behavior similar to `GNU-patch`, instead of rejecting patches as a whole.

2) Sampling of Patch Scenarios: Due to the large size of our dataset with more than 400,000 cherry picks (cf. Table III), we focus our analysis on patch scenarios for which we expect differences between patching techniques. To this end, we classify cherry picks into the classes 'trivial' and 'complex'. Trivial cherry picks do not require adjustments when being applied to the target version, and can be applied using a naïve patcher that blindly applies all changes according to their line number stated in the patch. In fact, for trivial patches, we observed that all patchers, including `mpatch`, performed perfectly in more than 99.99% of cases, with only rare cases causing issues. Hence, a new patching technique cannot improve the outcome for trivial patches. We hence discard trivial patches from further analysis (76.4% of all patches), and instead focus on the remaining 23.6% (101,196) complex patches to answer our research questions in Section V-B.

3) Automated Application of Patchers: To assess the effectiveness of a patcher, we replay each patch scenario in our sample. As illustrated in Figure 1, a patch scenario consists of

- a patch that we extract from the diff of source (A1) and changed source (A2) (i.e., the cherry),
- a target version (B1), that is the commit upon which the `git cherry-pick` command was originally applied,
- and the expected patched target (B2), which corresponds to the commit after the cherry pick, i.e., the ground truth.

To replay a patch scenario, we generate source, cherry, and target versions of the patch scenario, and we invoke a patcher with its command line interface. We then compare the patch outcome to the expected patched target (see Section IV-B4).

An additional step is required for `git cherry-pick`. In contrast to the other patchers, `git cherry-pick` does not directly reject changes but instead reports merge conflicts

TABLE II
OVERVIEW OF PATCH OUTCOME CLASSIFICATION

Change Type	Applied?	Location Correct?	#Observed Differences	Patch Outcome	Class
Required	✓	✓	0	*correct*	TP
Required	✓	×	2	*wrong location*	FP & FN
Required	×	N/A	1	*missing*	FN
Undesired	✓	N/A	1	*invalid*	FP
Undesired	×	N/A	0	*filtered/rejected*	TN

(cf. Section II-B3) that are written directly into the file. To appropriately compare `git cherry-pick` to the other patchers, we thus had to handle its conflicts. For comparability, we choose an approach that resembles the behavior of the other patches as close as possible. Other patchers write their rejected changes into a *rejects* file, instead of applying them. We mimicked this behavior for `git cherry-pick` by removing all git-markers and the *theirs* versions from a file after `git cherry-pick` finished. For conflicting locations, this left only the original content (the *ours* version), effectively simulating a rejection of conflicting hunks.

4) Classification of Patch Outcomes: Table II depicts our scheme for classifying a patch outcome. First, we compare the patched target version with the expected patch outcome using Unix `diff`, where we treat missing files as empty and ignore trailing whitespace, as it rarely carries important information. According to the observed differences in the patched target, we then classify the changes in a patch as exactly one of the five classes in Table II. The classes distinguish required changes that must be applied by a patcher, and undesired changes that must not be applied and hence rejected (first column). For required changes, there are three cases that may occur (second to last column): the change has been applied to the correct location (*correct*), the change has been applied to an incorrect location (*wrong location*), and the change has not been applied (*missing*). For undesired changes, we also distinguish between a change having been applied (*invalid*) anywhere, and a change not having been applied (*filtered/rejected*).

Using this methodology, we focus only on the patch application itself, effectively ignoring differences that are not directly related to the content of the patch, that is differences caused by a developer having manually performed additional changes while cherry-picking. We ignore these differences because the evaluated general-purpose patchers focus on applying patches, and do not alter the target in any other way. Furthermore, we only consider text files – excluding binary files: A small edit in a binary file can be just as problematic as a large one, and a diff cannot effectively represent differences in binary files.

5) Evaluation Metrics: We measure effectiveness of patchers in terms of five different metrics.

a) Precision and Recall: We measure precision and recall because they reflect whether required changes are correctly applied and undesired changes rejected. To determine precision and recall, we consider patchers as classifiers for changes [24, 12] that try to determine whether a change is *required* or *undesired*, with required changes being positive and undesired changes negative instances. Thus, for required changes, we count correctly applied changes as true positive (TP), and rejected changes as false negatives (FN) (cf. Table II). For undesired changes, we count applied changes as false positive (FP), and not-applied changes as true negative (TN). Required changes that were applied to the wrong location represent a special case because they cause an undesired change (at the wrong location) and a missing change (at the correct location); we counted them as both FP and FN, which also reflects the additional effort of developers in such cases.

Precision is the ratio of correctly applied required changes among all applied changes, which is given by $\frac{TP}{TP+FP}$, while recall is the ratio of required changes that were applied correctly, which is given by $\frac{TP}{TP+FN}$.

b) Patch Automation Percentage and Required Fixes: These metrics reflect the manual effort of developers patching. The automation percentage measures how often a patcher is able to apply a patch without human intervention. We determine the patch automation percentage as the ratio of the number of patches with 100% precision and recall (i.e., no fixes needed), and the total number of patches. We also measure the average number of fixes that are required after a patch application by counting the number of line-sized changes required to correct the artifact after the patcher finished.

c) Average Runtime in Seconds: Lastly, we measure the patch runtime, ignoring setup and clean-up tasks.

V. RESULTS

A. Overview of our Dataset of Mined Cherry Picks

In Table III, we present an overview of our gathered dataset allowing us to examine the prominence of change propagation in public software development practice. In total, we mined cherry-picks from 5,000 repositories (500 from each of the 10 most popular languages on GitHub). While in 4,304 projects we cannot find cherry picks based on the commit messages, we find a total of 423,717 cherry picks spread across 696 repositories. The column ‘cherry pick [%]‘ denotes the cherry-pick-to-commit ratio; i.e., how many percent of the commits are cherry picks. Next, the column ‘complex cherry pick [%]‘ indicates the percentage of sampled cherry picks that are complex, meaning the source and target do not match perfectly. The complex cherry picks represent the patching scenarios we studied for RQs 1 and 2.

Language-wise, we observe notable differences, specifically in the absolute and relative occurrence of cherry picks. The projects implemented mainly in C and C++ encompass half of all identified cherry picks in absolute numbers whereas JavaScript projects use them least frequently in absolute and relative numbers. Conversely, Java projects exhibit the highest cherry-pick-to-commit ratio on average: more than every 50$^{\text{th}}$ commit is a cherry pick. Moreover, we observe differences regarding the ratio of complex cherry picks among the identified cherry picks. Projects implemented in PHP exhibit the lowest ratio of complex cherry picks (on average: 13.2%), while C++ and Rust projects have the highest ratio with averages of 31.8% and 31.9%, respectively. The average ratio of complex cherry picks across all projects is 23.9%. Thus, almost every fourth cherry pick requires target-specific adjustments (e.g., applying changes to locations different from the source).

TABLE III
OVERVIEW OF OUR COLLECTION OF GITHUB PROJECTS.

main repository language	#sampled projects	#projects w. cherry picks	#cherry picks	cherry pick [%]	complex cherry pick [%]
C	500	74	108,595	1.292	19.1
C#	500	61	6,929	0.334	22.1
C++	500	112	108,085	1.437	31.8
Go	500	98	28,941	1.234	28.6
Java	500	91	81,859	2.232	20.6
JavaScript	500	43	3,243	0.187	22.4
PHP	500	54	22,593	0.830	13.2
Python	500	57	33,152	1.130	27.1
Rust	500	39	7,033	0.329	31.9
TypeScript	500	67	23,287	0.728	19.2
total	5,000	696	423,717	1.153	23.9

B. Evaluation of Patchers on our Dataset

To answer RQs 1 and 2, we studied the patchers introduced in Section II-B and `mpatch` on our dataset. Table IV presents the key results of our evaluation. For each patcher, we measured precision, recall, automation degree, the number of required fixes for a failed patch, and its execution time (cf. Section IV-B5). Values highlighted in bold font perform best for each metric and project language. The rightmost columns present the mean ($\bar{x}$) over all languages, the relative difference of these means of the patchers compared to `mpatch` ($\pm\%$), and the effect size $|r_{RB}|$, computed with the rank-biserial correlation [25]. To test the null hypothesis whether the results of `mpatch` or of another patcher stem from the same distribution, we used the Wilcoxon signed-rank test [26]. In all cases, the hypothesis got rejected with $p \ll 0.01$. This shows that `mpatch` behaves significantly different than the other patchers for all metrics measured. Next, we describe the results presented in Table IV in detail, illuminating the metrics followed by analyzing language-specific results.

Precision: All patchers exhibit high precision (> 0.9), with `git apply` showing the highest precision for every language but C++. `mpatch` performs slightly worse than the remaining tools, which perform 1.50% to 2.57% higher in precision.

Recall: `GNU-patch` and `git cherry-pick` achieve recalls of about 0.7 and 0.8, respectively; `git apply` performs worse with a recall of about 0.65. In contrast, `mpatch` scores 0.92 for its lowest recall level. Thus, it outperforms the remaining patchers across all project languages. On average, we observe that the second-best patcher, `git cherry-pick`, has a 13.53% lower recall than `mpatch`.

Patch Automation Percentage: All of the current patchers show a low degree of automation (`git apply`: 16%, `GNU-patch`: 40%, and `git cherry-pick`: 41%). In contrast, `mpatch` can automatically apply 59% of complex patches correctly. Thus, `mpatch` outperforms the remaining patchers over all languages; `git cherry-pick` shows a 30% lower automation percentage in relative terms.

Required Fixes: The number of lines developers are required to fix after patching shows a high volatility. `git cherry-pick` performs best with 34 fixes needed, while `mpatch` requires more lines to be fixed at 82 fixes on average. `GNU-patch` and `git apply` perform similarly with 104 and 107 fixes, respectively.

Runtime: Lastly, all patching techniques need much less than one second per patch on average although we observed outliers for all techniques, sometimes taking dozens of seconds. `GNU-patch` performed the fastest for all languages.

Language-wise, we observe some variance; e.g., all patchers have lower precision and recall for C++ and C# projects. Similarly for automation level, we observe the worst performance across all patchers for C++ and C#. Moreover, we find that patch scenarios in PHP projects require much fewer fixes than scenarios in other project languages. The number of required fixes varies largely w.r.t. the project languages and patchers; e.g., for JavaScript, patches applied by `mpatch` need ten times less fixes compared to the remaining patchers. For C++, Java, and C, `git cherry-pick` performed best while `mpatch` performed best in the remaining languages. In terms of execution time, there is no outstanding behavior.

All in all, `mpatch` performs slightly worse in precision than the other patchers but improves recall and the degree of automation significantly. Although `mpatch` requires less fixes than `git cherry-pick` for most languages, on average `git cherry-pick` performs best in this category.

C. Impact of `mpatch` on Frequently Patched Projects.

For RQ3, we compare the performance of `mpatch` to the currently best-performing patcher `git cherry-pick` in the five projects with the highest number of cherry picks. Table V presents the result of this comparison. The ratio of cherry picks to commits ('cherry pick [%]') shows that the projects utilize cherry picking very differently. Some projects mostly commit trivial cherry picks for which we do not expect differences between the patchers. For instance, 'intellij-community' features only 859 complex cherry picks out of 12,998 cherry picks in total. Next, the table presents the amount of required fixes on average per complex patch (columns 'required fixes `<tool>`'). We can then calculate the total number of required *manual* fixes in a project; e.g., for 'ceph' this would be $30.8 \frac{\text{fixes}}{\text{complex cherry pick}} \cdot 7{,}758$ complex cherry picks $=$ 238,946 fixes for `git cherry-pick` and $18.0 \cdot 7{,}758 =$ 139,644 for `mpatch`. The last two columns show the percentage of complex cherry picks without required fixes.

VI. DISCUSSION

A. Answers to Our Research Questions

*1) **RQ1**: Effectiveness of current general-purpose patchers:* In our evaluation of current patchers, `git cherry-pick` performed best. Compared to `GNU-patch`, it also has the advantage that it is integrated directly into a version control system. This saves manual effort to commit changes, especially for trivial patches that can be applied without fixes. However, it comes at the price of `git cherry-pick` being less applicable, specifically to git projects only. For instance, it is hardly possible to create a patch and transfer it to another project. Moreover, `git cherry-pick` does not outperform `GNU-patch` as its merge-based strategy is susceptible to any

TABLE IV
COMPARISON OF THE CONSIDERED PATCHERS FOR VARIOUS PROJECT LANGUAGES.

Metric	Patcher	Project Languages: Python	JavaS.	Go	C++	Java	TypeS.	C	C#	PHP	Rust	$\bar{x}$	±%	$\lvert r_{RB}\rvert$
Precision	`mpatch` (ours)	0.94	0.92	0.96	0.91	0.95	0.95	0.94	0.91	0.90	0.95	0.94		
	`GNU-patch`	0.96	0.93	0.97	0.93	0.96	0.97	0.95	0.93	0.92	0.96	0.95	1.50%	0.19
	`git apply`	**0.97**	**0.98**	**0.98**	0.94	**0.98**	**0.98**	**0.96**	**0.96**	**0.92**	**0.97**	**0.96**	2.57%	0.30
	`git cp`	0.95	0.95	0.97	**0.94**	0.97	0.97	0.95	0.94	0.91	0.96	0.95	1.68%	0.18
Recall	`mpatch` (ours)	**0.95**	**0.94**	**0.97**	**0.96**	**0.97**	**0.96**	**0.96**	**0.92**	**0.98**	**0.97**	**0.96**		
	`GNU-patch`	0.81	0.75	0.84	0.77	0.84	0.80	0.76	0.76	0.87	0.84	0.80	−16.90%	0.70
	`git apply`	0.70	0.59	0.75	0.63	0.70	0.71	0.55	0.62	0.77	0.74	0.66	−31.86%	0.86
	`git cp`	0.84	0.77	0.88	0.81	0.85	0.84	0.80	0.78	0.89	0.87	0.83	−13.53%	0.66
Autom. (%)	`mpatch` (ours)	**59.93**	**62.05**	**62.12**	**52.08**	**59.78**	**57.86**	**65.83**	**49.50**	**56.47**	**50.07**	**58.89**		
	`GNU-patch`	44.19	39.16	35.50	36.59	39.79	40.51	45.08	33.74	45.77	34.75	40.26	−31.63%	0.36
	`git apply`	24.24	14.63	13.85	17.19	11.55	20.13	13.10	15.30	26.81	16.56	16.05	−72.74%	0.76
	`git cp`	42.10	36.46	49.27	37.38	38.36	42.15	43.87	33.81	44.64	36.17	41.08	−30.21%	0.34
Req. Fixes	`mpatch` (ours)	**11.57**	**8.85**	**27.24**	169.82	143.27	**12.03**	32.35	**69.59**	**5.06**	**10.82**	81.53		
	`GNU-patch`	37.26	91.44	54.01	199.43	160.21	35.39	43.97	136.17	15.56	37.45	104.03	27.61%	0.36
	`git apply`	40.31	94.30	58.43	203.89	172.23	38.42	35.17	139.19	17.70	44.57	106.76	30.92%	0.55
	`git cp`	23.54	82.57	36.23	**33.80**	**57.72**	20.93	**20.75**	72.35	10.19	26.77	**34.12**	−58.19%	0.31
Time (s)	`mpatch` (ours)	0.06	0.05	0.17	0.25	0.09	0.24	0.06	0.05	0.05	0.09	0.13		
	`GNU-patch`	**0.03**	**0.03**	**0.04**	**0.04**	**0.04**	**0.04**	**0.03**	**0.03**	**0.04**	**0.05**	**0.03**	−73.49%	0.36
	`git apply`	0.16	0.06	0.26	0.33	0.39	0.41	0.31	0.22	0.12	0.09	0.30	127.36%	0.79
	`git cp`	0.15	0.09	0.15	0.34	0.27	0.28	0.23	0.17	0.13	0.14	0.24	84.47%	0.76

TABLE V
COMPARISON OF AUTOMATION POTENTIAL OF `MPATCH` AND `GIT CHERRY-PICK` IN PROJECTS WITH THE MOST CHERRY PICKS.

repository	cherry pick [%]	#cherry picks	#complex cherry picks	#required fixes `git cp`	#required fixes `mpatch`	fully autom. `git cp`	fully autom. `mpatch`
IntelliJ	2.45	12,998	859	6.8	3.0	56.7%	66.0%
Hadoop	21.02	14,553	3,618	15.6	5.0	34.2%	62.0%
FreeBSD	2.31	21,266	3,316	8.3	4.3	42.3%	67.0%
FFmpeg	17.99	23,774	3,829	3.3	1.2	55.0%	76.4%
ceph	22.01	32,651	7,758	30.8	18.0	36.0%	59.3%

change in the context being reported as conflict. In our evaluation, we inspected several patches manually, to understand the workflow of patchers and their strengths and weaknesses. We observed cases in which `git cherry-pick` reported extremely complex merge conflicts, even if the patch only contained one or two changes. For example, the cherry pick applied as commit 0539294 in the project moby changed a single line in the target, but upon replaying, git reports a merge conflict spanning several hundred lines. Such extreme cases highlight the limitations of the merge-based strategy.

In general, current patchers achieve high precision but fall short in the recall and patch automation rates. Overall, less than 50% of the complex patches can be automatically applied with current patchers. In terms of recall and automation, `git apply` performed much worse than `GNU-patch` and `git cherry-pick`. This is likely due to its strict rejection heuristic: any difference in the target's context leads to a rejection (cf. Section II-B2). Interestingly, this behavior only increases precision marginally. By comparing the results of `git apply` and `GNU-patch`, we find that a partially common context is a reliable indicator that a change is required as `GNU-patch` achieves a much higher recall with its more relaxed context-based strategy.

RQ1: *`git cherry-pick` is the most effective current patcher, when git histories are available. While current patchers can identify undesired changes reliably, they often reject required changes or apply them at wrong locations. To increase automation, new patch tools should improve recall as this promises better overall results.*

2) ***RQ2****: Effectiveness of `mpatch`:* Our results show that `mpatch` achieves consistently better recall than current patchers across all project languages. This also increases the patch automation considerably: `mpatch` performs 43% better relative to `git cherry-pick`, i.e., `git cherry-pick` performs 30% worse than `mpatch`. We observe a similar improvement for the Top-5 projects with the highest number of patches (cf. Table V). Here, `mpatch` clearly outperforms `git cherry-pick` with respect to the automation potential and number of required fixes.

While the mean number of required fixes is considerably lower for `mpatch` for most project languages, surprisingly, `mpatch` requires considerably more fixes on average than `git cherry-pick` for patches in C++ and Java projects. We re-analyzed our results omitting outliers; i.e., the Top-0.5% results w.r.t. the number of required fixes. The corresponding version of Table IV is part of our reproduction package [14]. This omission had a large impact on the average fixes per patch, while the other metrics remained roughly stable. For `mpatch`, the average fixes went down from 81.53 to 5.77; for the other patchers the impact was smaller, e.g., from 34.12 down to 9.95 for `git cherry-pick`. For the other metrics, we do not observe such extreme outliers. We interpret these extreme outliers as scenarios, where patchers without VCS information break completely whereas `git cherry-pick` can leverage additional information from the git history.

mpatch has almost perfect recall, but slight potential to increase precision. The filter heuristic employed by mpatch assumes that required changes probably affect matched text. We plan to examine other heuristics in the future.

Overall, our prototypical implementation of mpatch outperforms all tested patchers, including git cherry-pick, despite biases in favor of git cherry-pick (cf. paragraph VI-B0a). Thus, our prototype improves the state of the art and may be used right away.

***RQ2**: Our evaluation presents strong empirical evidence that mpatch outperforms current general-purpose patchers on a wide spectrum of patch scenarios. Using mpatch, practicioners could on average apply 40% more patches without human intervention than with the best current patcher.*

*3) **RQ3**: Potential impact of improving general-purpose patching with mpatch:* Finding that mpatch improves patching for individual patching scenarios, we also gauge its potential impact for developers, in general. To this end, we first consider the prominence of patching in public repositories, and second, the heavy reliance of some projects on patching.

While mining cherry picks from repositories, we were only able to identify cherry picks in 696 out of 5,000, but we may have missed patches that were applied with other tools than git cherry-pick. As we only selected cherry picks with dedicated, optionally-generated commit messages (cf. Section IV-A), it is likely that we missed an unknown number of cherry picks without this message, and patch scenarios that were created with other patchers or manual copy-and-pasting. Despite this limitation, we were able to identify about 423k patches, with about 101k of them being complex. This corresponds to about 144 complex cherry picks per project with cherry picks. Of these 101k complex patches, mpatch is able to automatically apply about 17k more patches than the best current patcher. For each of these patches, manual effort is saved in addition to requiring fewer fixes in most cases.

We also investigated the potential impact of mpatch in the Top-5 projects with the highest number of cherry picks (cf. Table V). Here, we observe that patching is a central aspect for three of these five projects, as more than 17% of commits are patches. For these projects, mpatch could lead to a substantial reduction in manual effort that is caused by fixing patches. The most extreme case we found is in ceph: 22% of their commits are patches. In ceph, mpatch could have applied 1,800 more patches automatically than git cherry-pick, while requiring only half the number of fixes.

***RQ3**: Our evaluation shows that mpatch could have a large positive impact on patching in many different projects and domains, especially projects that heavily utilize patching.*

B. Threats to Validity

a) Internal Validity: As with any software, bugs in our evaluation setup might lead to incorrect conclusions. To ensure the correctness of mpatch, we implemented unit and integration tests that check its match, filter, and apply phase in various patching scenarios. Whenever applicable, we also integrated implementations of trusted libraries into our prototype, e.g., for the LCS matcher. We further reviewed edge cases and anomalies to double-check our complete evaluation work flow.

In our evaluation we calculate various metrics that are influenced by the outcomes of Unix diff; we use it to assess the difference of the expected patch target and the achieved patch target. Such a diff is intrinsically heuristic, i.e., there are many different but valid diffs of two files. GNU-patch, git apply, and our mpatch use a diff as input, and thus may be biased by this. However, most other patchers are either also based on Unix diff directly or a derivate of it.

All patch scenarios in our evaluation were originally performed using git cherry-pick, which introduces two strong biases in its favor. First, git cherry-pick's patch results are often more similar to our evaluation's ground truth, because it was derived from git cherry-pick: each of our patch scenarios is the result of a developer using git cherry-pick, and then fixing its result, i.e., resolving merge conflicts. In each patch scenario, a developer thus started to resolve conflicts after they were presented the git cherry-pick results of our evaluation. Second, our dataset may miss cases where git cherry-pick performed exceptionally poor. Such cases could be either git cherry-pick crashing or creating too many merge conflicts to be managable. Instead of attempting to deal with these scenarios and commit their result, these scenarios may have just been skipped or handled with another tool. It is thus remarkable that mpatch still performed better than git cherry-pick for many metrics.

We used all patchers in their default configuration in our evaluation. Each patcher offers various fine-tuning options for specific tasks, and using these configurations may yield different outcomes. However, we chose to evaluate only the most straightforward, and likely most commonly used default configurations. We argue that this approach provides a good impression of a patcher's average performance.

b) External Validity: Our dataset is limited to public projects on GitHub and our observations thus might not be generalizable to closed-source projects. However, many professional developers participate in projects on GitHub and our dataset of diverse projects should cover many different programming practices. Similarly, our results may not generalize to smaller, less popular projects. However, a better patching tool would have a smaller impact for these, anyway.

In our evaluation, we only consider patch scenarios that stem from cherry picks and thus git cherry-pick. Other patchers (e.g., classic Unix diff and GNU-patch, manual propagation) are not reflected by our dataset and may exhibit different properties. Currently, our knowledge of patch scenarios from other tools is severely limited, and we have no way of collecting them, automatically. However, our dataset of patches is much bigger and covers more domains than in prior work, broadening our knowledge at least for cherry picks.

VII. Related Work

A. Techniques Related to Patching

1) Patch Backporting: Patch backporting creates a patch from changes applied to a newer version of software and transforms this patch to fit an older version. Most backporting techniques focus on backporting patches for the Linux kernel [12], some specialize on Linux device drivers only [27, 28], while others are applicable for code of a specific language, e.g., C code [12, 13] or PHP [29]. Harnessing the syntax, semantics, control flow or dependencies of their application domain, they may even outperform `GNU-patch`, i.e., `FixMorph` [12] and `TSBPORT` [13]. However, this effectiveness comes at the price of limited applicability. We consider a direct comparison with patch backporting out of the scope of this paper as backporting techniques are complementary to `mpatch`: While `mpatch` can be used for general-purpose patching, backporting may be used for their specialized use cases.

2) Patching in Variant-Rich Systems: Prior work investigated the potential to automate patching for clone-and-own variants (i.e., diverged versions) [24]. They simulated diverging versions of the Unix suite BusyBox and observed that `GNU-patch` achieved a high precision and recall of 0.92 and 0.93. While we report a similar precision for `GNU-patch`, we found that its recall does not generalize to patch scenarios in public projects, with an average recall of only 0.80.

Research on patch propagation [30], patch mutation [31], patch filtering [32], or differencing [33] for configurable software (i.e., software product lines) assumes the existence of explicit documentation on (1) the available software versions or variants, and (2) the relation of source code to configuration options (e.g., via C preprocessor annotations). *Variation* control systems [34, 35, 36, 37, 38, 39], such as ECCO [40, 41] and SuperMod [42], propagate changes to software variants but require a single representation of all software variants similar to software product lines. While such methods are effective when explicit knowledge on variability is present, they are neither applicable as general-purpose patchers nor account for diverged versions without this explicit knowledge.

B. Studies on Patching Practices

Businge et al. [43] conducted a study on patching between forks in Android, .NET, and JavaScript systems. They found that the rate of patching between forks is overall low and that most patches are applied to forks as pull requests (i.e., merges). They also investigated patching between two forks and observed that `git cherry-pick` is seldom used (9% of Android, 0.9% of .NET, and 2.5% of JavaScript). In contrast, we found cherry-picks in about 14% of the projects in our dataset. We suspect that this discrepancy is due to their more conservative method of considering only trivial cherry-picks. Ramkisoen et al. [44] investigated the "patch technical lag" in divergent forks: They found that it takes 27 weeks on average until a bug fix is propagated to a fork that requires it. Jang et al. [45] investigated vulnerability patching over time. They found that the majority of clones remain unpatched after one year, and that some clones remain unpatched even longer. All of these empirical findings suggest a need for better tool support that helps with identifying and applying patches.

Regarding the complexity of patching, Shariffdeen et al. [12] investigated the typical number of patches that are backported and the time required to backport them. They found that many patches are backported to Linux kernel versions (about 8% per Linux version) and that backporting requires more than 20 days for 80% of patches. Furthermore, when analyzing patches they observed that only 23% of backported patches were trivial. This suggests that patch backporting is more complex than patching in general, as we observed that more than 75% of patches in our dataset were trivial.

VIII. Conclusion

In this paper, we investigated the prominence of patching in public repositories and the effectiveness of patchers in complex patch scenarios. We mined a dataset of ca. 100k complex patch scenario from 5,000 public repositories on GitHub to study the effectiveness of current patchers. We observed that they apply patches with high precision, but low recall as current patchers struggle to identify the correct change locations confidently. This causes an overall low rate of automation of only 16% to 41% of the investigated patch scenarios. To address these shortcomings, we introduced `mpatch`, a novel match-based patching technique. `mpatch` achieves a higher recall and thereby higher automation rate than the best current patcher in the studied patch scenarios, by a margin of 30%. Lastly, we investigated the potential impact of `mpatch` on patching in practice. Out of the 100k complex patches in our dataset, `mpatch` could have correctly and automatically applied 17k patches more than the best current patcher. This directly impacts projects that heavily rely on patching. For instance, in ceph, we found more than 7,000 complex patches from which `mpatch` could correctly apply 25% more patches and reduce the number of necessary fixes by half for the remaining ones.

Our work shows that patching remains a challenging problem in software maintenance and evolution. Particularly, when the source and target of a patch diverge, current patchers fall short. While related techniques (e.g., patch backporting) are effective in cases of diverging source and target versions, they have stricter requirements, limiting their application to specific use cases and file formats. In contrast, our novel *general-purpose* patching technique, `mpatch`, can immediately boost patch automation in most cases in which these tools are not applicable, which reduces the burden of manual effort associated with failing patches immediately. Consequently, our work paves the way for more efficient maintenance and evolution of complex software projects, for instance, by integrating `mpatch` into version control systems. Additionally, our large dataset of complex patches can be used to evaluate the effectiveness of novel patching techniques, or to study patching practices in public repositories.

In conclusion, our design and implementation of a prototypical patcher demonstrated superior effectiveness compared to a family of tools that have evolved and matured over almost 40 years. This underscores the importance of re-evaluating even the most established methods and techniques in our more than vibrant research area of software engineering.

References

[1] J. Juhl, "Applying Patches To The Linux Kernel," Website: https://www.kernel.org/doc/html/v4.11/process/applying-patches.html, 2016, accessed: 2024-07-01.

[2] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, 2002.

[3] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Control With Subversion*. O'Reilly Media, Inc., 2004.

[4] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2013, pp. 101–110.

[5] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[6] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible Product Line Engineering With a Virtual Platform," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2014, pp. 532–535.

[7] S. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 151–160.

[8] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining Feature Traceability With Embedded Annotations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2015, pp. 61–70.

[9] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-Automated Feature Traceability With Embedded Annotations," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 529–533.

[10] P. M. Bittner, A. Schultheiß, T. Thüm, T. Kehrer, J. M. Young, and L. Linsbauer, "Feature Trace Recording," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021, pp. 1007–1020.

[11] T. Kehrer, T. Thüm, A. Schultheiß, and P. M. Bittner, "Bridging the Gap Between Clone-and-Own and Software Product Lines," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2021, pp. 21–25.

[12] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated Patch Backporting in Linux (Experience Paper)," in *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 633–645.

[13] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing OSS Patch Backporting with Semantics," in *Proc. SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 2023, pp. 2366–2380.

[14] A. Authors, "The reproduction package of this paper," Website: https://anonymous.4open.science/r/patching-with-matching-eval-36B3/README.md, Aug. 2024.

[15] T. Kehrer, U. Kelter, and G. Taentzer, "Consistency-Preserving Edit Scripts in Model Versioning," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2013, pp. 191–201.

[16] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.

[17] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of Model Comparison Tools," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2012, pp. 306–309.

[18] L. Wall, P. Eggert, W. Davison, D. MacKenzie, and A. Grünbacher, "GNU patch," Website: https://savannah.gnu.org/projects/patch/, 2009, accessed: 2024-07-01.

[19] L. Torvalds, J. C. Hamano *et al.*, "git-apply," Website: https://git-scm.com/docs/git-apply, 2023, accessed: 2024-07-01.

[20] ——, "git-cherry-pick," Website: https://git-scm.com/docs/git-cherry-pick, 2023, accessed: 2024-07-01.

[21] S. E. Inc., "Beyond Git: The other version control systems developers use," Website: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/, 2023, accessed: 2024-07-01.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[23] F. Beuke, "Githut 2.0 – a small place to discover languages in github," Website: https://madnight.github.io/githut/#/stars/2024/1, 2024, accessed: 2024-06-01.

[24] A. Schultheiß, P. M. Bittner, T. Thüm, and T. Kehrer, "Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 269–280.

[25] E. E. Cureton, "Rank-Biserial Correlation," *Psychometrika*, vol. 21, no. 3, pp. 287–290, 1956.

[26] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[27] L. R. Rodriguez and J. Lawall, "Increasing Automation in the Backporting of Linux Drivers Using Coccinelle," in *Proc. Europ. Dependable Computing Conf. (EDCC)*. IEEE, 2015, pp. 132–143.

[28] F. Thung, X. D. Le, D. Lo, and J. Lawall, "Recommending Code Changes for Automatic Backporting of Linux Device Drivers," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 222–232.

[29] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches," in *Proc. USENIX Security Symposium (USS)*. USENIX Association, 2022, pp. 1993–2010.

[30] G. K. Michelon, W. K. G. Assunção, P. Grünbacher, and A. Egyed, "Analysis and Propagation of Feature Re-

visions in Preprocessor-based Software Product Lines," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, T. Zhang, X. Xia, and N. Novielli, Eds. IEEE, 2023, pp. 284–295.

[31] P. M. Bittner, A. Schultheiß, S. Greiner, B. Moosherr, S. Krieter, C. Tinnes, T. Kehrer, and T. Thüm, "Views on Edits to Variational Software," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 141–152.

[32] T. Landsberg, C. Dietrich, and D. Lohmann, "Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Sep. 2024, to appear.

[33] P. M. Bittner, A. Schultheiß, B. Moosherr, T. Kehrer, and T. Thüm, "Variability-Aware Differencing with DiffDetective," in *Companion Proc. Int'l Conference on the Foundations of Software Engineering (FSE Companion)*. ACM, 2024, pp. 632–636.

[34] L. Linsbauer, T. Berger, and P. Grünbacher, "A Classification of Variation Control Systems," in *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2017, pp. 49–62.

[35] L. Linsbauer, F. Schwägerl, T. Berger, and P. Grünbacher, "Concepts of Variation Control Systems," *J. Systems and Software (JSS)*, vol. 171, p. 110796, 2021.

[36] S. Ananieva, S. Greiner, T. Kehrer, J. Krüger, T. Kühn, L. Linsbauer, S. Grüner, A. Koziolek, H. Lönn, S. Ramesh, and R. H. Reussner, "A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications," *Empirical Software Engineering (EMSE)*, vol. 27, no. 5, p. 101, 2022.

[37] S. Ananieva, T. Kühn, and R. Reussner, "Preserving Consistency of Interrelated Models During View-Based Evolution of Variable Systems," in *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2022, pp. 148–163.

[38] S. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski, "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System," in *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 323–333.

[39] E. Walkingshaw and K. Ostermann, "Projectional Editing of Variational Software," in *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2014, pp. 29–38.

[40] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO Tool: Extraction and Composition for Clone-and-Own," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2015, pp. 665–668.

[41] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability Extraction and Modeling for Product Variants," *Software and Systems Modeling (SoSyM)*, vol. 16, no. 4, pp. 1179–1199, 2017.

[42] F. Schwägerl and B. Westfechtel, "SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2016, pp. 822–827.

[43] J. Businge, M. Openja, S. Nadi, and T. Berger, "Reuse and Maintenance Practices Among Divergent Forks in Three Software Ecosystems," *Empirical Software Engineering (EMSE)*, vol. 27, no. 2, p. 54, 2022.

[44] P. K. Ramkisoen, J. Businge, B. van Bladel, A. Decan, S. Demeyer, C. D. Roover, and F. Khomh, "PaReco: Patched Clones and Missed Patches Among the Divergent Variants of a Software Family," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 646–658.

[45] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *Proc. IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 48–62.