# PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family


Poedjadevie Kadjel Ramkisoen
poedjadevie.ramkisoen@uantwerp.be
University of Antwerp
Belgium

John Businge
john.businge@uantwerpen.be
University of Antwerp
Belgium

Brent van Bradel
Brent.vanBladel@uantwerpen.be
University of Antwerp
Belgium

Alexandre Decan
alexandre.decan@umons.ac.be
University of Mons
Belgium

Serge Demeyer
serge.demeyer@uantwerpen.be
University of Antwerp
Belgium

Coen De Roover
Coen.De.Roover@vub.be
Vrije Universiteit Brussel
Belgium

Foutse Khomh
foutse.khomh@polymtl.ca
Polytechnique Montréal
Canada



## ABSTRACT

Re-using whole repositories as a starting point for new projects is often done by maintaining a variant fork parallel to the original. However, the common artifacts between both are not always kept up to date. As a result, patches are not optimally integrated across the two repositories, which may lead to sub-optimal maintenance between the variant and the original project. A bug existing in both repositories can be patched in one but not the other (we see this as a missed opportunity) or it can be manually patched in both probably by different developers (we see this as effort duplication). In this paper we present a tool (named PaReco) which relies on clone detection to mine cases of missed opportunity and effort duplication from a pool of patches. We analyzed 364 (source→target) variant pairs with 8,323 patches resulting in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. We achieve a precision of 91%, recall of 80%, accuracy of 88%, and F1-score of 85%. Furthermore, we investigated the time interval between patches and found out that, on average, missed patches in the target variants have been introduced in the source variants 52 weeks earlier. Consequently, PaReco can be used to manage variability in "time" by automatically identifying interesting patches in later project releases to be backported to supported earlier releases.




*ESEC/FSE 2022, 14 - 18 November, 2022, Singapore*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



## KEYWORDS

github, clone&own, variants, software family, forking, social coding, bug fixes, effort duplication, clone detection

**ACM Reference Format:**

Poedjadevie Kadjel Ramkisoen, John Businge, Brent van Bradel, Alexandre Decan, Serge Demeyer, Coen De Roover, and Foutse Khomh. 2022. PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


## 1 INTRODUCTION

Code reuse is the practice of using existing code to speed up the development process. "Traditional" code reuse is performed by declaring a dependency towards another library or another package [15]. An alternative code reuse is the "clone&own" paradigm [9–11, 31, 45]. One would opt for the paradigm of "clone&own" over the "traditional" code reuse because the involved projects have traceability links and easily share new updates.

The "clone&own" paradigm is a commonly adopted approach for developing multi-variant software systems, where a new variant of a software system is created by copying and adapting an existing one and the two continue to evolve in parallel [9–11, 31, 45]. As a result, two or more software projects will share a common code base as well as independent, project-specific code. The multi-variant software systems are referred to as a *software family*, or *family* in short [10, 11]. With an increasing number of variants in the family, development becomes redundant and maintenance efforts rapidly grow [7, 17, 40, 48]. For example, if a bug is discovered and fixed in

one variant, it is often unclear which other variants in the family are affected by the same bug and how this bug should be fixed in these variants. Although clone&own development paradigm has limitations, studies have reported their prevalence on social coding platforms like GitHub [9, 11].

This study aims to empirically quantify the extent to which *divergent variants* exhibit redundancy and missed essential updates concerning bugfixes. Therefore, we present a tool (named PaReco) that can support the maintenance of divergent variants. PaReco mines possibly interesting bugfixes (patches) from a pool of updates in a source variant and relies on clone detection to classify the patches as redundant, missed, or uninteresting in the target variants.

To the best of our knowledge, this is the first large-scale study on automatically identifying (and recommending) relevant bug fixes to developers of "clone&own" variants. Our contributions are threefold. (1) We analyzed 364 (source→target) variant pairs and validated the tool's output. This results in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. The curated datasets can be accessed in our replication package [3]. (2) We quantify how many cases of effort duplication and missed opportunities exist between divergent variants. Next, we investigated the time interval between such patches to assess the window of opportunity for relevant bug fixes. (3) We developed PaReco which can be used as-is to support the management of variability in "space" (concurrent variations of the system at a single point in time). This can be achieved through mining interesting patches from one variant (source) and classify the patches as interesting or not interesting to the target variants. PaReco can also be configured to manage variability in "time" (sequential variations of the system due to its evolution). This can be achieved by automatically identifying interesting patches in later project releases that can be backported to earlier releases of the project. To tool is available and released under an open-source license [39].

## 2 TERMINOLOGY, PROBLEM, AND CONCRETE EXAMPLES

In this section, we provide an overview of the problem, define the terminology used, and show a concrete example.

### 2.1 Terminology

Having explained the problem overview, let us now give the formal definitions of the terms used in our study.

- **`current_date`.** The date when we collected the dataset for this study on `2021-08-06`.
- **`divergence_date`.** The date after the last synchronization of variants. This is determined using the GitHub API. Each variant pair has its own `divergence_date`.
- **hunk.** A hunk is a grouping of differing lines between two versions of a file [19]. A hunk is written in the format `@@ -l,s +l,s @@` with `l` the starting line number, `s` the number of lines the change applies to for each respective file, `-` indicating the original file and the `+` indicating the new (modified) file.
- **buggy file.** This is a file containing buggy lines before the pull request to fix the bug is created.
- **patched files.** These are files that are integrated back into the main development branch at the pull request integration with buggy lines removed and new ones added.
- **diff_file.** The resulting file after applying the `diff` tool [18] on the buggy and the patched file. It contains both the removed lines from the buggy file and added lines in the patched file.
- **patch.** A patch is a collection of one or more diff_files. In our study, we specifically refer to a patch when this collection of diff_files stems from a pull request that was created to fix a bug.
- **`git_head` file.** The latest version of a file retrieved from the `git_head` on the `current_date` in the main branch of the target repository.
- ***social fork.*** These are forks that are created for isolated development to fix a bug, feature, refactoring and thereafter merged back into the mainline [50].
- ***divergent variant.*** A variant fork is created by splitting off a new development branch to steer development into a new direction while leveraging the code of the mainline project [9, 11]. Variants in divergent variant pairs contain unsynchronized commits between themselves. We use the GitHub API to identify unsynchronized commits in a pair, where one variant is ahead by $X$−commits and behind by $Y$−commits. In the unsynchronized commits we do not go deeper to identify the commits that are integrated using techniques that change the commit ID [14].

### 2.2 Classifying patches - Illustration

Figure 1 is an illustration of clone&own, where `variant2` (forked repository) was forked from `variant1` (original repository). When `variant2` was created (`fork_date`), it inherited all commits from `variant1`. Then, between the `fork_date` and `divergence_date`, both variants synchronised commits with each other, keeping both variants even. After the `divergence_date`, the variants stopped synchronizing commits. As a result, all commits after the `divergence_date` are unique to the respective variant.

Let us assume that the developer of `variant1` identified a bug after the `divergence_date` spread across files `foo`, `bar`, and `lot`. The developer decided to create a *social fork* of the source repository, patched the buggy files, and finally integrated the patch back into the main branch of the source repository using a pull request. There are four possible scenarios on the `git_head` of `variant2`:

(1) The developer of `variant2` could have patched the buggy file in one of the previous commits before the commit at the `git_head`. This is a case of effort duplication (ED).
(2) The files at the `git_head` of the target still contains the buggy lines. This is a case of missed opportunity (`MO`). We can even calculate how long the target branch has missed the patch by calculating the distance between the patch integration date and the current date of the `git_head`.
(3) The file at the `git_head` of the target contains both the buggy and the patched lines. In this case both effort duplication and missed opportunity are present. This is a split case (`SP`).
(4) The file at the `git_head` of the target does not contain both the buggy and the patched lines. This case would be interesting (`NI`).

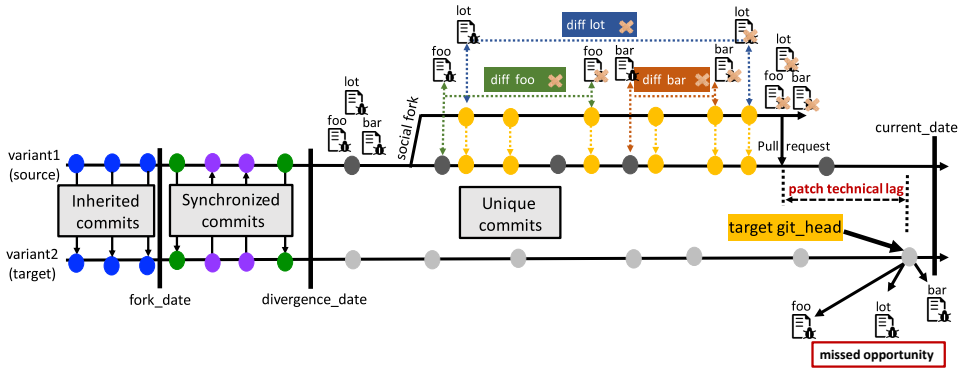


**Figure 1: Illustration of the patch classification from source to target variant.**

> This scenario could happen when the developer of `variant2` has replaced the buggy lines with new lines, but the tool we use is not able to recognize these new lines as clones.

Note that for illustration purposes, we only present an example where a patch is `variant1` as source and `variant2` target. However, in the above example `variant1` and `variant2` are interchangeable.

## 2.3 Motivating example

To put the terminologies and problem into perspective, we present one concrete motivating example of a missed opportunity (MO). Due to space limitations, examples of effort duplication and split (SP) cases can be found in the online appendix [3]. In this example, `variant1` represents the source (upstream) – `qmk/qmk_firmware`, and `variant2` represents the target (fork) – `sekigon-gonnoc/qmk_firmware` that are hosted on GitHub. The variants have diverged between `2018-10-05` and `2021-08-06` (`current_date`), adding `5,315` and `117` unique commits in `variant1` and `variant2`, respectively. One of the upstream pull requests (number `12587`) contains one commit `cc0a5f0` with one file. The file `tmk_core/common/chibios/eeprom_teensy.c` was changed to fix a `gcc10 build warning` for `issue #12587`. The pull request was merged on `2021-04-20`.

Listing 1 shows the buggy code of `variant1` [20]. Listing 2 shows the patched code that fixed the bug in `variant1` [21]. Listing 3 shows the `diff_file`. Listing 4 shows the latest version of that code in `variant2` at `2021-07-16`.

**Listing 1: Buggy lines in file `eeprom_teensy.c` at commit `3a5afd8` of the source repository**

```
1                 return;
2             }
3         } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
4         flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1);
5     }
```

**Listing 2: Patched lines in `eeprom_teensy.c` at commit `cc0a5f0` of the source repository**

```
1                 return;
2             }
3         } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
4         flashend = (uint32_t)(p - 1);
5     }
```

**Listing 3: Diff file for PR–1287 from the source repository**

```
1   @@ -363,7 +363,7 @@
2
3         } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
4   -     flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1)
        ;
5   +     flashend = (uint32_t)(p - 1);
```

**Listing 4: Unpatched lines in file `eeprom_teensy.c` in the `git_head–1200fa9` of target repository**

```
1                 return;
2             }
3         } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
4         flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1);
5     }
```

We can see that the code in `variant2` is identical to the buggy code in `variant1`. However, the fixed line is not found in the `git_head` file of `variant2`. This means that the bug is still present in the forked repository, even though a patch exists in the source repository that fixes the bug. We classify this as a missed opportunity (MO) in `variant2`, because the patch that fixed the bug in `variant1` is not applied to `variant2`. The patch however, can be applied to `variant2` to fix the bug that is still there.

The developers of `variant2` may have missed the patch (PR `12587`) applied in `variant1` since the patch is buried in a pool of changes that could be interesting. Furthermore, we have shown a straightforward `MO` example for illustration purposes that one can easily observe. However, patches can range from easy to complex. For example, a pull request may contain many commits, files, and changed lines. In a study very related to ours, Jang et al. [25] state that finding all unpatched code clones is tricky and involves numerous considerations. For example, how many lines of code need to be similar for a case to be reported? Is one copied line enough, or are we only interested in multiple line matches? Should whitespace matter? Should the order of statements matter, and if so, should we only consider some syntactic classes? Jang et al. [25] created the clone detection tool `ReDeBug` to find unpatched code clones (`MO`) in OS distribution-sized code bases (> 1 billion LOC) that include code written in many different languages. `ReDeBug` is a lightweight syntax-based code clone detection tool that identifies unpatched code clones at scale.

# 3 STUDY DESIGN

Our overarching goal is to understand the extent of patch redundancy, and how many divergent variants miss important patch.

## 3.1 Research Questions

RQ1 *How many cases of effort duplication and missed opportunities exist between divergent variants?* Since the variants are divergent, effort duplication implies that variant developers could be independently fixing common bugs. In the case of SP, only a part of the patch is implemented in the target, while for MO, the target is still buggy. This RQ aims at finding out the prevalence of these cases in variant pairs. Target variant developers can use the patches of MO and SP as a starting point to fix the buggy code in their variants. The cases of ED can be used in follow-up studies to investigate (how?) and (why?) target variant developers clone the patches even though the variants have diverged.

RQ2 *How much patch technical lag exists between the source and target variants in divergent variants?* This RQ is a follow-up of RQ1 on the missed patches (i.e., MO/SP). We would like to understand how long the patches introduced in the source variants and classified as MO and SP in the targets go unnoticed by the target variants. Gonzalez-Barahona et al. [22] proposed the concept of technical lag to reflect how outdated a software system is, concerning its upstream dependencies. In this study, we define a new technical-lag-based metric called *patch technical lag* that measures how outdated a target variant is concerning the applied patches of MO and SP in the source variant. We aim to get a better understanding of patch technical lag in the software families. The insights will help manage and control patch technical lag, through tools designed to monitor and recommend the missed patches as soon as they are introduced.

## 3.2 Data Collection

**Step 1** *Divergent variant pair identification:* To identify divergent variant pairs, we leveraged the dataset from Businge et al. [9, 11] who report a total of over 1.5K variant pairs. We were interested in actively maintained divergent variant pairs. In our first filter we retained variant pairs that were updated at least not earlier than six months back from the `2021-08-06`. We next retained included pairs written in `Java`, `C`, `PHP` or `Ruby`, as these are the programming languages that our clone detection tool can process. To ensure that we have divergent variant pairs, we also applied another filter to select pairs where there was at least six months between `divergence_date` and the earlier of the two dates of `variant1 update_date` and `variant2 update_date`. Finally, since we identified the patches from pull requests, we ensured that at least one of the variants in the pair had merged one pull request. After all the filtering, we retained a total of **182 diverged variant pairs**. Moreover, since source and target can be interchanged, we can extend this to a total of **364 source→target pairs**.

We wanted to compare the diverged commits of the 182 mainline–variant pairs. To this end, we collected unsynchronized commits in the variant pairs. The boxplots in Fig. 3 show the distributions by the programming languages we considered. While it is not surprising that the number of commits to the mainline is always higher than

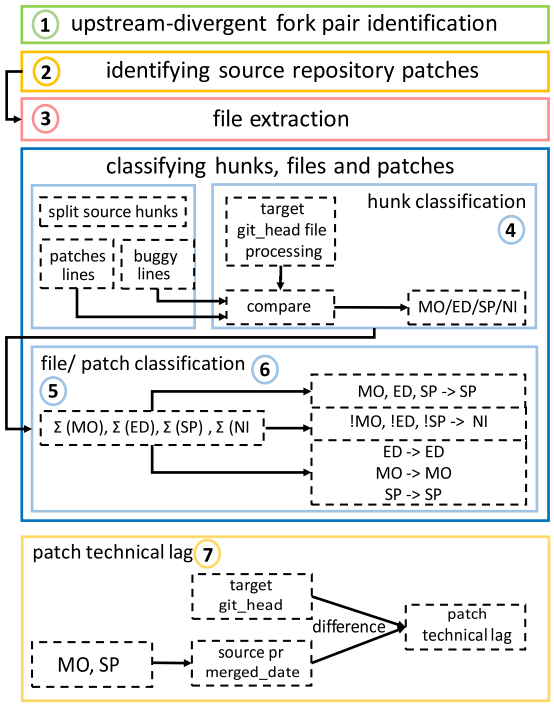


**Figure 2: Method overview.**

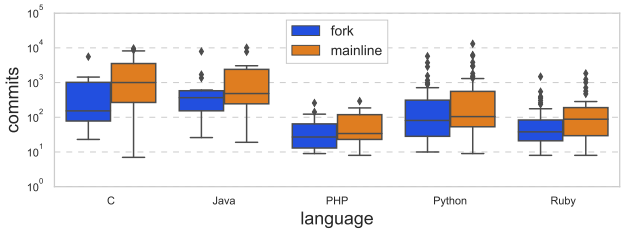


**Figure 3: Distribution of diverged commits in the variant pairs; categorized in the different programming languages.**

to the fork, it is interesting that most forks also have a pretty high number of commits. This gives us confidence that we are studying real variants as opposed to social forks.

**Step 2** *Source Repository Patch Identification:* To identify source repository patches, we mine the pull requests (PRs) that were integrated between the `divergence_date` and the `current_date` (see Fig 1). We specifically look for PRs that contain keywords related to bug-fixes, such as *fix*, *resolve*, *addresses*, and *crash*. The full list of keywords is described in previous studies [13, 38, 41]. The bug fixing keywords have been manually validated by Castelluccio et al. [13] achieving a precision of 87.3% and a recall of 78.2%, respectively.

**Step 3** *File Extraction:* We extract the *buggy*, *patched*, and *diff* files from the PR commits in the source repository (see Section 2.3 and Fig 1). Then we extract the corresponding files at the `git_head` of the target variant. To find the correct code to extract, we reuse

and extend the clone detection tool ReDeBug [25]. First, for every modified file in the PR of the source repository, our tool will find the corresponding file at the `git_head` in the target repository. We identify the files at the `git_head` by comparing paths [1] of the PR commits and the files at the `git_head`. We only consider the file paths that match. For simplicity, during the file extraction at the `git_head`, our tool currently considers the deleted, moved, and renamed files as missing files. The tool then performs normalization (i.e., removes language comments, removes all non ASCII characters, removes redundant whitespaces except new lines, and converts all characters to lower case). Next the tool tokenizes the extracted files from both the source and the target variants. Representing source code as a token sequence enables the detection of clones with different line structures, which cannot be detected by the line-by-line algorithm. Tokenization is performed using `n-grams`, where we maintain the `n = 4` as used in the original tool. In the RedeBug tool, `n-grams` are computed based on lines. For every set of 4 consequent lines of code, one `n-gram` is created. For example, for a file with 5 lines of code, two `n-grams` of size 4 are considered. After tokenization, the `diff` file in the source repository is extracted. Since ReDeBug only cares about unpatched code clones, the tool was implemented to detect in the buggy snippets (missed opportunities). The tool performs a clone detection between the source and the target to determine if the buggy lines in the buggy file are still present in the same file in the target. In this study we care about both the patched (effort duplication) and unpatched (missed opportunity) code clones. Therefore, in addition to the `patched` snippet identification performed in ReDeBug, our tool compares the source and target to determine if the target contains patched lines that are present in the `diff` file of the source variant (ED).

**Step 4** *Hunk classification:* To classify a hunk, our tool checks for a code snippet with patched lines or the buggy lines in the commit at the `git_head` of the target. We perform this check by comparing the tokens computed from 4-grams generated by ReDeBug. If the snippet contains only buggy lines, then PaReco classifies the hunk as MO. If the snippet contains patched lines, then PaReco classifies it as ED. If the snippet contains both buggy and patched lines then PaReco classifies it as SP. The snippet is classified as NI if PaReco cannot find both the buggy and patched lines.

**Step 5** *File classification.* At this step, we aggregate the code snippets (hunks) classifications into the classification of files. In addition to the hunks' classes, files in the target can take on two other classes: (i) cannot classify (CC) class is where a pull request contains only files written in programming languages that the clone detection tool cannot process. (ii) non-existent (NE) class is where the pull request contains files missing in the target variant. If a file contains more than one hunk from different classification, we first focus on the hunks in the interesting classes of MO, ED, and SP during the file classification. In any given file of interest in the target variant, if we can identify at least one hunk classified as MO, ED, and SP, we assume that the associated patch in the source variant could be interesting to the target. To this end, we classify a file according to the most prevalent hunk class of MO, ED, and SP. For example, if a file has 10 hunks, one is classified as MO, and the remaining nine hunks classified as NI, CC, or NE, then the file is classified as MO. SP class prevails as the file classification in the case of ties between MO / ED / SP. In case none of MO / ED / SP is present, then the file takes on the class of the most prevalent class of NI / CC / NE.

**Step 6.** *Patch classification.* The patch classification is an aggregation of the file classifications. Like for files in Step 5, if a patch contains more than one file, we first focus on the interesting classes of MO, ED, and SP. We then follow the same criteria discussed in Step 5, to aggregate a patch from its classified files.

**Step 7**. *Patch technical lag calculation.* We calculate patch technical lag as the elapsed time from when a developer integrates a patch into the main development branch of the source variant and the time of the commit at the `git_head` of the target variant that still contains buggy lines (see illustration in Figure 1). The time when the patch was integrated in the source variant can easily be determined using the GitHub API v3 to extract the pull request `merge_date`. In this RQ, we only considered target variants that had at least one patch classified as either MO or SP. We identified a total of 97 target variants having a total of 1,109 patches classified as MO or SP.

## 4 RESULTS & DISCUSSION – RQ1

*How many cases of effort duplication and missed opportunities exist between divergent variants?*

*RQ*1 aims to investigate how often there are patches classified as MO, ED, or SP among the 364 variant pairs in our dataset. Here we present the results of the patch classification using PaReco and the accuracy of the patch classification. We validate the accuracy of PaReco manually in Section 4.2.

### 4.1 Tool patch classifications

We applied PaReco on the 8,323 patches identified in 364 source variant repositories. Table 1 shows how the 8,323 patches from the 182 diverged variant pairs were classified. We present the aggregate statistics to show the distribution of the patch classifications in the target repositories. The upper part presents the patch classification where source variant is the upstream repository and the target variant is the forked repository. The lower part of the table presents the statistics with the source and target interchanged between upstream and fork.

Figure 4 presents grouped boxplots for the statistics in Table 1. The plot shows how the patches from the source variants are shared among the different patch classes in the target variants. Each class is represented by two boxplots. The boxplots on the left of each class represent the results in the upper part of Table 1 and those on the right represent the results in the lower part of Table 1.

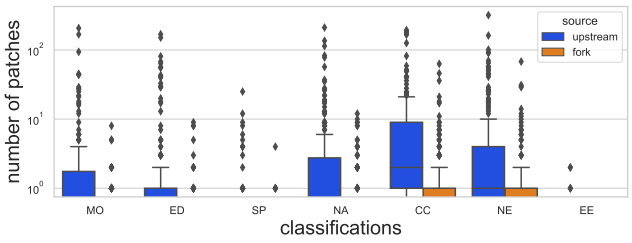


**Figure 4: Distribution of the patch classification.**

**Table 1: Summary statistics for patches identified in the source variant and classified in the target variants.**

| Metric | Mean | Min | Median | Max | Total |
|---|---|---|---|---|---|
| **upstream (source), fork (target)** | | | | | |
| Missed opportunity (MO) | 5.3 | 0 | 0 | 207 | 957 |
| Effort duplication (ED) | 5.8 | 0 | 0 | 169 | 1,062 |
| Split (SP) | 0.5 | 0 | 0 | 25 | 92 |
| Not Applicable (NI) | 6.6 | 0 | 0 | 211 | 1,198 |
| Cannot classify (CC) | 12.6 | 0 | 1 | 194 | 2,292 |
| Not existing file (NE) | 9.7 | 0 | 2 | 319 | 1,773 |
| Error (EE) | 0 | 0 | 0 | 2 | 7 |
| Patches | 40.6 | 0 | 1 | 629 | 7,381 |
| **fork (source), upstream (target)** | | | | | |
| Missed opportunity (MO) | 0.3 | 0 | 0 | 8 | 51 |
| Effort duplication (ED) | 0.3 | 0 | 0 | 9 | 54 |
| Split (SP) | 0.1 | 0 | 0 | 4 | 9 |
| Not Applicable (NI) | 0.6 | 0 | 0 | 12 | 120 |
| Cannot classify (CC) | 2.1 | 0 | 0 | 63 | 378 |
| Not existing file (NE) | 1.8 | 0 | 0 | 68 | 330 |
| Error (EE) | 0 | 0 | 0 | 0 | 0 |
| Patches | 5.2 | 0 | 0 | 132 | 942 |

From the results of Table 1 and Figure 4, we can see that most patches originated from the upstream variants; only few patches originated from the forked variants. We also observe that most patches are classified in the negative classes of NI, CC, and NE. This shows that the majority of the patches contain NI files, CC files, or NE files. However, we do observe a good number of patches that have been classified in the interesting / positive classes of MO, ED, and SP: a total of 2,225 of the 8,323 (26.7%) patches.

When considering the upstream repository as the source and the fork as the target (i.e., upper part of the table), we observe that 72 of the 182 (39.5%) variant pairs contain 957 of the 2,225 (43%) patches that are classified as MO. We also observe that 60 of the 182 (33%) variant pairs contain 1,062 of the 2,225 (47.7%) patches that are classified as ED. Finally, we observe that 21 of the 182 (11.5%) variant pairs contain 92 of the 2,225 (4.1%) patches that are classified as split cases. When the source and the target are interchanged to fork→upstream (i.e., lower part of the table), we observe that 26 of the 182 (14%) variants contain a total of only 114 of the 2,225 (5%) patches classified as positive classes of MO, ED, and SP).

We observe that very few fork variants integrate patches that could be interesting to the upstream variants from the results. This is not surprising since when we look back in Fig. 3, we can see that the upstreams have more updates than their fork counterparts. We also observe that most patches are classified as uninteresting classes of (NI), CC, and NE. Our follow-up studies will extend PaReco to process files in more programming languages to gain interesting patches from the CC class. We will also extend PaReco to extract renamed and moved files from the target variant to gain interesting patches from the NE class. The curated dataset for Fig. 4 can be found in the online-appendix [3].

Recall that in Section 3.2–Step 6 we stated that a patch is classified MO, ED, or SP if it contains at least one file classified as interesting.

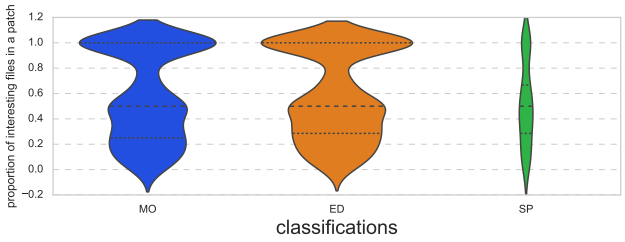


**Figure 5: Distribution of interesting files in the patches. For example, a patch on the ED box plot with a y-axis value of 1.0, indicates that all the files in the patch were classified ED.**

Figure. 5 contains violin plots that show the distribution of patches concerning the proportion of interesting files in each patch. For example, a point on the violin plot with x-axis label ED and a value close to zero on the y-axis indicates that the patch has many files with the most significant proportion classified as one of NE, NI, or CC and a tiny proportion classified as ED. The size of the plots indicates the number of data points in the category. Focusing on the bigger-sized violin plots of ED and MO we can see the upper quartiles are both on 1.0. However, we can also see that $Q_4 - Q_3 > Q_2 - Q_1$, indicating that the plots are positively skewed. This implies that ED and MO classes have more patches above the median, where the proportion of interesting files in the patches is high. Details of Fig. 5 can be found in the online-appendix [3].

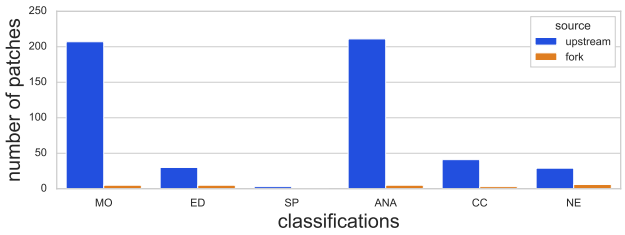


**Figure 6: All classifications of patches considered for spack/spack (upstream) and BlueBrain/spack (fork).**

**Concrete example.** The variant pair spack/spack (upstream) and BlueBrain/spack (fork) are both projects package manager for supercomputers. The upstream is owned by Spack [44] while the fork is owned by the Blue Brain[6]. BlueBrain/spack was forked on 2018-03-12. The two repositories diverged on 2020-12-06. As of 2022-02-11, spack/spack had 1,150 unique commits of 24,706 commits, while BlueBrain/spack had 1,657 unique commits of 25,213 commits.

Figure 6 presents the results of patch classifications in the form of a grouped bar plot of the variant pair. The first two bars labelled MO, the first bar shows the number of patches in the source spack/spack (source), that are classified as MO in the target BlueBrain/spack (target). The second bar (short) on MO shows the results when the source and target interchanged. We can see that in the variant pair shown in Figure 6 there are more cases of MO than ED.

***Summary: Tool patch classifications**: Of the 2,225 interesting patch classifications, the most prominent class is effort duplication (ED); comprising a total of 1,116 (50.2%) patches. Next is missed opportunity (MO), comprising a total of 1,008 (45%) patches. Finally, we identify a total of 101 (5%) split cases (SP), containing both MO and ED hunks. We have also shown that most interesting patches have the upstream repository as the source variant. There are very few cases having the fork as the source variant.*

## 4.2 Accuracy of patch classifications

We use six metrics to evaluate the accuracy of our tool: true positives (TP), false positives (FP), false negatives (FN), precision ($\frac{\#TP}{\#TP+\#FP}$), recall ($\frac{\#TP}{\#TP+\#FN}$) and the F1-score. We manually analyse these metrics on the classification results of the 8,323 patches.

**Criteria for ground-truth establishment.** To find the real classifications of a patch, we mimic the behaviour of our tool manually. For a given patch, we find and extract all the `diff` files from the source variant and the corresponding files in the commit at the `git_head` of the target variant. For each hunk in each `diff` file, we manually look at the hunk lines and search if they exist in the file at the `git_head` of the target variant. We do this as follows: We search for the first line of the hunk in the target variant. If a match is found we go to the next line in the hunk and continue until we find a line prefixed with "+" or "-", i.e., the added patched lines or the deleted buggy lines. The other lines are called the context. It is important to note that all other matches that we find in the file of the target variant should be in the same order as listed in the hunk. If we only find the buggy lines in the file of the target variant, then we classify it as MO; if we find the patched lines only, we classify it as ED; if we find both, then we have a SP, otherwise it is an NI. Once all the hunks have been manually classified, we follow the criteria in Section 3.2–Step 5 to classify the files and Step 6 to classify the patches. For the files written in a language that our tool cannot process, we verify if the extension of that file indeed falls out of the scope of our tool, and for the files that are found to not exist in the target, we open the snapshot of the target in GitHub and check if that file indeed does not exist.

**Accuracy of measurement.** We calculated a sample from our population of 8,323 patches using an online sample size calculator [2]. We used the following parameters: confidence level = 95%, margin of error = 5%, population proportion of = 26% / 74% (corresponding to 2,225–26% for positive cases and 6,098–74% negative cases). From this we determined a sample size of 286 cases for the manual analysis. The manual analysis was conducted by the first author of this paper and the second author validated the labels.

However, while conducting the manual analysis, the author observed that big patches containing many files / hunks and a lot of changed lines of code took very long and yet, they were not obvious to decide. To mitigate the challenge, we decided to filter out the complex cases. We excluded patches that had changes in more than five files, more than 15 lines added, and more than 15 lines removed. Using this criteria we filtered out 1,794 complex case patches (21.6%) from the total population of patches. The new population of 6,528 patches contained 1,632 (25%) positive cases (MO, ED, SP) and 4,896 were negative cases (NI, CC, NE). The new sample size using the population proportion of 25% while maintaining the values of the other parameters becomes 276 patches.

We group the results on two levels of classification:

(1) *First level.* Contains the interesting cases (MO, ED, and SP) representing the positive and uninteresting cases representing the negative cases (NI, CC, and NE). We also present the classification results for the three levels of granularity of the bug fix (i.e., patch, file, and hunk). So, for example, if a patch is classified by the tool as any of MO, ED, and SP (predicted, positive) and during the manual classification, we find out that the patch is any of NI, CC, and NE (actual, negative), then we label the case as false positive (FP).
(2) *Second level.* Here we go deeper into the positive cases to see if they were correctly classified. For example, if a patch is classified by our tool as MO (predicted, positive) and during the manual analysis, we observe that it is supposed to be ED (actual, negative). Therefore, we label the case as false positive (FP).

In Table 2 we present the confusion matrix and in Table 3 we present the precision, recall, and accuracy for the *First level* of classification. From Table 3 we observe a precision of > 89%, a recall of > 74%, and an accuracy of > 86% for all the levels of granularity. This means that our tool is good at correctly classifying both the positive interesting classes and the negative uninteresting classes.

**Table 2: Confusion matrix for the classifications on different levels after manual validation of the results of PaReco. (+) – positive cases and (-) – negative cases.**

| | | Predicted | | | | | |
|---|---|---|---|---|---|---|---|
| | | Patch | | Files | | Hunks | |
| Actual | + | 81 | 20 | 103 | 36 | 149 | 46 |
| | - | 8 | 167 | 10 | 353 | 17 | 113 |
| | | + | - | + | - | + | - |

**Table 3: Precision, recall and accuracy for the different classification levels for the results in Table 2.**

| | Precision | Recall | Accuracy | F1-score |
|---|---|---|---|---|
| Patches | 91.0% | 80.2% | 88.0% | 85.3% |
| Files | 91.1% | 74.1% | 86.3% | 81.7% |
| Hunks | 89.8% | 76.4% | 89.4% | 82.6% |

Table 4 shows the confusion matrix for sampled data from our dataset on the *Second level*, for the MO, ED, and SP classes at the three different levels of granularity (patch, file, and hunk). Table 5 shows the precision, recall and accuracy corresponding to the results in Table 4. For both MO and ED we observe relatively high values of precision, recall, and accuracy (all ≥ 84%). Our validation results are comparable to the study of Kim et al. [30] who validated ReDeBug vulnerable code clone discovery (ED in our case) and reported a precision of 85%. This gives us confidence that our tool is relatively good at identifying interesting patches. Looking at the results of the SP cases, we observe low values for the precision and recall at the granularity levels file and hunk, yet those of the patch granularity are all 100%. The low values originate from the imbalanced data, since we have few positive cases and many negative cases. This issue of data imbalance is visible from the high

accuracy for SP at all levels of granularity. The dataset for SP is biased towards the most prominent class (the negative class), which is the reason for the low levels of precision and recall. From the results presented, we conclude that our tool is relatively good at performing patch classifications.

**Table 4: Confusion matrix for interesting classifications.**

| | | Predicted | | | | | |
|---|---|---|---|---|---|---|---|
| | | MO | | ED | | SP | |
| | | Patch | | | | | |
| Actual | + | 34 | 1 | 40 | 4 | 3 | 0 |
| | - | 4 | 50 | 6 | 39 | 0 | 86 |
| | | Files | | | | | |
| | + | 42 | 3 | 52 | 2 | 1 | 2 |
| | - | 8 | 126 | 7 | 118 | 2 | 174 |
| | | Hunk | | | | | |
| | + | 55 | 9 | 60 | 9 | 10 | 2 |
| | - | 7 | 227 | 12 | 217 | 15 | 271 |
| | | + | - | + | - | + | - |

**Table 5: Precision, recall, accuracy, and F1-score for confusion matrix in Table 4.**

| | Precision | Recall | Accuracy | F1-score |
|---|---|---|---|---|
| | MO | | | |
| Patches | 89.5% | 97.1% | 94.4% | 93.1% |
| Files | 84.0% | 93.3% | 93.9% | 88.4% |
| Hunks | 88.7% | 85.9% | 94.6% | 87.3 |
| | ED | | | |
| Patches | 87.0% | 90.9% | 88.8% | 88.9% |
| Files | 88.1% | 96.3% | 95.0% | 92.0% |
| Hunks | 83.3% | 87.0% | 93.0% | 85.1% |
| | SP | | | |
| Patches | 100% | 100% | 100% | 100% |
| Files | 33.3% | 33.3% | 97.7% | 33.3% |
| Hunks | 40.0% | 83.3% | 94.3% | 54.0% |

**Incorrect Classifications.** During the manual analysis we identified 98 misclassified hunks. After analyzing the incorrect classifications, we summarized them into four categories and indicate if the issue is resulting from the *original code* or our *extended code*.

(1) *Hunk classification failure – extended code (44 cases):* For various reasons we found that some hunks were misclassified.

(2) *One line hashing – original code (24 cases):* During the manual analysis, we observed that some hunks were divided into n-grams of size one instead of four, resulting in misclassification.

(3) *Issues with comments – original code (22 cases):* In some hunks, we noticed that Python and C comments that are supposed to be ignored were also hashed and considered in the matching. Multi-line comments that are only partially present in the hunk were not removed by the regular expressions used for this during normalization.

(4) *File extension misunderstood – original code (4 cases):* PHP is one of the programming languages that our tool can process. We noticed in four misclassified cases that the tool could not guess the MIME type of the PHP files to be PHP, causing the patches to be classified as CC.

In follow-up studies these issues will be analysed critically so as to have a minimal misclassifications as possible. We have already submitted issues to the ReDeBug repository in GitHub regarding misclassifications. We are also planning to address the issues and we shall submit a pull request to the repository when we are confident that the issue has been solved in our tool.

***Summary: Accuracy of patch classifications**: Although PaReco classified only 26% as interesting patches from the total patches. The validation exercise reveals relatively high results precision, recall, accuracy, and F1-score for patch classifications of 91%, 80%, 88% and 85%, respectively.*

## 4.3 Discussion & Implications

RQ1 focused on automatically identifying a patch in one of the source repositories (variant1 or variant2) and then correctly classifying the patch as interesting or as not interesting. In Section 4.1 we already discussed that PaReco misses a lot of interesting patches that PaReco classified as NI, NE, CC. We also discussed how we plan to extend PaReco to identify more interesting patches. As a preliminary analysis, we have analysed the patches in the CC category and observed other frequent languages such as JavaScript, Scala, Kotlin, JSON, C++, Yaml, and Rust. The tool can be extended to analyze these languages and, hence, gain more on the interesting patches that can be identified. Although our study is in the early stages, the results reveal that clone-and-own variant on GitHub exhibits redundant development and miss important updates. We believe that the study is in the right direction towards supporting the maintenance clone-and-own variant. We also believe that although PaReco is still a work in progress, developers can find PaReco useful in supporting the maintenance clone-and-own variants. Furthermore, follow-up studies will also focus on user studies with the developers regarding the tool's results. For example, a user study can reveal the threshold of the proportion of interesting files in a patch (c.f. Fig. 5) for a patch to be considered interesting.

***Actionable result**: PaReco as-is can be used by developers to manage clone-and-own variants. Follow-up studies will be focused on extending PaReco to a patch recommender tool.*

**Missed opportunity.** The results of MO are expected due to the diverged status of the variants. However, it does provide insights into the number of patches in the source variants that go undetected in the target variants. The results are interesting since they reveal that, as much as the variants have diverged, some interesting changes like patches in common files could be propagated to improve the quality of the variants. The previous studies on variants [9, 11] reported over 80% of the variants pairs have uncommon maintainers. We replicated the same experiment in this study and observed 89% of the variants pairs have uncommon maintainers. The results in this study are not surprising since we consider only variant pairs that have diverged. To this end, as a result of distinct development teams, the variant developers may be aware that there could be some interesting changes, but these changes are not easy to find in a pool of other changes.

**Split (SP).** These cases are also interesting since they reveal that the target variant only contains part of the patch applied in the

source variant and the other part is missed. In addition to the patch being incomplete in the target variant, the expertise of the developers fixing the patch might also vary. It is plausible that a less experienced developer fixed the incomplete patch. To this end, the developer of the target variant can make use of the split cases and fix the missing part of the patch in their repositories.

> ***Actionable result**: This study shows that there is great potential to automate the identification of those difficult to find changes that can be recommended to variant developers. Variant maintainers can already use* PaReco *as-is to uncover interesting patches from a source variant and integrate them into their repository.*

**Effort Duplication.** Since we study only divergent pairs, this implies that the majority of fork variants either re-implement patches already implemented in the upstream or they clone only the interesting patches from the upstream into their projects. In the case of cloning only the interesting patches, there are a number of ways this can be achieved: (1) copying the patch code snippets from the source repository and pasting it in the target; or (2) synchronizing the variant pairs using GitHub tools that change the commit ID during the integration, such as PR squash / rebase, and other git tools that completely change the commit history such as git squash. Patch integration using these tools would merge all the differences in the variants, which is not something that the variant developers would desire [29]. Moreover, the study of Businge et al. [11] revealed that the two techniques of PR squash / rebase are infrequently used techniques to integrate change between variants. To this end, it is likely that such integration techniques may be less prominent in explaining the many cases of effort duplication. Another method that can be used to clone the patches is (3) git cherry-picking, where the developer integrates only desired commits [12]. However, from the study of Businge et al. [11] it was observed that the cherry-picking was less frequently used as an integration technique. Furthermore, since we observe considerable numbers of MO, it is unlikely that the integration techniques that change the commit history are being used.

A follow-up study, engaging with the developers, could help reveal the frequently used to patch integration techniques and (why?) they are preferred. Developer engagement could also tell the developers' challenges in incorporating the patches. Furthermore, in Fig. 5 the ED cases with a low proportion of interesting files indicate that these patches are not exact but share some clones. The variable part of these patches can be a source for further investigation. For example, if different developers implement the same patch, one of the patches in the variant pair might be more elegant than the other. If this happens to be the case, the more elegant patch should be recommended to the other variant developer.

> ***Actionable result**: To avoid reinventing the wheel, developers can use* PaReco *to identify interesting patches in a family of variants.*

# 5 RESULTS & DISCUSSION - RQ2

*How much patch technical lag exists between the source and target variants in divergent variants?*

## 5.1 Results

Here we present the results of the patch technical lag in our target variant projects. Figure 7 is a line plot showing how much technical lag there is in each of the 1,109 patches classified as MO or SP in the target variants; it plots the technical lag in each patch. A point on the line plot represents a patch (x-axis) and how much technical lag in terms of weeks (y-axis). The median of the technical lag for these patches is 27 weeks (over half a year), which is relatively high. This implies that patches introduced in the source variant spend at least 27 weeks before they are noticed by the target variants. The median value could be more as of today, since the `current_date` as seen in Figure 1 is as of 2021-07-28 when this data was collected.

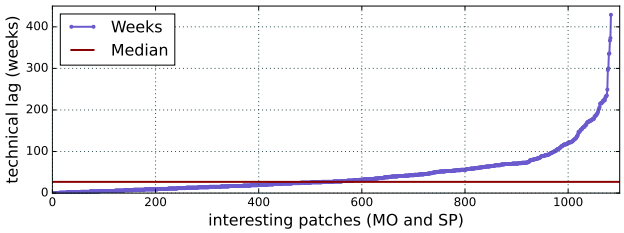


**Figure 7: A line plot showing the patch technical lag of the patches of MO and SP in the target variants.**

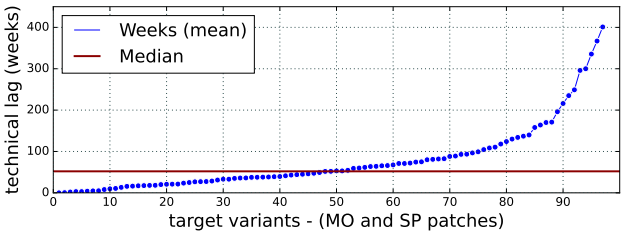


**Figure 8: A line plot showing the patch technical lag of the target variants with respect to the patches of MO and SP.**

Since each variant may have one or more patches classified as MO or SP, we calculate the technical lag of a target variant based on the technical lag of the patches in that variant. We take the mean of all the patches' technical lag as the technical lag of the target variant and plot it in Figure 8. Figure 8 is a line plot showing how much technical lag there is in each of the 97 target variants. A point on the line plot represents a target variant (x-axis) and the technical lag in terms of weeks (y-axis). We observe a median of 52 weeks as the technical lag of the target variants, which is relatively high.

> ***Summary – RQ2**: We have found that most of the missed and the incomplete patches in the target variants have a patch technical lag of 27 weeks or more. We have also observed that the patch technical lag per variant is about 52 weeks.*

## 5.2 Discussion & Implications

Our empirical results have revealed that many variants exhibit patch technical lag. However, a survey with variant developers reported by Businge et al. [9] show that there is limited code integration

between variants. The authors further reveal that variant developers indicated reasons for lack of code integration such as: "variants have nothing to share", "variants have technically diverged", "variants are implementing different technology". This implies that target variant developers may be ignorant of interesting patches of MO in the source variants. Furthermore, the developers may not have the time and effort to identify the missed patches since an interesting patch may be buried in a pool of other uninteresting changes. These developers could use PaReco to find these interesting patches and reduce the patch technical lag. PaReco can uncover and recommend the interesting patches to the target variants as they are introduced in the source variants.

Although our tool supports variability management in the space dimension (divergent variants), the tool could also manage variability in time (sequential releases of one variant). The study of Decan et al. [16] reports that older releases of packages in the package managers of npm, Cargo, Packagist, and RubyGems, still have dependent packages. This implies that to support the dependent packages of the different package releases, the owners have to maintain the releases in parallel. Therefore, a patch in a later release has to be backported to all the earlier releases with dependent packages. With many releases involved, not all patches found later can be interesting to earlier releases. PaReco can also be used to mine patches in a later release of a project and classify the patches as MO in earlier releases of the same project. The classified MO patches can be used to benefit backporting in earlier buggy releases.

> ***Actionable result**: By specifying a specific time interval,* PaReco *can mine a patch from the source variant and classify the patch as interesting or not interesting to the target variant.* PaReco *can also be used to manage variability in "time" by automatically identifying interesting patches in later project releases that can be used for backporting in earlier still supported buggy releases.*

## 6 RELATED WORK

Throughout earlier sections of the paper, we discussed a number of studies that relate to ours. In this section, we shall only discuss the studies that we have not yet discussed.

**Software product-line (SPL)**. A more systematic way of developing variants is through SPL, which consists of a set of similar software products (i.e., variants) with well-defined commonalities and variabilities. Horcas et. al. [24] survey SPL tools and give 12 different roadmaps for existing tools that can handle all or some part of the SPL processes. Lapeña et. al. [32] implemented an approach that ranks legacy code of similar products based on the requirements of the new product and searches for relevant methods in the highest ranked legacy code, decreasing the amount of code the developers need to verify during clone&own.

**Managed clone&own variants**. The common goal of studies on managed clone&own variants is to support the engineering of multi-variant software systems by reducing their limitations. Mahmood et al. [35] design, formalize, and prototype a lightweight method that generalizes clone-management and product-lines through exploiting the spectrum between the two extremes of ad hoc clone&own and fully integrated platform. Bittner et al. [5] proposed an approach that ranks the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements. To support the ongoing development of clone&own projects, there are studies that have proposed a feature trace recording a method to infer feature traces in source code changes [26, 34, 36].

**Clone detection**. Since we employ a clone detection technique, our work can also be related to different studies on clone detection. Hou and Zhang [37] present a survey on the different methods and technologies used for clone detection. They discuss text-based, token-based, tree-based, and metric-based clone detection and program dependency graphs. Zhang and Sakurai [49] recently performed a comprehensive review of clone detection tools from a security perspective. In total they included 8 tools from different studies, including ReDeBug, VUDDY and VGraph. The VUDDY [30] clone detection tool identifies type-1 and type-2 clones, but only supports C and C++. Another tool VGraph [8] uses a technique mostly similar to ReDeBug, as they focus on the lines of source code that are modified during the patching process. However, they represent the code as a graph instead of text.There are also other tools like SourcerCC [43], CPMiner [33], CCFinder [27]. These use a token-based method to tokenize source code and a measure to quantify the overlap between tokens for clone detection, but unlike the other three tools, no information from the patch is used. Hat et al. [23] analyzed shared files and their variants on GitHub, and present changes might be useful for meta-maintenance of clone-and own variants. Kawamitsu et al. [28] identify clone&own reuse on file-level based on the Longest Common Subsequence similarity between a copy and candidate origin file in C programs.

## 7 THREATS TO VALIDITY

**Construct validity**. These threats concern the relation between the theory behind the experiment and the observed findings. They can be mainly due to imprecision in the measurements we performed. Imprecisions in our measurements could occur during patch extraction from the source target and during clone detection to identify interesting patches for the target. For patch extraction, we have used keywords that earlier studies have validated. For clone detection, we have extended the tool ReDeBug that has been tested on OS-sized distribution projects. We have also gone ahead to validate our results manually, and our validation results are comparable to earlier studies by Kim et al. [30] that have also manually validated the ReDeBuG tool in a similar context. Based on the method employed to extract files for comparison between the source and the target, we acknowledge possible imprecisions since we considered all files that PaReco could not find in the target as non-existent files. Last but not least, we also acknowledge a possible threat resulting from considering the most prevalent positive classes of MO, ED, SP of the file class as the class of patch.

**Internal validity**. These threats concern choices and factors internal to the study that could influence the observations we made. Given that the method we use to identify the patches is entirely based on bug-fixing keywords in the merged pull requests, the tool may suffer from false positives (wrong patches) and false negatives (missed patches). To reduce the false negatives one could identify

more patches by searching commit messages. However, a patch may spread in more than one commit [4, 42, 46, 47], making identifying a complete patch difficult, if not impossible. More false-negative patches could be missed by PaReco through other integration techniques like direct integration or using other methods like `git push`. Although we do not control the false negatives, they do not impact the classification's accuracy. Regarding the false positives, while performing the manual validation on the patches, we analyzed the PR titles, and we did not find any false positives. Furthermore, these false positives do not impact the classification's accuracy since the tool would classify them as not interesting (NI). The variant pair filtering criterion using an educated estimation of cut-off dates and the filtering of complex cases using the number of changed files and changed lines of code could also be a threat to our findings.

**External validity.** The threats concern whether the results can be generalized outside the scope of this study. We cannot claim that the results represent all the patches since, while validating the tool's accuracy, we excluded the complex cases. However, our results are still representative since less complex cases considered in the validation exercise constituted 78.4% of the total population. While our dataset can be regarded as representative of variants on GitHub, we do not make any claim about its generalizability to other social coding platforms (e.g., BitBucket/GitLab).

## 8 CONCLUSION AND FUTURE WORK

To understand the extent of redundant and missed patches among divergent variants, we conducted an empirical investigation on 364 source→target divergent variant pairs. To this end, we introduced and used PaReco, a tool that relies on clone detection to identify the redundant and missed patches among variants in both space and time. Form the 364 source→target pairs analyzed, we found 2,225 interesting patches classified with an overall precision, recall, accuracy and F1-score of 91%, 80%, 88% and 85%, respectively. 47.7% of interesting patches are classified as effort duplication that are found in only a third of the divergent variant pairs. In follow-up work we plan to extend PaReco with more capabilities and accuracy, and to transform it into a patch recommender tool.

## REFERENCES

[1] "". 2022. File Path. raw.githubusercontent.com/[*user*]/[*repo*]/[*branch*]/[*path*].

[2] "". 2022. "Sample Size Calculator". https://www.calculator.net/sample-size-calculator.html.

[3] annonymous. 2022. "Online Appendix". https://figshare.com/s/4952174de6c21fa9874f.

[4] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. The Missing Links: Bugs and Bug-Fix Commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 97–106.

[5] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1007–1020.

[6] Bluebrain. 2022. Blue Brain Project. https://portal.bluebrain.epfl.ch.

[7] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A Theory of Software Product Line Refinement. *Theor. Comput. Sci.* 455 (2012), 2–30.

[8] Benjamin Bowman and H. Howie Huang. 2020. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS P)*. 53–69.

[9] John Businge, Alexandre Decan, Ahmed Zerouali, Tom Mens, and Coen De Roover Serge Demeyer. 2022. Variant Forks – Motivations and Impediments. In *Proceedings of the 29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. Accepted.

[10] John Businge, Moses Openja, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. 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, 625–634.

[11] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and Maintenance Practices among Divergent Forks in Three Software Ecosystems. *Journal of Empirical Software Engineering* 27, 2 (2022).

[12] Brian W. Fitzpatrick C. Michael Pilato, Ben Collins-Sussman. 2008. *Version Control with Subversion*. Addison-Wesley. 864 pages.

[13] Marco Castelluccio, Le An, and Foutse Khomh. 2019. An empirical study of patch uplift in rapid release development pipelines. *Empir. Softw. Eng.* 24, 5 (2019), 3008–3044. https://doi.org/10.1007/s10664-018-9665-y

[14] Scott Chacon and Ben Straub. 2014. "Git Tools - Rewriting History". https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History.

[15] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Softw. Eng.* 24, 1 (2019), 381–416.

[16] Alexandre Decan, Tom Mens, Ahmed Zerouali, and Coen De Roover. 2021. Back to the Past – Analysing Backporting Practices in Package Dependency Networks. *IEEE Transactions on Software Engineering* (2021), 1–1.

[17] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *2013 17th European Conference on Software Maintenance and Reengineering*. 25–34. https://doi.org/10.1109/CSMR.2013.13

[18] The Python Software Foundation. 2022. difflib — Helpers for computing deltas. https://docs.python.org/3/library/difflib.html.

[19] GitHub. 2021. "Online Appendix". http://www.gnu.org/software/diffutils/manual/html_node/Hunks.html.

[20] GitHub. 2021. "Online Appendix". buggy_file in source (MO): https://raw.githubusercontent.com/nerdvegas/rez/3a5afd8/src/rezplugins/shell/cmd.py.

[21] GitHub. 2021. "Online Appendix". Patched file in source (MO): https://raw.githubusercontent.com/nerdvegas/rez/25469bc/src/rezplugins/shell/cmd.py.

[22] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*, Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli (Eds.). Springer International Publishing, 182–192.

[23] Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Christoph Treude. 2021. *Same File, Different Changes: The Potential of Meta-Maintenance on GitHub*. IEEE Press, 773–784. https://doi.org/10.1109/ICSE43902.2021.00076

[24] Jose-Miguel Horcas, Monica Pinto, and Lidia Fuentes. 2019. Software product line engineering: a practical experience [research]. 1–13.

[25] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *IEEE* (05 2012), 48–62.

[26] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 61–70.

[27] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28 (08 2002), 654– 670.

[28] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. 2014. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 305–314. https://doi.org/10.1109/SCAM.2014.17

[29] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 21–25.

[30] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 595–614. https://doi.org/10.1109/SP.2017.62 2017 IEEE Symposium on Security and Privacy, SP 2017 ; Conference date: 22-05-2017 Through 24-05-2017.

[31] Jacob Krüger and Thorsten Berger. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 432–444.

[32] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. 2016. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *Proceedings of the 20th International Systems and Software Product Line Conference*. Association for Computing Machinery, New York, NY, USA, 194–203.

[33] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192. https://doi.org/10.1109/TSE.2006.28

[34] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2018. Variability Extraction and Modeling for Product Variants. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. Association for Computing Machinery, New York, NY, USA, 250.

[35] Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1658–1670.

[36] Gabriela Karoline Michelon. 2020. Evolving System Families in Space and Time. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B*. Association for Computing Machinery, New York, NY, USA, 104–111.

[37] Hou Min and Zhang Li Ping. 2019. Survey on Software Clone Detection Research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*. Association for Computing Machinery, New York, NY, USA, 9–16.

[38] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, USA, 120.

[39] patchesandmissedmatches. 2022. patchesandmissedmatches. https://github.com/patchesandmissedmatches/patchesandmissedmatches. (2022).

[40] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with Variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China). Association for Computing Machinery, New York, NY, USA, 329–332.

[41] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM* 60, 10 (Sept. 2017), 91–100. https://doi.org/10.1145/3126905

[42] Christophe Rezk, Yasutaka Kamei, and Shane Mcintosh. 2021. The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects. *IEEE Transactions on Software Engineering* (2021), 1–1.

[43] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168.

[44] Spack. 2021. "Online Appendix". https://spack.io.

[45] Stefan Stanciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wasowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 323–333.

[46] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 832–837.

[47] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and Exploiting the Correlations between Bug-Inducing and Bug-Fixing Commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 326–337.

[48] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems* (Shanghai, China) *(SEAS '06)*. Association for Computing Machinery, New York, NY, USA, 61–67.

[49] Haibo Zhang and Kouichi Sakurai. 2021. A Survey of Software Clone Detection From Security Perspective. *IEEE Access* 9 (2021), 48157–48173.

[50] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub. In *International Conference on Software Engineering* (Seoul, South Korea). ACM, 268–269.