Noname manuscript No.
(will be inserted by the editor)

Reuse and Maintenance Practices among Divergent Forks in Three Software Ecosystems

John Businge · Moses Openja · Sarah Nadi · Thorsten Berger

Received: date / Accepted: date

Abstract With the rise of social coding platforms that rely on distributed version control systems, software reuse is also on the rise. Many software developers leverage this reuse by creating variants through forking, to account for different customer needs, markets, or environments. Forked variants then form a so-called software family; they share a common code base and are maintained in parallel by same or different developers. As such, software families can easily arise within software ecosystems, which are large collections of interdependent software components maintained by communities of collaborating contributors. However, little is known about the existence and characteristics of such families within ecosystems, especially about their maintenance practices. Improving our empirical understanding of such families will help build better tools for maintaining and evolving such families.

We empirically explore maintenance practices in such fork-based software families within ecosystems of open-source software. Our focus is on three of the largest software ecosystems existence today: Android, .NET, and JavaScript. We identify and analyze software families that are maintained together and that exist both on the official distribution platform (Google play, nuget, and npm) as well as on GitHub, allowing us to analyze reuse practices in depth. We mine and identify *38 software families*, *526 software families*, and *8,837 software families* from the ecosystems of Android, .NET, and JavaScript, to study their characteristics and code-propagation practices. We provide scripts for analyzing code integration within our families. Interestingly, our results show that there is little code integration across the studied software families from the three ecosystems. Our studied families also show that techniques of direct integration using *git* outside of GitHub is more

J. Businge
Mbarara University of Science and Technology, Uganda and
University of Antwerp, Belgium, johnxu21@gmail.com, **Corresponding Author**

M. Openja
SWAT Lab., École Polytechnique de Montréal Montréal,Canada, openjamosesopm@gmail.com

S. Nadi
University of Alberta, Canada, nadi@ualberta.ca

T. Berger
Ruhr University Bochum, Germany and
Chalmers | University of Gothenburg, Sweden, thorsten.berger@rub.de

commonly used than GitHub pull requests. Overall, we hope to raise awareness about the existence of software families within larger ecosystems of software, calling for further research and better tools support to effectively maintain and evolve them.

1 Introduction

The increased popularity of social-coding platforms such as GitHub made *forking* a powerful mechanism to easily clone software repositories for creating new software. A developer may fork a *mainline repository* into a new *forked repository*, often transforming governance over the latter to a new developer, while preserving the full revision history and establishing traceability information. While forking allows isolated development and independent evolution of repositories, the traceability allows comparing the revision histories, for instance, to determine whether one repository is *ahead* of the other (i.e., contains changes not yet integrated in the other). It also allows easier commit propagation across the repositories.

Many studies on forking exist, often focusing on the reasons and outcomes [43–45, 52, 60–63] or on the community dynamics as influenced by forking [25]. The community typically distinguishes between two kinds of forks [63]: *social forks* that are created for isolated development with the goal of contributing back to the mainline and *divergent forks* that are created for splitting off a new development branch, often to steer the development into another direction without intending to contribute back, while leveraging the mainline project that defines or adheres to some standards [58]. Divergent forks are more relevant for supporting large-scale software reuse—the focus of this paper.

Studies on divergent forks usually rely on general heuristics to identify as many forks as possible, without systematically verifying that these are indeed divergent forks. Additionally, when studying code propagation techniques, existing studies do not consider the intricacies of git to identify the possible types of code propagation (e.g., offline git rebasing without using GitHub at all), but focus only on pull requests. To address the first challenge of identifying divergent forks, we use the insight that there are particular ecosystems that have a systematic way of publishing "members" of the ecosystem. For example, most Android apps are published on the Google Play store. Similarly, most Eclipse plug-ins are distributed on the Eclipse marketplace. The advantage of such ecosystems is that each member has a unique ID that identifies it. Thus, given an open-source GitHub repository and its fork, we can verify whether the fork is actually an independent version of the original mainline (which is a core criteria of a divergent fork) by checking that both the mainline and the fork are listed as separate entries in the corresponding distribution platform. To address the second challenge of considering the git intricacies, we design a technique that identifies the majority of code propagation techniques on Git and GitHub by leveraging all commit meta data. Inspired by the notion of *software families* (a.k.a., *program families* [3, 5, 20, 23, 33, 48, 57]—portfolios of managed and similar software systems in an application domain—we use the term *software family*, or *family* for short, to refer to a mainline repository and its corresponding divergent forks. We refer to each family member as a *variant*.

We present a large-scale empirical study on reuse and maintenance practices via code propagation among software families in software ecosystems. We take the above considerations into account and study three large-scale ecosystems in different

technological spaces: `Android`, `JavaScript`, and `.NET`. Android is one of the largest and most successful software ecosystem with substantial software reuse [4,36,39,53]. The `JavaScript` ecosystem distributes its packages through `npm`, which is by far the largest package manager with over 1.82M package distributions.[1] The `.NET` ecosystem has a package management system, `nuget`, that is moderately large with over 261K packages.[1] As such, our three selected ecosystems vary in their nature (apps versus packages), their programming languages (Java, JavaScript, and C#), and their sizes (in terms of their distribution platforms).
Our study addresses two main research questions:

RQ1 *What are the characteristics of software families in our ecosystems?*
We investigate general characteristics of the families and their variants, including the number of variants per family and the divergence of application domains, developer ownership, and variant popularities within the families. We also determine the frequencies of variant maintenance, looking at releases numbers. This allows putting the studied maintenance and co-evolution practices into context.

RQ2 *How are software families maintained and co-evolved in our ecosystems?*
To determine management practices, we investigate how code is propagated between the mainline and its divergent forks in the family. For example, are pull requests used as the main propagation technique? Is code propagated only from the mainline to the forks, or is there propagation in the other direction, too? We study the code propagation mechanisms used as well as the kinds of changes being propagated.

To the best of our knowledge, our work is the first to provide a large-scale in-depth study of code-propagation practices in divergent forks. Understanding these code-propagation strategies exercised by developers can help in building better tool support for software customization and code reuse. We analyze pairs of mainline and fork open source projects whose package releases are available in package distribution platforms of the three ecosystems: Android comprising *38 software families*, .NET comprising *526 software families*, and JavaScript comprising *8,837 software families*.

Our results show that the majority (82 %) of forks we study are owned by developers different than those of the within a family. Such distinction of ownership gives us confidence that we are studying real divergent forks. Interestingly though, we find little code propagation across all the mainline–fork pairs in the three ecosystems we studied. The most used code propagation technique is `git merge/rebase` that is used in 33 % of Android mainline-fork pairs, 11 % of JavaScript pairs, and 18 % of .NET pairs. We find that cherry picking is less frequently used, with only 9 %, 0.9 %, and 2.5 % of Android, JavaScript, and .NET pairs using it, respectively. Among the three pull request integration mechanisms we studied (merge, rebase, and squash), the most used pull request integration mechanism is the merge option in the direction of fork→mainline, where 2.4 %, 7 %, and 11 % of the pairs in Android, JavaScript, and .NET use this strategy. We find that integrating commits using squashed or rebased pull requests is rare in all three ecosystems. Overall, we find that when code propagation occurs, it seems that fork developers perform this propagation directly through *git* and outside of GitHub's built-in pull

[1] As seen on Libraries.io by June 2021

request mechanism. This observation implies that simply relying on pull requests to understand code propagation practices in divergent forks is not enough.
In summary, this work makes the following contributions:

- We propose leveraging the main distribution platforms of three ecosystems to precisely identify divergent forks. We devise a technique to identifying families in these ecosystems by using data both from GitHub and the respective distribution platform.
- In contrast to previous studies on code propagation strategies that either focused only on pull requests or on directly comparing commit IDs, we are the first to study code propagation while considering pull requests with the options of squash / rebase as well as *git* rebased and cherry-picked commits.
- We analyze the prevalence of code propagation within software families as well as the types of propagation strategies used.
- We synthesize implications of our results for code reuse tools.
- We provide an online appendix [1] containing our datasets, intermediate results, and the scripts to trace code propagation between any mainline-fork pair.

An earlier version of this work appeared as a conference paper [11]. It focused on analyzing code propagation at the commit level within only the Android ecosystem. It also provided preliminary insights on the reasons why different app variants exist. This article extends the conference paper as follows. First, we extend our analysis with two more ecosystems of moderate to large scale. Second, we substantially improve our identification of code integration methods by not focusing solely on pull requests or direct comparison of commit IDs. Instead, we are the first to consider most types of code propagation techniques, including rebasing, squashing, and cherry-picking commits. Third, we contribute a toolchain for analyzing code propagation between any mainline–fork pair. (iv) We provide more discussion of the implications of our results.

Parts of RQ1 for the JavaScript ecosystem have been previously presented as a workshop paper [7]. In this article, our additional contributions for RQ1 for the JavaScript ecosystem are the following. First, we refine the JavaScript dataset by ensuring that the mainline-fork pairs exist both on GitHub and the npm package manager. To this end, we eliminate a total of 2,456 mainline-fork pairs where either the mainline or fork were deleted from GitHub, but their package releases still existed on the npm package manager. Second, we provide a more detailed description of how the dataset was collected and provide the full refined dataset in the replication package. Third, we create an additional dataset of new families from the .NET ecosystem. Fourth, in addition to the new characteristic `variant ownership` as well as more illustrative graph comparisons, we discuss the characteristics of the mainline-fork pairs across all three ecosystems.

2 Background on Code Propagation Strategies

We now discuss the mechanisms offered by GitHub and similar social-coding platforms to propagate code among different repositories. We describe characteristics of these mechanisms and the kind of metadata they generate, which an automated identification technique can potentially rely on.

While a mainline and a forked repository are under no obligation to synchronize any changes, developers commonly propagate their code changes (e.g., new features

Table 1 Changes of commit metadata during code propagation for the different kinds of code propagation with GitHub or Git facilities

	Pull Requests			Git Commands		
Metadata changed	Merge	Squash	Rebase	Cherry-pick	Merge	Rebase
Commit ID	No	Yes	Yes	Yes	No	No
Author Name	No	Yes	No	No	No	No
Author Date	No	Yes	No	No	No	No
Committer Name	No	Yes/No	Yes/No	Yes/No	No	No
Committer Date	No	Yes	Yes	Yes	No	No
Commit Message	No	Yes	No	No	No	No
File details	No	No	No	No	No	No

Yes metadata change No no change of metadata

or bug fixes) among repositories via commit integration [28, 46]. For tracing such propagation, however, the metadata provided by GitHub is not always reliable. For instance, Kalliamvakou et al. [29] and Kononenko et al. [31] found a large number of pull requests appearing as not merged while they were actually merged. The authors find that it is not uncommon for destination repositories to resolve pull requests outside GitHub. This is why our work considers both commit integration through GitHub and commit integration directly using git, but outside GitHub.

In the following, we describe code propagation using GitHub and git facilities. Table 1 provides details on the relationship between commits across forked repositories based on the respective code propagation technique used. To collect the information in this table, we read the official references [59][2,3] and online resources[4] as well as created toy repositories to mimic the various integration scenarios in order to verify this information. We use these insights for creating our code propagation traceability technique described in Section 3.3.

2.1 Propagation with GitHub Facilities

A *pull request* has a *head ref*, which is the reference for the source repository and a branch a developer wants to pull commits from; we refer to it as the *source branch*. A pull request also has a *base ref*, which is the reference for the destination repository into which the pulled commits are integrated into; we refer to it as the *destination branch* for clarity. The source and destination branches may belong to the same repository or to different repositories. When studying code propagation in a software family, we are mainly interested in pull requests from one source repository in the family to another destination repository in the same family.

Once a pull request is submitted on GitHub, a developer can use its user interface to integrate the commits in the pull request into the destination branch using one of these three options: (i) merge the pull request commits, (ii) rebase the pull request commits, and (iii) squash the pull request commits.

[2] `https://www.atlassian.com/git/tutorials/merging-vs-rebasing`

[3] `https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-request-merges`

[4] `https://cloudfour.com/thinks/squashing-your-pull-requests/`

- *Merge pull request commits* is the default. When the developer chooses this option, the commit history in the destination branch will be retained exactly as it is. As can be seen from Table 1, the metadata of the integrated commits from the source branch remain unchanged in the destination branch. However, a new *merge commit* will be created in the destination branch to "tie together" the histories of both branches [2].
- *Rebase and merge pull request commits*: When the integrator selects the *Rebase and merge* option on a pull request on GitHub, all commits from the source branch are replayed onto the destination branch and integrated without a merge commit. From Table 1, we can see that using this integration technique, the commit metadata between source and destination preserves the author name, author date, and commit message but alters the commit ID, committer name, and committer date. The committer name becomes the name of the developer from the destination repository who rebased and merged the pull request. Note that if the developer who submitted the pull request is coincidentally the same as the developer who integrates it (e.g., because the developer works on both repositories), then the committer name will remain the same [2].
- *Squash and merge pull request commits*: When the integrator selects the *Squash and merge* option on a pull request on GitHub, the pull request's commits are squashed into a single commit. Instead of seeing all of a contributor's commits from the source branch, the commits are squashed into one commit and included in the commit history of the destination branch. Apart from the file details, all other commit meta data changes. The committer name changes unless, similar to above, the original committer and the developer merging the pull request are the same [2].

2.2 Propagation with Git Facilities (Cherry Pick, Merge, and Rebase Commits)

A developer may also not rely on the GitHub user interface and instead choose to integrate commits from a source branch into a destination branch outside GitHub using one of the git integration commands. The integrator has to first locally fetch commits from the source branch (for example mainline) that contains the commits they wish to integrate into their branch. They then perform the integration locally using one of four options outlined below ((i) git merge, (ii) git rebase, (iii) git cherry-pick, and (iv) other Git commands that rewrite commit history) and afterwards, push the changes to their corresponding GitHub repository[5].

- *Git cherry-pick commits*: Cherry picking is the act of picking a commit from one branch and integrating it into another branch. Commit cherry picking can, for example, be useful if a mainline developer creates a commit to patch a pre-existing bug. If the fork developer cares only about this bug patch and not other changes in the mainline, then they can cherry pick this single commit and integrate it into their fork. As shown in Table 1, the author name, author date, commit message, and file details of the cherry picked commit remain the same in the destination branch. The commit ID, committer name, and committer date however do change. Note that the committer name may remain the same

[5] `https://www.atlassian.com/git/tutorials/merging-vs-rebasing`

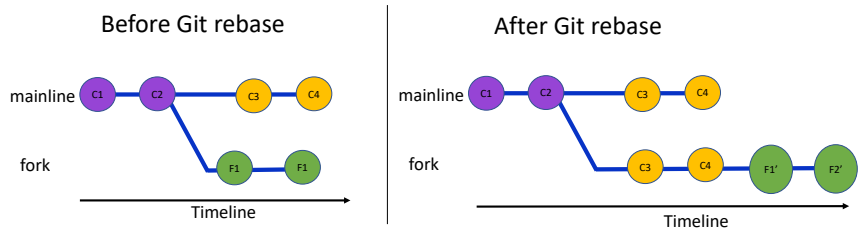


Fig. 1 Illustration of git rebase

if the integrator is the same developer who performed the original commit in the source branch.

- *Git merge commits*: Like in the pull request merge, git merge also preserves all the commit metadata and creates an extraneous new merge commit in the destination branch that ties together the histories of both branches.
- *Git rebase commits*: Rebasing is an act of moving commits from their current location (following an older commit) to a new head (newest commit) of their branch [18]. Git rebase deviates slightly from rebasing pull requests on GitHub as it does not change the committer information. To better understand git rebase, let us explain it with an illustration based on the experiments we carried out. On the left-hand side of Fig. 1, we have a mainline repository and a fork repository where each repository made updates to the code through commits C3 and C4 in the mainline and commits F1 and F2 in the fork. The fork developer observes that the new updates in the mainline are interesting and decides to integrate them using rebasing. After rebasing, the commit history will look the right side of Fig. 1. Notice that the IDs and the order of the integrated commits C3 and C4 in the fork branch are unchanged. However, the IDs of commits F1 and F2 change to F1' and F2'. In this case, Git rebase is like the fork developer saying "*Hey, I know I started this branch last week, but other people made changes in the meantime. I don't want to deal with their changes coming after mine and maybe conflicting, so can you pretend that I made [my changes] today?*" [59].
- *Other Git commands that rewrite commit history*: Git has a number of other tools that rewrite commit history, including changing commit messages, commit order, or splitting commits [17]. These commands include: `git commit --amend`, `git rebase -i HEAD~N`, and `git --squash`, etc. Most of these commands significantly change the history and the meta data of commits. If the integrator uses any of these commands in the destination repository, then there is no straightforward way to match the integrated commits across the two repositories [17].

3 Methodology

Our goal is to improve the empirical understanding of maintenance practices, specifically code propagation in software families. We identify and analyze software families by using data from both GitHub and the distribution platforms of the three ecosystems.

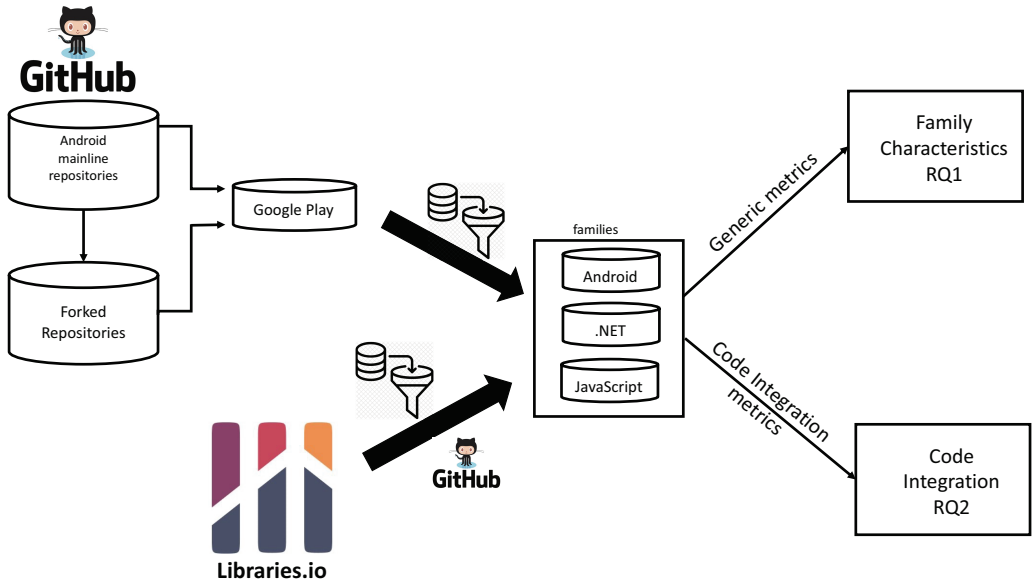


Fig. 2 Illustration of our data sources and our ecosystem analysis process

3.1 Identifying Software Families

Given the different nature of our studied ecosystems in terms of what information each distribution platform stores and how this information is accessed, we employ different techniques to identify Android families versus JavaScript and .NET families. Figure 2 shows an overview of this process. We extract families in the Android ecosystem from GitHub and Google Play while the families in .NET and JavaScript are extracted from `Libraries.io`.[6]

3.1.1 Identifying Android Families

We are interested in identifying families of real Android apps that are evidently used by end users. Taking all GitHub repositories with Android apps into account would also include toy apps or course assignments. To this end, we identify source repositories of apps that also exist on Google Play. We mainly match GitHub repositories and Google Play apps via their unique identifier—the package name contained in the app manifest file (`AndroidManifest.xml`). Such manifest files also declare the app's components, necessary permissions, and required hardware and Android version. As such, each Android app in a software family must have a unique package name, which excludes any forked repositories where the package name was not modified. More specifically, we identify Android families using a relatively conservative filtering approach as follows.

1. Using GitHub's REST API v3, we identify 79,338 *mainline* repositories matching the following criteria: (1) is not a fork; (2) the repository contains the word

[6] https://libraries.io/

"Android" in the name/description/readme; (3) has been forked at least twice; (4) was created before 01/07/2019 (we mined on 14/12/2019, so we used the date 01/07/2019 to obtain repositories that have some history) (5) has an `AndroidManifest.xml` file; (6) has a description or readme.md file; and (7) has a number of forks ≥ 2 to reduce the chance of finding student assignments [41].

2. To ensure that we are collecting real-world apps, we check if the identified mainline repositories exist on Google Play. From each repository's `AndroidManifest.xml` file, we extract the app's package name and check its existence on Google Play. In total, we find 7,423 mainline repositories representing an actual Google Play app [8].
3. We filter out duplicate mainline repositories containing `AndroidManifest.xml` files with the same package name. Such duplicates easily arise when an app's source code is copied without forking. Since package names are unique on Google Play, only one of these duplicate repositories can actually correspond to the Google Play app. We manually select one repository from these duplicates by considering repository popularity (number of forks and stars on GitHub), repository and app descriptions on both GitHub and Google Play, as well as the developer name on GitHub and Google Play. In some cases, the Google Play app description conveniently linked to the GitHub repository. As a result of this step, we discard 1,232 repositories and are left with 6,191 mainline repositories.
4. To ensure that we study repositories with enough development history, we filter out mainlines with fewer than six commits in their lifetime, according to the median number of commits in GitHub projects found by prior work [29]. This leaves us with 4,337 mainline repositories.
5. We filter out mainline repositories without any active forks, which have no commit after the forking date and were probably abandoned. This leaves us with 1,166 mainline repositories, which have a total of 12,025 active forks altogether.
6. We remove forks that have the same package name as their mainline. If no forks remain for a given mainline, we also remove this mainline. For the forks with different package names than their corresponding mainline, we check the existence of the fork's package name on Google Play in order to ensure that the fork is also a real (and different) Android app. This leaves us with 69 app families comprising of 95 forks.
7. Finally, by manual inspection, we filter out forked repositories whose app package name points to a Google Play app that is not the correct app. This analysis is based on the observation that, sometimes, fork developers copy code including the `AndroidManifest.xml` from another app without changing the package name. This practice results in the forked app's package name pointing to an app that exists on Google Play, but that is not the one hosted in the GitHub repository. We inspect the `Readme.md` and unique commit messages in the GitHub repository and the respective Google Play description page. Eliminating all mismatched apps leaves a total of **38 app families comprising of 54 forked apps**—our final dataset to answer the research questions.

3.1.2 Identifying JavaScript and .NET Families

A family in the JavaScript and .NET ecosystems comprises packages of libraries of applications written in the respective language. Similar to the Android ecosystem, we only consider packages that exist as source-code repositories on GitHub and

on the ecosystem's main distribution channgels: npm and nuget. The metadata of a package release on the package managers of npm or nuget is similar. On both package managers, a package's metadata include: source repository of the package (GitHub, GitLab, BitBucket), number of dependent projects/packages, number of dependencies, number of package releases, and the package contributors. Fortunately, most of the data of 37 package managers for different ecosystems can be found on one central location Libraries.io, which is a platform that periodically collects all data from different package managers. In addition to the metadata for a specific package on a given package manager, Libraries.io also extends the package metadata with more information from GitHub. For example, it stores a Forkboolean field, which indicates whether the corresponding repository of a package is a fork. Such a field Forkboolean can help us identify forked repositories that have published their packages. Note that this is different from the Android ecosystem where such explicit traceability does not exist, which is why we first mine repositories from GitHub and then filter out those that are published on Google Play. In contrast, with .NET and JavaScript, we mine the families directly from Libraries.io. We extract the families from the latest Libraries.io data dump release 1.6.0 that was released on January 12, 2020. The meta-model for the data on the Libraries.io data dump can be found online.[7] We extract .NET and JavaScript families from Libraries.io with the following steps:

1. Using the package's field Platform, we filter out the packages that are distributed on nuget and npm package managers.
2. Next, we use the field Forkboolean to identify repositories that are forks, and use the field Fork Source Name with Owner to identify the fork repository name as well as the parent repository (mainline). We extract all fork repositories that map to published packages on nuget and npm.
3. Next, we merge the sets of packages from Step 1 and Step 2 to identify only packages that make a mainline-fork pairs (i.e., where the fork repository and its corresponding mainline in the set in Step 2 have their packages present in the set in Step 1. Using the GitHub API, we then verify that indeed the mainline parent of the divergent fork and they are still existing on GitHub so as to eliminate wrong pairs and (e.g., those that have been deleted from GitHub). From the .NET ecosystem, we identify a total of *526 software families* having a total of *590 mainline–fork variant pairs*. From the JavaScript ecosystem, we identify a total of *8,837 software families* having a total of *10,357 mainline–fork variant pairs*. Similar to Android families, a family in .NET and JavaScript contains at least one mainline and one or more variant forks.

3.2 Identifying Family Characteristics (RQ1)

We now describe how we identify characteristics of the identified families and their variants (i.e., mainlines and forks) for our three ecosystems.

We define and calculate various metrics as follows. Table 3 in Section 4 summarizes all metrics (and provides their values). Note that, given the different nature of these ecosystems and the type of information available for each, some metrics are specific to only some of the ecosystems. For example, *FamilySize* is a metric we

[7] https://libraries.io/data

can calculate for all variants in all the three ecosystems. On the other hand, given the difference in nature of Android variants and JavaScript/.NET packages, we need to calculate variant popularity differently across the ecosystems (downloads and reviews versus dependents and dependencies).

In the following, we discuss the goal of each metric and how we calculate it. Overall, we define metrics to capture general characteristics of variants, of variant maintenance activities, of variant ownership, and of variant popularity. For repositories in the Android ecosystem, we extract the metrics from GitHub and Google Play. For repositories in the .NET and JavaScript ecosystems, we extract the metrics from GitHub and Libraries.io.

3.2.1 General Characteristics

Family Size: We record the number of variants (metric *FamilySize* in Table 3) for all families in the three ecosystems. Intuitively, a family with $FamilySize = 2$ has one mainline and one fork, while a family with $FamilySize = 3$ has one mainline and two forks.

Variant Package Dependencies: ecosystems provide a huge bazaar of software that can be reused through explicit package dependencies [21]. Since a divergent fork inherits functionality from the mainline and may also continuously synchronize with the mainline to acquire new changes, one would expect that the number of package dependencies for a mainline and fork would be the same. However, it would be interesting to see cases where they are not the same. In this context, for example, if the fork has more dependencies, it could mean that fork is implementing new features that are not in the mainline. We extract the number of dependencies from Libraries.io. For Android, we extracted the dependencies from the apps *Gradle* files on GitHub.

Android variant categories: Using the variant's metadata available on Google Play, we also determine its variant category (e.g., Business, Finance, Productivity) and extract its description. We also record whether the variants are listed under the same category on Google Play, which helps us understand the nature of the variants in a family.

3.2.2 Identifying Maintenance Activities (JavaScript & .NET only)

A repository with many releases shows that it is being actively maintained since each release indicates either bug fixes or / and new features being introduced. To this end, we are interested in seeing the relationship between the mainline and the fork in terms of the number of package releases on the package distribution platforms. We collect the number of package releases for variants in the .NET and JavaScript ecosystems from Libraries.io. The metrics related to variant maintenance activity are *PackageReleasesMLV* for the mainline variants and *PackageReleasesFV* for the fork variants. Unfortunately the package manager for variants in Android ecosystem (Google Play store) does not keep history for the applications, and therefore we cannot extract variant releases from there. An alternative to collect the variant releases in the Android ecosystem is to collect them from the repositories

themselves using the GitHub API. Unfortunately, we found that using the GitHub API to collect the list of releases of a repository returns zeros for most of the repositories even when a repository has releases. For example, we can see that the Android divergent fork `imaeses/k-9`[8] has releases. However, when we access the fork using the GitHub API for a list of releases[9], we can see that it returns an empty list. To this end, we decided not to collect package releases for the variants in the Android ecosystem.

3.2.3 Identifying Variant Ownership Characteristics

We would like to identify whether the mainline and fork variant have common owners. This is interesting to study since we determine if whether variant fork are started by the owners of the mainlines or if they are started different developers not in the mainline. We define the owner of a repository as a contributor who has access rights of integrating changes into the repository (i.e., a repository committer). As we explained in Section 2, based on the different kinds of commit integration techniques, it might be difficult to identify the original repository of a given commit (especially in cases where a mainline has many forks). To this end, we identify a repository committer (owner) as one who has merged at least one pull request, since we are certain that only contributors who have access rights to a repository can integrate changes. We consider that the mainline and a fork variant have common owners if there exists at least one common owner between them. With this criteria, both the mainline and fork variant should have at least one same developer (not a `bot`) who merged a pull request in both repositories. This means that our ownership criteria relies on each variant merging at least one pull request. Since we have very few variant pairs in the Android ecosystem, this would reduce further the very small dataset of variant pairs. To this end, we apply the described method only on the variants of .NET and JavaScript ecosystems, which have moderately large to very large dataset of variant pairs and use a different criteria to identify the owners of Android variants that we explain later. Since all the variants are published in Google Play, then each variant has an owner. We identify only 89 of the *590 mainline–fork variant pairs* in the .NET ecosystem where both the mainline and fork variant had any merged PR by a real developer. For the JavaScript ecosystem we identify only 89 of the *10,357 mainline–fork variant pairs* where both the mainline and fork variant had any merged PR by a real developer.

For the variant pairs in the Android ecosystem, we employ another method to identify ownership that covers all the dataset. We mine ownership from Google Play store. On Google play store, each variant has an attribute `developer id` or `dev id`, which is the name of the developer/company (owner) that uploads the variant on its updates on the marketplace.

3.2.4 Identifying Variant Popularity

We want to understand the popularity of the variants we are studying in terms of whether they are widely used in their respective ecosystems. We extract the popularity metrics from the distribution platform of each of our studied ecosystems.

[8] https://github.com/imaeses/k-9/releases
[9] https://api.github.com/repos/imaeses/k-9/releases

We use a different popularity measure for variants in the Android ecosystem than those from .NET and JavaScript.

- *Android variants*: For the variants in the Android ecosystem, we define two popularity metrics for the number of downloads on Google play, *DownloadsMLV* and *DownloadsFV* for the mainline and divergent fork respectively. We also define two popularity metrics for the number of reviews on Google play *ReviewsMLV* and *ReviewsFV* for the mainline and divergent fork, respectively.
- *JavaScript and .NET variants*: For variants in these two ecosystems, we record the number of other packages on the JavaScript and .NET that depend on the mainline and the fork variants (*DependentPackagesMLV* and *DepenedntPackagesFV* respectively). We also record the number of other projects on GitHub that depend on the mainline and variant (*DependentProjectsMLV* and *DependentProjectsFV* respectively). All the variant's dependent packages / projects are extracted from Libraries.io. The package and project dependents are a good way of measuring popularity since they give an indication of what other packages / projects are interested in the functionality provided by the variant.

3.3 Identifying Code Propagation (RQ2)

Answering RQ2 requires determining whether and how any code was propagated among the variants of a software family. To identify code propagation, we rely on categorizing commits in the history of the mainline and the forks based on the possible types of code propagation we discussed in Section 2.

Figure 3 describes the different kinds of commits in a family, illustrated for mainline variant and any of its divergent forks. We identify two broad categories of commits: (1) *common commits* are those that exist in both the mainline variant and the forked variant and represent either the starting commits that existed before the forking date or propagated commits, and (2) *unique commits* that exist only in one variant. For each mainline-fork pair in a family, we first identify common and then unique commits as follows.

3.3.1 Identifying Common Commits

To ensure we correctly categorize commits, we perform the following steps in this exact order. Once a commit is categorized in one step, we do not need to analyze it again in the following steps. We consider only the default repository branch **master/main branch** for both the mainline and forks.

- **Inherited commits:** The *fork date* is the point in time at which the fork variant is created. At that point, all commits in the fork are the same as those in the mainline, and we refer to them as *InheritedCommits*. In Fig. 3, the *InheritedCommits* are the purple commits 1, 2, and 3. To extract these commits for either variants, we collect all the commits since the first commit in the history until the fork date.
- **Pull-Request commits:** We first collect the merged pull requests in each repository and identify the pull requests whose source and destination branches belong to the analyzed repository pair. The GitHub API `:owner/:repo/pulls/:pull_number` provides all the information of a given pull request. One can identify the

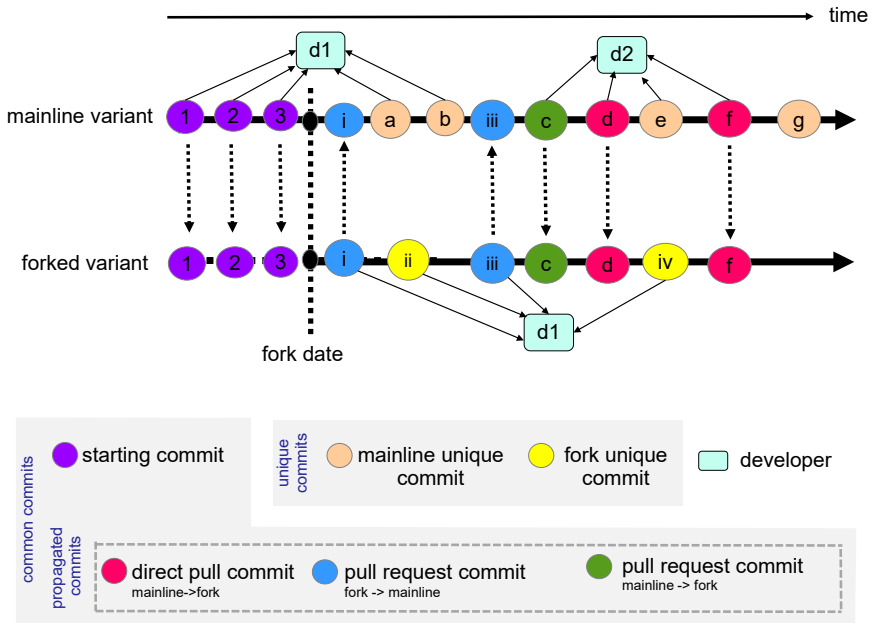


Fig. 3 Illustration of the different types of commits present in a fork variant (FV) and its corresponding mainline variant (MLV).

source and destination branches using the pull request objects `['head']['repo']['full_name']` and `['base']['repo']['full_name']` from the returned `json` response, respectively. Based on the source and destination information, we can always identify the direction of the pull request as *fork→mainline* or *mainline→fork*, as shown in Fig. 3. For each pull request, we collect the pull request commits `pr_commits` using the GitHub API `:owner/:repo/pulls/:pull_number/commits`. Regardless of how a pull request gets integrated, the commit information in the source repository is always identical to that in `pr_commits`. Thus, we can always identify the pull request commits in the source repository by comparing the IDs of the commits in `pr_commits` to those in the history of the source repository. The tricky part is identifying the integrated commits in the destination repository. Based on the information discussed in Section 2 and summarized in Table 1, we can identify the pull request commits in the destination repository as follows:

- *Merged pull request commits:* Based on Table 1, the commit IDs of pull request commits integrated using the default merge option do not change. Thus, to identify these commits, we simply compare the IDs of the `pr_commits` to those in the commit history of the destination repository.
- *Rebased pull request commits:* Recall from Table 1 that integrated commits from a rebased pull request have different commit IDs on the destination branch. Thus, we identify the rebased commits in the destination branch

by comparing the remaining unchanged commit metadata, such as author name, author date, commit message, and file details.
  - *Squashed pull request commits:* As part of a squashed pull request's meta data, GitHub records the ID of the squashed commit on the destination branch in the `merge_commit_sha` attribute[10]. Using this ID, we can identify the exact squashed commit in the destination repository. For extra verification, we also compare the changed files of all commits in the pull request with the changed files in the identified squashed commit.

- **Git merged commits:** After identifying all commits related to pull requests, we now analyze any remaining unmatched commits to identify if they might have been propagated directly through Git commands. Recall from Section 2 that this includes merged, rebased, and cherry-picked commits.
  - *Git cherry-picked commits:* We locate cherry-picked commits in the source and destination commit histories by comparing the following commit metadata: `commit ID`, `author name`, `author date`, `commit message` and `filenames` and `file changes`. We can also identify the source and the destination branches of the cherry picked commits by looking at the committer dates of the matched commits. We mark the commit with the earlier committer date to be from the source branch and that with the later date to be in the destination branch.
  - *Git merged and Git rebased commits:* At this point, we have already identified all integrated pull request commits as well as cherry picked commits. Thus, any remaining commits that have the same ID in the histories of both variants must have been propagated through git merge or git rebase. As shown in Table 1 and Fig. 1, any commits integrated through git rebase have exactly the same ID and meta data in both the source and destination branch. Similarly, commits integrated through git merge also have the same exact information. While we can differentiate git-merged and git-rebased commits by finding merge commits (those with two parents) and marking any commits between the merge commit and the common ancestor as commits that are integrated through git merge, this differentiation is not important for our purposes. We are only interested in marking both types of commits as propagated commits. Thus, for our purposes, we can identify commits integrated via Git rebase or Git merge, but do not differentiate between them. Similar to pull requests, both types of commits may be pulled from any of the branches to the other. However, unlike pull requests, it is not possible to identify which variant the propagated commit originated from. This is because of the nature of distributed version-control systems where commits can be in multiple repositories, but there is no central record identifying the commits' origin. Since it is common for commits to be pulled from the mainline and pushed into the fork repository as a result of the fork trying to keep in sync with the new changes in the mainline, we make an assumption that all commits that we identify as integrated through git merge or git rebase are pulled from the mainline variant and pushed into the fork variant.

[10] https://developer.github.com/v3/pulls/

3.3.2 Identifying Unique Commits

To identify the unique commits between the mainline and fork we use the GitHub API `compare`.[11] This API compares between the mainline branch and fork branch, as one of the items, return the diverged commits that comprise the number of commits a given branch (say mainline branch) is ahead of the other branch (fork branch) as well the number of commits the branch is behind the other. The commits that the mainline branch is ahead of the fork branch are the unique commits to the mainline, while the commits the mainline is behind the fork are the unique commits to the fork.

3.3.3 Verifying our Commit Categorization Methods

We verify our methods of identifying common commits for the different commit propagation techniques discussed in Section 3.3.1 in two phases: we first test our scripts on six toy projects we created ourselves, where we intentionally include at least one example of each commit propagation technique and verify that the commits are correctly categorized. Second, we manually analyze some of the results of our scripts from a sample of six real mainline–fork pairs that are part of our data collection from each ecosystem, and which we provide all details for in our online appendix [1]. From earlier version of this work in the conference paper [11], we noticed integrated pull requests between mainline and the variant forks were very rare. To this end, when testing our scripts, in addition to the variant forks which have very a limited number of integrated commits, we also use social forks that have lots of integrated commits with their mainline counterparts. In this section, we will discuss only the following 3 pairs, which we show in Table 2:

- (`dashevo / dash-wallet, sambarboza / dash-wallet`): The repository `sambarboza / dash-wallet` is a *social fork*. The mainline `dashevo / dash-wallet` has a total 445 PRs. Our scripts identifies that 74 of these 445 pull requests were integrated from the fork repository `sambarboza / dash-wallet` into the mainline repository `dashevo / dash-wallet`. We show the details of these 74 PRs in Table 2. Our technique identified that 3 of the 74 PRs were integrated using the PR merge option (all together having a total of 13 commits). There were 43 of the 74 PRs that were integrated using PR squash option (having a total of 194 commits), 2 of the 74 PRs used the PR rebase option having a total of 6 commits, and the integration option of the 26 PRs was unclassified (having a total of 167). We identified a total of 405 commits that were integrated using the *git* merge / rebase integration option and no commit was integrated using *git* cherry-pick option.
- (`flagbug / YoutubeExtractor, Kimmax / SYMMExtractor`): The repository `Kimmax / SYMMExtractor` is a *variant fork*. The mainline `flagbug / YoutubeExtractor` has a total of 32 pull requests. Our scripts identifies that 2 of the 32 PRs were integrated from the fork repository `Kimmax / SYMMExtractor` into the mainline repository (`lagbug / YoutubeExtractor` (see details in Table 2). The two PRs were integrated using the merge PR option having a total of two commits that were integrated. We also identified a total of three commits that were integrated using the *git* merge / rebase integration option and 1 commit was integrated using *git* cherry-pick option.

[11] https://docs.github.com/en/rest/reference/repos#compare-two-commits

- (`TerriaJS / terriajs, bioretics / rer3d-terriajs`): The repository `bioretics / rer3d-terriajs` is a *variant fork*. The fork `bioretics / rer3d-terriajs` has a total of 10 pull requests. Our scripts identifies that 9 of the 10 pull requests were integrated from the mainline `TerriaJS / terriajs` into the fork `bioretics / rer3d-terriajs`. The 9 PRs had a total of 101 commits. There were no commits integrated using the PR squash and PR rebase options. A total of 1,825 were integrated using the option *git* merge / rebase integration option and only 10 commits integrated using *git* cherry-pick option.

Given the above results of our scripts, we select some of the identified code propagation techniques and manually verify them. For each analyzed mainline–fork pair, we randomly sample a pull request from each identified pull request integration technique that were returned by our scripts. We manually analyze those sampled pull requests and their commits, including the commit metadata to verify the correctness of the identified propagation technique. For each of these sampled pull requests, we also randomly select two commits and manually analyze them to make sure they have been correctly classified. For example, in the pair [`getodk / collect (D), lognaturel / collect (S)`] (`lognaturel / collect` is a social fork), our script reveals that the commits in the pull requests numbered 3531, 3462 and 3434 were integrated using merging, squashing and rebasing, respectively. We manually verify that these pull requests have been in fact integrated using these techniques by looking at their commit metadata. Similarly, for the pair [`dashevo / dash-wallet (D), sambarboza / dash-wallet (S)`] (`sambarboza / dash-wallet` is a social fork), we verify that the commits in the pull requests number 421, 333, and 114 were integrated using merging, squashing, and rebasing, respectively. We also look at the results returned by integration outside GitHub (`git merge/rebase` and `git cherry-pick`). For example, our results indicate that the pair [`FredJul/Flym (D), Etuldan/spaRSS (S)`] (`Etuldan/spaRSS` is a variant fork), has no commits integrated using pull requests but had 34 and five commits integrated using `git merge/rebase` and `git cherry-picking`, respectively. We manually verify these five latter commits and confirm their correctness.

As the pair `dashevo / dash-wallet, sambarboza / dash-wallet` from Table 2 shows, there were some pull requests that our scripts were not able to classify. As part of our manual verification, we find that the GitHub API indicates that they are integrated into the destination repository since their `merge_date` is not `null`. On deeper investigation, we discover that all the unclassified pull request commits were integrated into a different branch from the `master` branch. For example, pull requests 514 and 512 from the fork `sambarboza / dash-wallet` were both integrated in the branch `evonet-develop` on the mainline repository. We also observed that both pull requests had an integration build test failure (`Travis CI`). This explains why the commits are missing in the history of the `master` branch and why our scripts could not classify those integrated commits.

One would wonder if we have a threat to construct validity since we do not consider the commit integration into other branches other than the default (main/master). For example, the scenario we presented above of unclassified pull requests that were integrated in the development branch ("staging"), but that were missing in the main branch since they failed the integration build test. If any of the 167 are integrated from the staging branch into the master branch using any of the integration techniques that do not completely rewrite the commit history

Table 2 Sample mainline–fork pairs showing numbers of integrated commits through different integration techniques. The first two mainline–fork pairs in the table we have S = source (fork) and D = destination (mainline). The last mainline–fork pair we have S = source (mainline) and D = destination (fork)

		Technique	# PRs	# Commits
		Android		
dashevo / dash-wallet (D), sambarboza / dash-wallet (S)	PR	Merged	3	13
		Squashed	43	194
		Rebased	2	6
		Unclassified	26	167
	Git	Merge/rebase		405
		Cherry-pick		0
		Total	74	785
		.NET		
flagbug / YoutubeExtractor (D), Kimmax / SYMMExtractor (S)	PR	Merged	2	2
		Squashed	0	0
		Rebased	0	0
		Unclassified	0	0
	Git	Merge/rebase		3
		Cherry-pick		1
		Total	2	6
		JavaScript		
TerriaJS / terriajs (S), bioretics / rer3d-terriajs (D)	PR	Merged	9	101
		Squashed	0	0
		Rebased	0	0
		Unclassified	0	0
	Git	Merge/rebase		1,825
		Cherry-pick		10
		Total	9	1,936

(i.e., PR merge /squash / rebase, *git* merge / rebase / cherry-pick), then our script would always identify them as commits that were integrated between the mainline and the fork using the *git* merge / rebase option. As such, our script minimizes the threat to validity of the unclassified pull requests.

Our manually verified data for both the toy projects and the real projects gives us confidence that our scripts can correctly identify the commits integrated through different integration mechanisms in any mainline–fork pair of any repository.

3.3.4 Fork Variability Percentage

To quantify how much a fork differs from its mainline, we define a metric *variability percentage* as follows:

$$VariabilityPercentage = \frac{unique_{FV}}{(unique_{FV} + CommonCommits)} \times 100 \quad (1)$$

where $CommonCommits$ = *Pull Request commits* + *Git commits* + *InheritedCommits* as shown in Figure 3. *VariabilityPercentage* measures the percentage of unique commits in a fork, when compared to all the commits in that fork. A lower percentage means that most of the changes in the fork are either starting commits (i.e., the fork did not make many changes after the fork date) or merged commits that are propagated from/to the mainline. Both these cases indicate that the functionality in the fork is not much different/variable from that in the mainline. On the other hand, a higher *VariabilityPercentage* indicates more specific customizations in the fork.

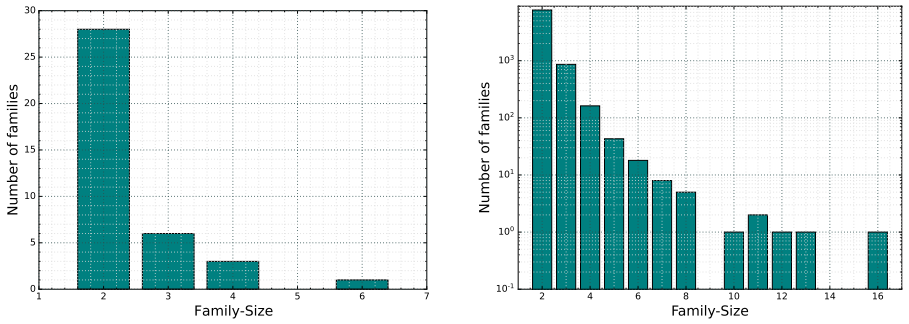
4 Variant Family Characteristics (RQ1)

We now present the characteristics of our identified software families within the ecosystems. Table 3 shows all the metrics we defined with values.

Table 3 Metrics characterizing our families

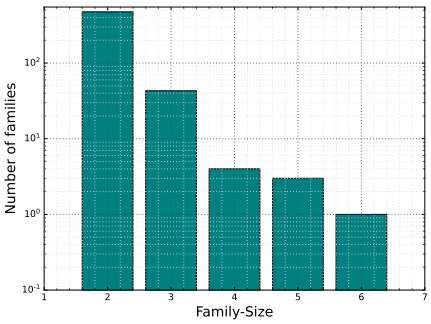
Metric	Mean	Min	Median	Max	Description
			FamilySize		
Android apps	2.4	2	2	7	Number of variants in an Android family
.NET apps	2.1	2	2	7	Number of variants in a .NET family
JavaScript apps	2.2	2	2	16	Number of variants in a JavaScript family
			App Dependencies (.NET & JavaScript)		
PackageDependenciesMLV	40.4	0	26	140	Number of mainline variant packages dependencies on Android
	2.3	0	1	49	Number of mainline variant packages dependencies on .NET
	11.8	0	7	267	Number of mainline variant packages dependencies on JavaScript
PackageDependenciesFV	22	0	22	81	Number of of fork variant packages dependencies on Android
	2.0	0	1	25	Number of of fork variant packages dependencies on .NET
	9.8	0	6	605	Number of fork variant packages dependencies on JavaScript
			App Popularity (Android)		
DownloadsMLV	2,211K	1	50K	100M	Number of downloads of the mainline variant from Google Play
DownloadsFV	5,479K	5	1K	100K	Number of downloads of the fork variant from Google Play
ReviewsMLV	27K	0	547	631K	Number of reviews of the mainline variant on Google Play
ReviewsFV	2.8K	0	45	161K	Number of reviews of the fork variant on Google Play
			App Popularity (.NET & JavaScript)		
DependentPackagesMLV	106	0	0	27K	Number of packages that depend on the mainline app on .NET
	80	0	2	26K	Number of packages that depend on the mainline app on JavaScript
DepenednPackagesFV	0.4	0	0	19	Number of .NET packages that depend on the fork app on .NET
	1.7	0	0	2K	Number of JavaScript packages that depend on the fork app on JavaScript
DependentProjectsMLV	133	0	0	33K	Number of .NET projects that depend on the mainline app on GitHub
	140	0	0	83K	Number of JavaScript projects that depend on the mainline app on GitHub
DependentProjectsFV	0.5	0	0	82	Number of .NET projects that depend on the fork app on GitHub
	2	0	0	5K	Number of JavaScript projects that depend on the fork app on GitHub
			App Maintenance (.NET & JavaScript)		
PackageReleasesMLV	14.6	1	2	188	Number of mainline variant packages dependencies on .NET
	15	1	8	1117	Number of mainline variant packages dependencies on JavaScript
PackageReleasesFV	3.6	1	2	54	Number of of fork variant packages dependencies on .NET
	4	1	2	341	Number of fork variant packages dependencies on JavaScript

[MLV] mainline variant [FV] forked variant



(a) Android Families

(b) JavaScript Families

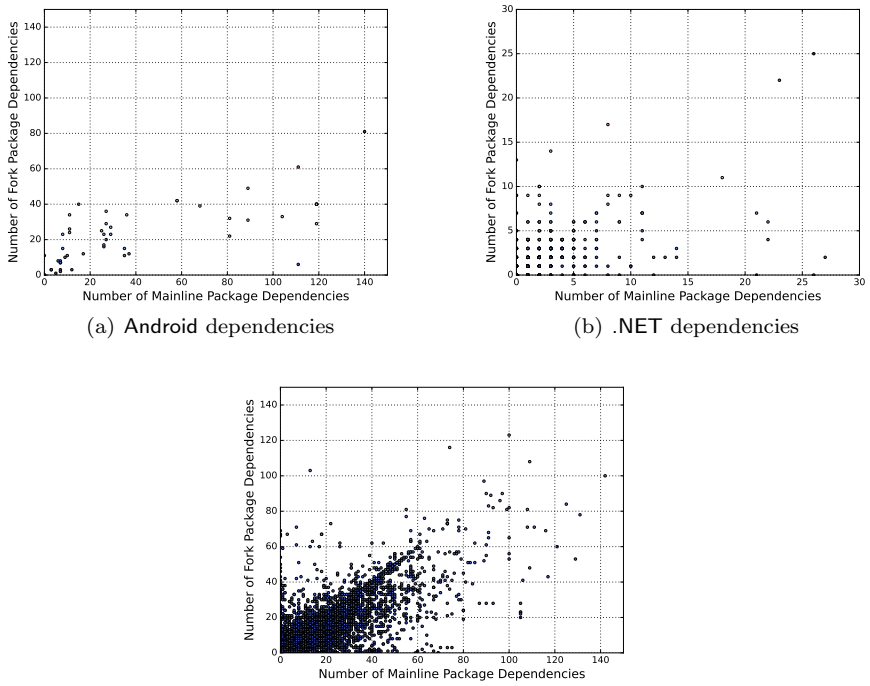


(c) .NET Families

Fig. 4 Distribution of family sizes (number of variants in a family) of the three ecosystems. A variant family contains one mainline variant and at least one or more fork variants. The presented data corresponds to *38 software families*, *8,837 software families*, and *526 software families*. Note that y-axes of Fig. 4(b) and Fig. 4(c) are presented on logarithmic scales. The axes of figures are also presented on different scales for visibility purposes.

4.1 General Variant Characteristics

- *Variant Family FamilySize*. Figure 4 shows the number of variants (i.e., family size) in each of the variant families of the three ecosystems we studied.
  We can see that the distributions of family sizes for all three ecosystems are right-skewed with most families having two members. Specifically, 28 (73%) of *38 software families*, 7,731 (87%) of *8,837 software families*, and 475 (90%) of *526 software families* have only two variants. The three distributions also show that larger families are rather seldom in all three ecosystems, but that the largest family sizes we observe are part of the JavaScript ecosystem. When identifying variant families from the different ecosystems, we observe that although Android is considered one of the largest known ecosystems [36, 39, 53], identifying its variant families is rather difficult compared to the software packaging ecosystems (JavaScript and .NET) we studied. In the Android ecosystem is not compulsory to record any source repository of an Android variant on Google Play. To this end, we went through the lengthy process described in Section 3.1.1, applying a number of heuristics on GitHub repositories to identify families.
- *Variant Package Dependencies*: In Fig. 5, we present two scatter plots showing the graph of mainline dependencies versus the fork dependencies. Figure 5(a) to



(a) Android dependencies

(b) .NET dependencies

(c) JavaScript dependencies

Fig. 5 Scatter plots of mainline and fork variant dependencies of other packages on the ecosystems. The datasets mainline–fork variants of *54 mainline–fork variant pairs* for Android, *590 mainline–fork variant pairs* for .NET and *10,357 mainline–fork variant pairs* for JavaScript. **Note**: *The graphs are presented on different scales for visibility purposes.*

Fig. 5(c) show the scatter plots of the number fork variant package dependencies (y-axis) versus the number of mainline variant package dependencies (x-axis) for Android, .NET and JavaScript variants, respectively. A point in any of the scatter plots represents the number of package dependencies of a given fork variant (y-axis) and the number of package dependencies of the counterpart mainline variant (x-axis). In all scatter plots, its not surprising that the number of package dependencies for a fork and its corresponding mainline are correlated. This confirms that fork variants inherit the original dependencies of the mainline. However, we also observe points in all the scatter plots where one variant has more dependencies than the other. This means that the variant with more packages dependencies has functionality that is not included in the counterpart variant. Although the observation is more prominent for the mainline variant since we see many points below the diagonal lines for the two graphs (the forks do not keep in sync with the mainline), it is interesting that we also have some fork variants with more dependencies. Follow-up studies could investigate what and why new functionalities related to the used dependencies are being introduced in the variants.

- *Android variant categories*:

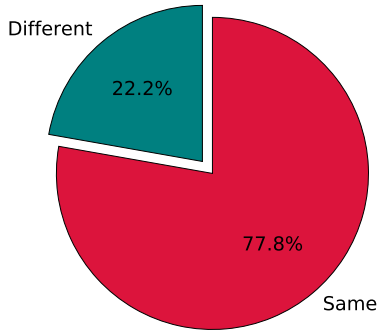


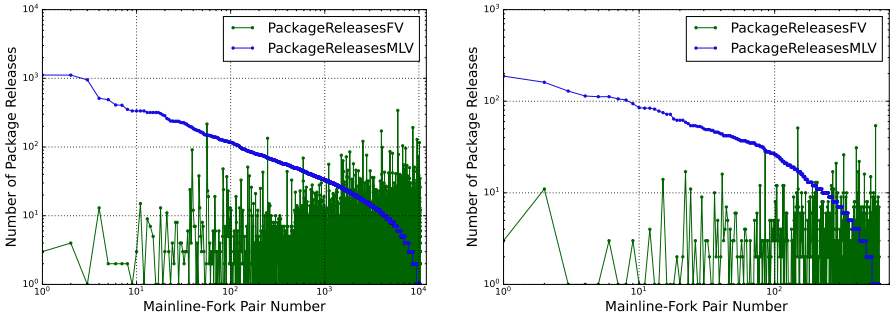
Fig. 6 Relationship between the variant categories listed on Google Play for each variant in the Android Mainline–Fork Pairs. *Same* = mainline–fork variant pairs share the same category and *Different* = mainline–fork variant pairs share different category.

Figure 6 shows the distribution of variants in the different categories on Google Play. We can see that 12 of the 54 forks (22%) are listed in a different category from the mainline, which suggests that these variants serve different purposes. However, the majority of pairs include variants in the same category.

4.2 *Variant Maintenance Activity (JavaScript & .NET)*

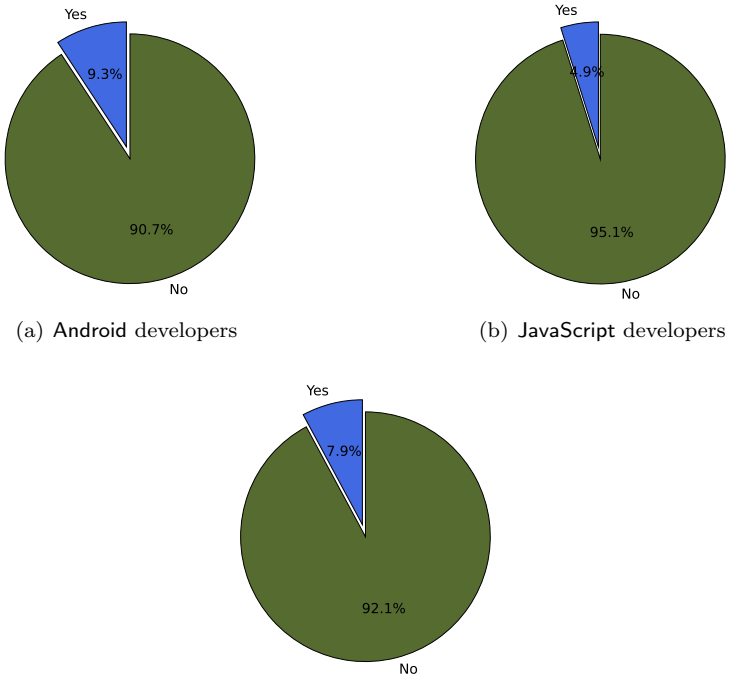
Figure 7 shows the release distributions for both the mainline and the fork variants in the JavaScript and .NET ecosystems. Each point on the x-axis represents a pair, and we sort the pairs by the number of mainline package releases. Figure 7(a) shows that the majority of mainline variants has multiple releases. Specifically, 5,888 of the 8,835 (67 %) mainline variants have ≥ 5 package releases on the JavaScript package manager. The fork variants have fewer, but still multiple releases. Specifically, 2,389 of the 10,357 mainline variants (23 %) have ≥ 5 package releases on the JavaScript package manager. Interestingly, from the plot we also observe a number of forks having more releases than their mainlines. Looking at Fig. 7(b), for .NET variants, we observe a similar distribution like that of JavaScript variants in Fig. 7(a). These results are interesting, since they indicates that developers of forked variants usually do not make a one off package distribution. They are continuously distributing new releases of their packages, further emphasizing that these are indeed variant forks.

***Observation 1–RQ1**: Families in fact exist in our three software ecosystems. We collected 38, 526, and 8,837 different families. While both the mainlines and forks have multiple releases, the number of releases is significantly higher than those of the forks. Still it indicates that the latter are usually not one-shot releases; with some having even more than their mainlines.*



(a) JavaScript (b) .NET

Fig. 7 Number of releases of mainlines (each blue point is a mainline) and their forks (each green point is a fork belonging to the mainline on the respective x coordinate) for the ecosystems of JavaScript and .NET.



(a) Android developers (b) JavaScript developers

(c) .NET developers

Fig. 8 Variant owners for the mainline–fork variant pair for the three ecosystem. Yes = mainline–fork variant pair has common developers and No = mainline–fork variant pair do not have common developers. The datasets of mainline–fork variant pairs of 54 from Android, 985 from JavaScript, and 89 from .NET ecosystems. **Note**: *The graphs are presented on different scales for visibility purposes.*

4.3 Variant Ownership Characteristics

Figure 8 shows the percentage of common owners in the mainline–fork variant pairs of our three studied ecosystems. For the Android variants the analysis is based on all the data we collected (54 mainline–fork variant pairs). However, for the .NET and JavaScript variants we only analysed a subset of the .NET and JavaScript mainline–variant pairs, respectively, due to the criteria we set out to identify variant ownership in Section 3.2. From Fig. 8, we can see relatively the same percentages of the common (Yes) and not common (No) developers across the three ecosystems. Overall, our results imply that the majority of forked variants are started and maintained by developers different from those maintaining the mainline counterparts.

> ***Observation 2–RQ1****: The majority of the mainline–fork variant pairs for the three ecosystems we investigated are owned by different developers (91 % for Android variants, 95 % of JavaScript variants and 92 % of .NET variants). This implies that the majority of forked variants in our datasets are started and maintained by developers different from those maintaining the mainline counterparts.*

4.4 Variant Popularity Characteristics

Figure 9 shows the variant popularity for the variants in the three software packaging ecosystems of Android, JavaScript, .NET.

- *Android variants*: Figure 9(a) shows the variant downloads distribution for both the mainline and fork variants where each point on the x-axis represents a pair and we sort the pairs by the number of mainline downloads. We observe that the majority of the mainline variants are quite popular, 27 of the 38 mainline variants (71%) have ≥10K downloads. For fork variant popularity in terms of downloads, we observe that 10 of the 54 fork variants (19%) having ≥ 10K downloads. We believe it is natural that the mainline variants are more popular than their fork counterparts, since we assume they have been released first on Google Play[12]. Figure 9(b) shows the variant reviews distribution for both the mainline and fork variants where each point on the x-axis represents a pair and we sort the pairs by the number of mainline reviews. We observe a similar distribution for number of reviews like those observed in the number of downloads. This is not surprising since previous studies have found downloads and reviews to be correlated [10]. Overall, the variant popularity we observe gives us confidence that our data set consists of real variants.
- *JavaScript and .NET variants*: In Fig. 9(c)–9(f) we present the popularity graphs for the variants in the two ecosystems of .NET and JavaScript. Figure 9(c) shows the dependent packages distributions for both the mainline and fork variants where each point on the x-axis represents a pair and we sort the pairs by the number of mainline dependent packages. We observe that the majority of mainline variants are quite popular, 6,157 of the 10,357 mainline variants

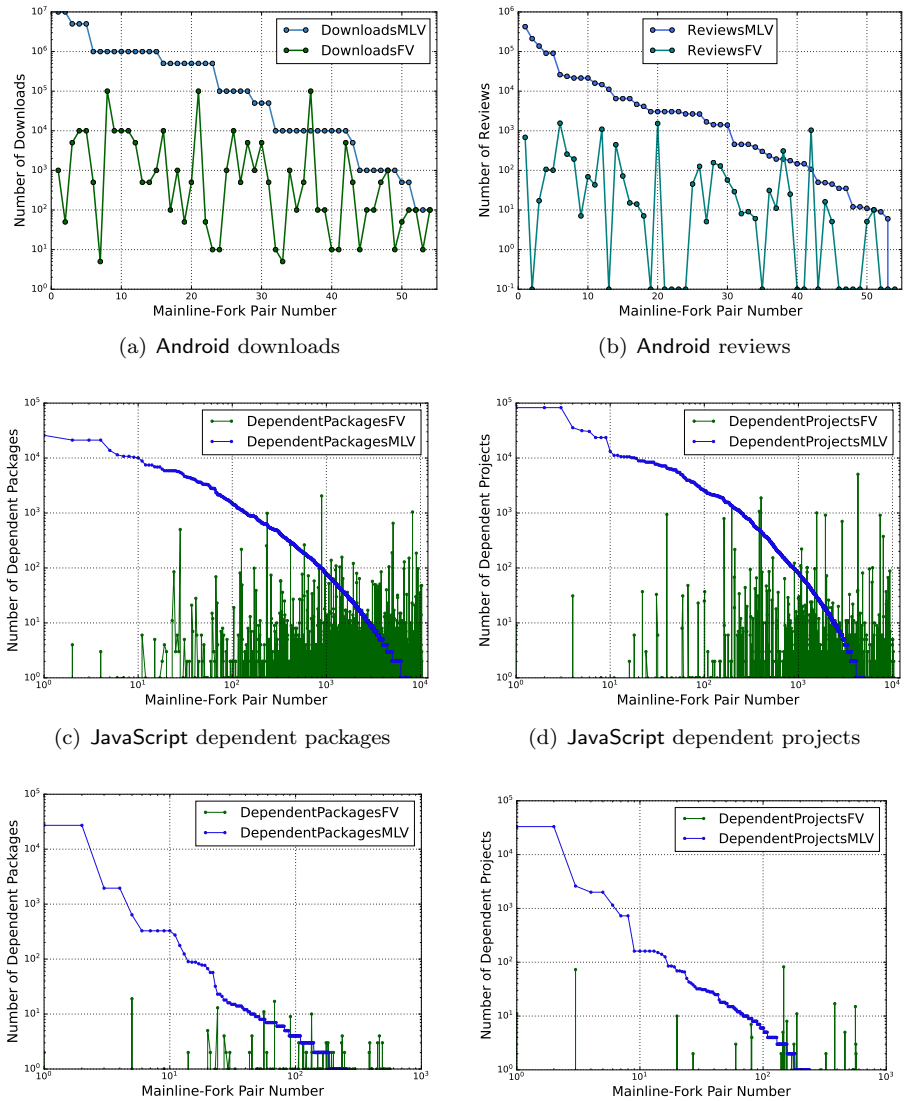
[12] Note that Google Play does not keep release history of its variants so it is not possible to obtain the first listing date of each variant

(59 %) having at least two dependent packages. For fork variants, we observe that 1,624 of the 10,357 mainline variants (16 %) having at least two dependent packages. Figure 9(d) shows the dependent projects distributions for both the mainline and fork variants for the variants in the JavaScript ecosystem. Each point on the x-axis represents a pair and we sort the pairs by the number of mainline dependent project. We also observe a similar distribution for number of dependent projects such as that observed in the number of dependent packages. The remaining two graphs, Fig. 9(e) and Fig. 9(f), show the same data for the .NET ecosystem, and both show similar trends to those observed for JavaScript.

Comparing the popularity of all the ecosystems, we observe that the mainline variants are more popular than the fork variant counterparts. This is not surprising since the forks are clones of the mainline. However from Fig. 9, in all the three ecosystems, its interesting to observe a few fork variants being more popular than their mainline counterparts. In a follow-up study it would be interesting to investigate possible explanations why the variants are more popular than their mainline counter parts. Comparing the popularity of the variants in the JavaScript and .NET ecosystems, we observe that on average the variants in the JavaScript ecosystem are more popular than the variants in the .NET ecosystem. We also observe that the fork variants in the .NET in the ecosystem are less popular (have fewer dependent packages/projects) than the variants in the JavaScript ecosystem. In a follow-up study it would also be interesting to investigate why variants in JavaScript families are more popular than the variants in .NET families and also why the fork variant variants in the JavaScript families are more popular than the fork variant variants in the .NET families.

Table 4 Example of mainline–fork variant pairs from the Android ecosystem showing statistics on the app popularity.

mainline	fork	mainline downloads	fork downloads	mainline reviews	fork review
TobyRich / app-smartplane-android	TailorToys / app-powerup-android	10K	100K	106	1,034
opendatakit / collect	kobotoolbox / collect	1,000K	100K	3,049	1,527



(a) Android downloads (b) Android reviews

(c) JavaScript dependent packages (d) JavaScript dependent projects

(e) .NET dependent packages (f) .NET dependent projects

Fig. 9 Distributions of mainline and fork variant variants' popularity metrics for the variants in the three ecosystems of Android, JavaScript and .NET. The datasets of *54 mainline–fork variant pairs* for Android, *10,357 mainline–fork variant pairs* for JavaScript, and *590 mainline–fork variant pairs* for .NET ecosystems.

Table 5 Example of mainline–fork variant pairs from the .NET and JavaScript ecosystems showing statistics on the popularity and maintenance activities.

	mainline	fork	mainline dependent packages	fork dependent packages	mainline package releases	fork package releases
.NET	Flurl.Signed	Flurl.Http.Signed	3	10	6	10
	Ninject	Portable.Ninject	638	19	75	14
JS	selenium	selenium-server	97	2,046	2	51
	gulp-istanbul	gulp-babel-istanbul	5,867	11	24	14

[JS] JavaScript

Table 4 and Table 5 present a few examples showing the variant popularity (for all the three ecosystems) and variant maintenance activities (for only .NET and JavaScript). In Table 5 columns `mainline` and `fork` we use the `package_names` of the variants since repository names on GitHub were too long. In both tables, we present two interesting examples of variant pairs that we randomly picked: (1) *abandoned mainlines:* the first variant pair in each of the ecosystems has the fork variant more popular that the mainline. When we compared the last release dates of the variants in all the ecosystems, we observed that the mainlines seem to have been abandoned while the fork variant continued to evolve. This is the reason the fork variants are more popular. In Table 5 we can also see that the fork variants have more releases than the mainlines. (2) *Co-evolution:* the second pair in each of the ecosystems we present another interesting case of co-evolution of both the mainline and fork variant. are continuously being maintained and where both are popular. In this cases, it would be interesting co-evolution of the variants in both technical and social aspects. *Technical*: for example investigating if the variants are complementary or they are competing? *Social*: What can we learn about the variant communities?

Observation 3–RQ1*: Although the mainline variants are more popular, which is not surprising, there is quite a number of fork variants that are also popular. We also observe a few of the fork variants being more popular than their mainline counterparts. This again tells us the forks we are studying are indeed variant forks being used by the community of other developers (in the cases of .NET and JavaScript variants) and for Android variants, being downloaded and installed on user phones. We have pointed out some interesting research directions that can be investigated in follow-up studies.*

5 Code Propagation in the Software Families (RQ2)

So far, we have analyzed the characteristics of the software families in across our three ecosystems. Our results from RQ1 give us confidence that the fork variants in our data set are indeed variant forks. In RQ2, we present the results of how variants in the same family co-evolve. Specifically, we are interested in their code propagation practices to understand if the variants evolve separately or if they propagate code between each other after the forking date. We present the results of code propagation between family variants in terms of propagated commits, while differentiating the propagation mechanisms we explained in Sections 2 and 3.3. Recall that these commit types determine the various code propagation strategies (e.g., pull requests versus direct integration through git). Tables 6–9 show the metrics we use in this RQ to measure the types of propagated commits in the ecosystems of Android, JavaScript, and .NET. Where applicable, we specify the direction of the propagated code, i.e., mainline→fork or fork→mainline. Recall from Section 3.3.1 that we do not differentiate between `git merge` and `git rebase` commits and that we assume that all integrated `git merge` and `git rebase` commits are in the direction mainline→fork. This is why Tables 7 and Table 8 show only one metric $gitPull_{MLV\text{-}FV}$ to represent these two commit integration types. Tables 6–9 show the summary of the descriptive statistics of all the metrics we use to investigate code propagation at the commit level for all the three ecosystems of Android, JavaScript, and .NET.

5.1 Pull Request Propagation (Commit Integration Inside GitHub)

We present the results of the pull request integration techniques: merge, rebase and squash (as well as the unclassified PRs) for the mainline–fork pairs in all the three ecosystems of Android, JavaScript, and .NET. In Table 6 the results of the summary statistics and in Table 7 we present the details of the summary statistics. We also present the distributions of the integration in both directions in Fig. 10.

Figure 10 shows the box plots showing the distributions of the different PR integration techniques. For example for the variants in the Android ecosystem, the distribution of the PR integration in both directions of mainlines→fork and fork→mainline are shown in Fig. 10(a). There was only one pull request in each direction of integration. Both pull requests were integrated using the PR merge option. There was no PR integrated using any of the other PR integration options. We can see that in all the boxplots the majority of the mainline–fork variant pairs have zero PRs integrated in either direction. This implies that most of the pairs do not integrate PRs between themselves.

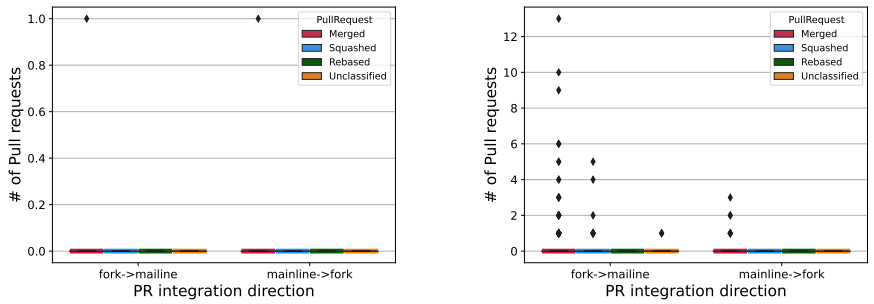
Table 7 shows the details of summary statistics in the distributions. For example, in the top section of Table 7 (Android variants) and in the first row, we observe 1 of the 54 mainline–fork variant pairs that integrated 1 PR having a total of 5 commits, using the `merge` pull request option, in the direction of mainline→fork. In the same row, in the direction of fork→mainline, we observe 1 mainline–fork pair that integrated 2 PRs, having a total of 427 commits, using the `merge` pull request option, in the direction of fork→mainline.

We can see that for Android variants only 1 of the 54 (1.9 %) mainline–fork pairs integrated commits using the `merge` pull request option. We observe more or less sim-

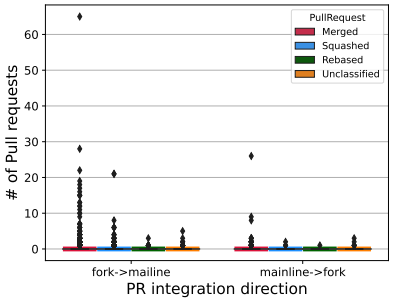
Table 6 Pull request (inside GitHub) code propagation practices, at commit level, for the *54 mainline–fork variant pairs*, *10,357 mainline–fork variant pairs*, and *590 mainline–fork variant pairs* in the Android, JavaScript, .NET ecosystems, respectively.

Metric	Mean	Min	Median	Max	Description
			Android variants		
$mergedPRs_{MLV\text{-}FV}$	0.31	0	0	15	Number of merged PR from the mainline to the fork variant.
$mergedPRs_{FV\text{-}MLV}$	0.09	0	0	4	Number of merged PR from a given the fork to the mainline variant.
$prMergedCommits_{MLV\text{-}FV}$	8.33	0	0	427	Number of merged PR commits from the mainline to the fork variant.
$prMergedCommits_{FV\text{-}MLV}$	0.57	0	0	28	Number of merged PR commits from the fork to the mainline variant.
$prSquashed_{MLV\text{-}FV}$	0	0	0	0	Number of squashed PR from the the mainline to the fork variant.
$prSquashed_{FV\text{-}MLV}$	0	0	0	0	Number of squashed PR from a given the fork to the mainline variant.
$prRebased_{MLV\text{-}FV}$	0	0	0	0	Number of rebased PR from the the mainline to the fork variant.
$prRebased_{FV\text{-}MLV}$	0	0	0	0	Number of rebased PR from a given the fork to the mainline variant.
			.NET variants		
$mergedPRs_{MLV\text{-}FV}$	0	0	0	3	Number of merged PR from the mainline to the fork variant.
$mergedPRs_{FV\text{-}MLV}$	0.2	0	0	13	Number of merged PR from a given the fork to the mainline variant.
$prMergedCommits_{MLV\text{-}FV}$	0.2	0	0	30	Number of merged PR commits from the mainline to the fork variant.
$prMergedCommits_{FV\text{-}MLV}$	1.2	0	0	207	Number of merged PR commits from the fork to the mainline variant.
$prSquashed_{MLV\text{-}FV}$	0	0	0	0	Number of squashed PR from the the mainline to the fork variant.
$prSquashed_{FV\text{-}MLV}$	0	0	0	5	Number of squashed PR from a given the fork to the mainline variant.
$prSquashedCommits_{FV\text{-}MLV}$	0.1	0	0	14	Number of squashed PR commits from the fork to the mainline variant.
$prRebased_{MLV\text{-}FV}$	0	0	0	0	Number of rebased PR from the the mainline to the fork variant.
$prRebased_{FV\text{-}MLV}$	0	0	0	0	Number of rebased PR from a given the fork to the mainline variant.
			JavaScript variants		
$mergedPRs_{MLV\text{-}FV}$	0	0	0	26	Number of merged PR from the mainline to the fork variant.
$mergedPRs_{FV\text{-}MLV}$	0.4	0	0	4	Number of merged PR from a given the fork to the mainline variant.
$prMergedCommits_{MLV\text{-}FV}$	0.1	0	0	399	Number of merged PR commits from the mainline to the fork variant.
$prMergedCommits_{FV\text{-}MLV}$	0.57	0	0	28	Number of merged PR commits from the fork to the mainline variant.
$prSquashed_{MLV\text{-}FV}$	0	0	0	2	Number of squashed PR from the the mainline to the fork variant.
$prSquashed_{FV\text{-}MLV}$	0	0	0	21	Number of squashed PR from a given the fork to the mainline variant.
$prSquashedCommits_{MLV\text{-}FV}$	0.4	0	0	52	Number of squashed PR commits from the mainline to the fork variant.
$prSquashedCommits_{FV\text{-}MLV}$	0	0	0	109	Number of squashed PR commits from the fork to the mainline variant.
$prRebased_{MLV\text{-}FV}$	0	0	0	2	Number of rebased PR from the the mainline to the fork variant.
$prRebased_{FV\text{-}MLV}$	0	0	0	3	Number of rebased PR from a given the fork to the mainline variant.
$prRebasedCommits_{MLV\text{-}FV}$	0.4	0	0	4	Number of rebased PR commits from the mainline to the fork variant.
$prRebasedCommits_{FV\text{-}MLV}$	0	0	0	25	Number of rebased PR commits from the fork to the mainline variant.

ilar trends for the mainline–fork variants pairs in the other two ecosystems. For the JavaScript mainline–fork variant pairs, we observe 99 of the *10,357 mainline—fork variant pairs* (1 %) integrating commits, using the `merge` pull request option, in the direction of mainline→fork and 724 of the *10,357 mainline–fork variant pairs* (7 %) in the direction of fork→mainline. We observe very few mainline–fork variant pairs, in the JavaScript software packaging ecosystem, integrating commits using the pull request `squash/rebase` options in either integration directions. For the mainline–fork variant pairs in the .NET ecosystem, we observe 9 of *590 mainline–fork variant pairs* (1.5 %) and 67 of the *590 mainline–fork variant pairs* (11.3 %) integrating commits, using the `merge` pull request option, in the direction of mainline→fork and fork→mainline, respectively. We did not observe any commits integrated using the `rebased` pull request option in either integration direction, while for the commits integrated using the `squash` pull request option, we only observed integration in the direction of fork→mainline accounting for 13 of the *590 mainline–fork variant pairs* (2 %).



(a) Android (b) .NET developers



(c) JavaScript developers

Fig. 10 Distribution of pull requests in both integration directions for variants in the three ecosystems. The datasets of *54 mainline–fork variant pairs*, *590 mainline–fork variant pairs*, and *10,357 mainline–fork variant pairs* from the ecosystems of Android, .NET, and JavaScript, respectively. **Note**: *The graphs are presented on different scales for visibility purposes.*

We observe that there are more mainline–fork variant pairs integrating commits in the direction of fork→to mainline as opposed to mainline→fork irrespective of the PR integration option used. For Android variants we observed 1 pair each in either direction (1.9 % each); for JavaScript variants we have 867 of *10,357 mainline–fork variant pairs* (8.4 %) in the direction of fork→mainline to 105 of *10,357 mainline–fork variant pairs* (14 %). Regarding the pull request integration options, we can see that the `merge` pull request option is clearly the most frequently used in all integration directions and in all the three ecosystems. In all three software packaging ecosystems, the `squash` and `rebase` options are rarely used. However, comparing the two PR options, `squash` and `rebase`, we observe that the `squash` PR option is used more often.

Observation 1–RQ2: *Code propagation using PRs is rarely used in all the mainline–fork variant pairs from the three ecosystems that we studied. Unsurprisingly, we have observed that PRs in the direction of fork→mainline are more than those in the direction of mainline→fork. However, although low numbers are observed, there are some PRs in the direction of mainline→fork. We have also observed that, in all the three ecosystems, the most used integration option is by far the* `merge` *PR option. The* `squash` *and* `rebase` *PR option are less frequently used*

Table 7 Number of mainline–fork pairs, pull requests involved in code propagation in our dataset of *54 mainline–fork variant pairs*, *10,357 mainline–fork variant pairs*, and *590 mainline–fork variant pairs* from the ecosystems of Android, JavaScript, and .NET, respectively. For example, the Android apps, the first row in the direction of mainline→fork, only 1 fork variant merged 1 PR from the mainline containing 5 commits and in the direction of fork→mainline, only 1 mainline merged 2 PRs containing 427 commits.

		Mainline→Fork			Fork→mainline		
		Pairs	PRs	Commits	Pairs	PRs	Commits
		Android variants					
PR	Merged	1	1	5	1	2	427
PR	Rebased	0	0	0	0	0	0
PR	Squashed	0	0	0	0	0	0
PR	Unclassified	0	0	0	0	0	0
Git	Cherry-pick	5	n/a	250	4	n/a	136
Git	$gitPull_{MLV\text{-}FV}$	18	n/a	13,198	n/a	n/a	n/a
		.NET variants					
PR	Merged	9	13	96	67	139	721
PR	Rebased	0	0	0	0	0	0
PR	Squashed	0	0	0	13	21	72
PR	Unclassified	0	0	0	3	3	9
Git	Cherry-pick	15	n/a	99	16	n/a	138
Git	$gitPull_{MLV\text{-}FV}$	106	n/a	5,601	n/a	n/a	n/a
		JavaScript variants					
PR	Merged	99	162	1,862	724	1,394	4,523
PR	Rebased	1	1	4	11	13	67
PR	Squashed	5	6	72	132	250	1,048
PR	Unclassified	7	10	33	23	32	134
Git	Cherry-pick	95	n/a	275	91	n/a	251
Git	$gitPull_{MLV\text{-}FV}$	1,180	n/a	40,001	n/a	n/a	n/a

in mainline–fork variant pairs all the three ecosystems, although the `squash` *PR option is more used that the* `rebase` *PR option. The low numbers could be attributed to the fact that the fork variants are created not to submit PRs but to diverge away from the mainline to solve a different problem. A follow-up study involving a user study could investigate motivation behind fork variant are creation and why there is limited collaboration between mainline and fork variants.*

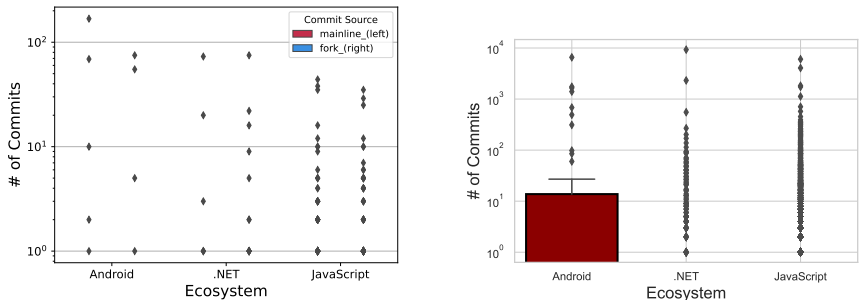
Table 8 Git based (outside GitHub) code propagation practices, at commit level, for the *54 mainline–fork variant pairs*, *10,357 mainline–fork variant pairs*, and *590 mainline–fork variant pairs* in the Android, JavaScript, .NET ecosystems, respectively.

Metric	Mean	Min	Median	Max	Description
			Android variants		
$gitCherrypicked_{MLV\text{-}FV}$	4.6	0	0	168	Number of git cherry-picked commits from the the mainline to the fork variant.
$gitCherrypicked_{FV\text{-}MLV}$	2.5	0	0	75	Number of git cherry-picked commits from the fork to the mainline variant.
$gitPull_{MLV\text{-}FV}$	244	0	0	6567	Number of git merged/rebased commits from the the mainline to the fork variant.
			.NET variants		
$gitCherrypicked_{MLV\text{-}FV}$	1.5	0	0	42	Number of git cherry-picked commits from the the mainline to the fork variant.
$gitCherrypicked_{FV\text{-}MLV}$	0.4	0	0	148	Number of git cherry-picked commits from the fork to the mainline variant.
$gitPull_{MLV\text{-}FV}$	9.5	0	0	2,317	Number of git merged/rebased commits from the the mainline to the fork variant.
			JavaScript variants		
$gitCherrypicked_{MLV\text{-}FV}$	4.6	0	0	168	Number of git cherry-picked commits from the the mainline to the fork variant.
$gitCherrypicked_{FV\text{-}MLV}$	0	0	0	70	Number of git cherry-picked commits from the fork to the mainline variant.
$gitPull_{MLV\text{-}FV}$	3.7	0	0	6,035	Number of git merged/rebased commits from the the mainline to the fork variant.

5.2 Git Propagation (Commit Integration Outside GitHub)

In this section we present the results of commit integration outside GitHub relating to *git* `cherry-pick` and *git* `merge` / `rebase` ($gitPull_{MLV\text{-}FV}$). The summary statistics of these two commit integration techniques are presented in Table 8. In Table 7, the detailed results corresponding to the summary statistics in Table 8 are presented We first present the results of *git* `cherry-pick`, and we follow with the results of *git* `merge` / `rebase`.

- *git* `cherry-pick` *commit integration*: Like we stated in Section 3.3 commits can be cherry-picked from mainline in two directions: mainline→fork or fork→mainline. The two metrics: $gitCherrypicked_{MLV\text{-}FV}$ and $gitCherrypicked_{FV\text{-}MLV}$ (in Table 8) corresponding to the two commit integration directions for the mainline→fork and fork→mainline, respectively, in the three ecosystems. In Fig. 11 we present boxplot distributions corresponding to the results in Table 8. We can see all the distributions only show outliers, meaning that most pairs do not have cherry-picked commits. The detailed statistics in Table 7 reveal the same results. For example, the upper part of Table 7 presenting the Android variants, we can see that there are only of 5 of the *54 mainline–fork variant pairs* (9 %) that integrated a total of 250 commits in the direction of mainline→fork. In the direction of fork→mainline there were 4 of the *54 mainline–fork variant pairs* (7.4 %) integrating a total of 136 commits. Like the results of pull request integration presented earlier, we can also clearly see that commit integration using *git* `cherry-pick` is rarely used in the mainline–fork variant pairs in all the three ecosystems we have studied. Unlike pull request integration where the developer has to sync upstream or downstream the new changes, with *git* `cherry-pick` the developer have to search for specific commits to integrate. This



(a) *git* cherry-pick (b) *git* merge/rebase

Fig. 11 Distribution of commits integrated outside GitHub. The datasets of *54 mainline–fork variant pairs*, *10,357 mainline–fork variant pairs*, and *590 mainline–fork variant pairs* from the ecosystems of Android, JavaScript, and .NET, respectively. **Note**: *The graphs are presented on different scales for visibility purposes.*

requires to first look into the pool of new changes and identify the ones of interest to cherry-pick. If the mainline and fork variant have diverged solving different problems, then finding the interesting commits in the new changes might be laborious. We hypothesize that this could be one of the reasons why there are few numbers of commits observed in mainline–fork variant pairs in the three ecosystems. A follow up study to confirm or refute this hypothesis would add value to this study.

- *git* `merge` / `rebase` *commit integration*: In Table 8 we can see metric $gitPull_{MLV\text{-}FV}$ representing the the *git* `merge` / `rebase` commit integration in the direction of mainline→fork, in the three ecosystems. Again we can see that the all the medians for all the metric in all the three ecosystems are all zeros. Figure 11 shows three boxplots showing the distributions of $gitPull_{MLV\text{-}FV}$ metric for the mainline–fork variant pairs in the three ecosystems. From the boxplots, we can also observe that the medians are all zeros. In Table 7 we present the detailed statistics for the metric $gitPull_{MLV\text{-}FV}$. For Android mainline–fork variant pairs, we observe 18 of the *54 mainline–fork variant pairs* (33 %) with a total of 13,198 commits being integrated in the direction of mainline→fork. For .NET mainline–fork variant pairs, we observe 106 of the *590 mainline–fork variant pairs* (18 %) with a total of 5,601 commits being integrated in the direction of mainline→fork. And finally for JavaScript mainline–fork variant pairs, we observe 1,180 of the *10,357 mainline–fork variant pairs* (11 %) with a total of 40,001 commits being integrated in the direction of mainline→fork. We can see that although *git* `merge` / `rebase` still rarely used in the mainline–fork variants in all the three ecosystems, it is more used than the other two options of pull requests and *git* `cherry-pick`. We can conclude that `git merge / rebase` is the most used code integration mechanism between the variants in variant families. Again, we speculate that the lack of integration mainline–fork variant pairs could be as a results of the variants diverging to solve different problem from those being solved by their mainline counterparts.

Table 9 Unique commits and variability percentage for the *54 mainline–fork variant pairs*, *10,357 mainline–fork variant pairs*, and *590 mainline–fork variant pairs* in the Android, JavaScript, .NET ecosystems, respectively.

Metric	Mean	Min	Median	Max	Description
			Android variants		
$unique_{MLV}$	1,122	0	228	18,961	Number of unique commits in the mainline variant in a given mainline–fork variant pair.
$unique_{FV}$	98.3	1	16	1,646	Number of unique commits in the fork variant in a given mainline–fork variant pair.
InheritedCommits	1,884	10	755	29,110	Number of common commits between a given fork and the mainline variant.
VariabilityPercentage	15	0	2.7	93.8	Percentage of unique commits according to Equation 1.
			.NET variants		
$unique_{MLV}$	102.2	0	3	10,789	Number of unique commits in the mainline variant in a given mainline–fork variant pair.
$unique_{FV}$	16.2	0	5	605	Number of unique commits in the fork variant in a given mainline–fork variant pair.
InheritedCommits	224.5	0	42.1	20,538	Number of common commits between a given fork and the mainline variant.
VariabilityPercentage	20	0	11	99	Percentage of unique commits according to Equation 1.
			JavaScript variants		
$unique_{MLV}$	33.5	0	3	10,223	Number of unique commits in the mainline variant in a given mainline–fork variant pair.
$unique_{FV}$	12.8	0	5	1,229	Number of unique commits in the fork variant in a given mainline–fork variant pair.
InheritedCommits	111.5	14	32	66,861	Number of common commits between a given fork and the mainline variant.
VariabilityPercentage	22.3	0	14	99	Percentage of unique commits according to Equation 1.

***Observation 2–RQ2**: Like the integration technique using PRs, we also observe that `git merge/rebase` and `git cherry-pick` integration techniques are also less frequently used in the variants in the three ecosystems. However, we observe that integration using `git merge/rebase` is the most commonly used integration mechanism between the mainline–fork varianOverts in all the three ecosystems which occurs in the integration direction of mainline→fork. In general a follow-up study to investigate why most variants do not share code would reveal reasons for the low numbers of integration.*

5.2.1 Fork Variability Percentage

This section present the results of variability percentage (metric *VariabilityPercentage*) for the fork variants in the three ecosystems. In Table 6 we present the summary statistics for the metrics used to calculate *VariabilityPercentage* in Equation 1. Figure 12 presents the distributions of the metric *VariabilityPercentage* of the fork variants in the three ecosystems. We can see that the medians are 2.7 %, 11 %, and 14 %, forks variants in the three ecosystems of Android, .NET, and JavaScript, respectively. A high value of the metric *VariabilityPercentage* implies that the fork differs from its mainline counterpart. For the fork variants in the Android ecosystem, we observe

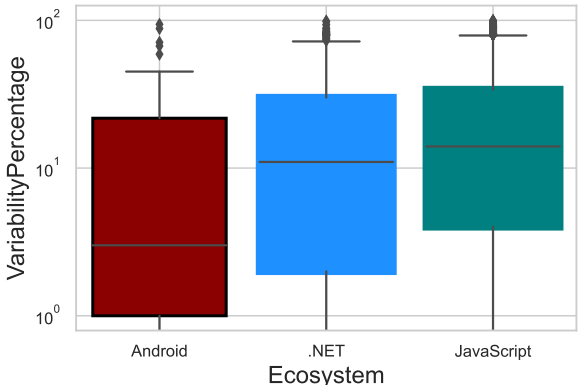


Fig. 12 Distribution of fork variability percentage–*VariabilityPercentage* for the variants in the three ecosystems. The datasets of 54 fork variants, 10,357 fork variants, and 590 fork variants from the ecosystems of Android, JavaScript, and .NET, respectively.

quite a number of the forks, 35 of the 54 (35 %), have a high *VariabilityPercentage* (>= 10%). The fork variants from the .NET ecosystem, we also observe the majority of the forks, 281 / 590 (53 %), have a high *VariabilityPercentage* (< 10%). Lastly, the fork variants in the JavaScript ecosystem, we also observe quite the majority of the forks, 6,076 / 10,357 (58 %), have a relatively high *VariabilityPercentage* (< 10%).

> ***Observation 3–RQ2:*** *The majority of the fork variants in the three ecosystems of Android, JavaScript, and .NET highly differ from their mainline counterparts (i.e., they have higher numbers of unique commits). The findings of forks variants differing from their mainlines could be used to support our earlier finding relating to limited commit integration in the mainline–fork variant pairs in the three ecosystems.*

5.3 Summary

We have presented results of code propagation practices among mainline–fork variant pairs from the three ecosystems of Android, .NET, and JavaScript. Overall, in all the studied mainline–fork variant pairs of the three ecosystems, we observe infrequent code propagation, regardless of the type propagation mechanism or direction. The most used code propagation technique is `git merge/rebase`, which is used in 33 % of Android mainline-fork pairs, 11 % of JavaScript pairs, and 18 % of .NET pairs. For integration using pull requests, developers often integrate code in the direction of fork→mainline compared to those in the direction of mainline→fork, in all the mainline–fork variants. The code integration in the direction of mainline→fork is often done using the `merge` pull request option or `git merge/rebase` outside GitHub. Moreover, the `squash` and `rebase` pull request options are less frequently used in mainline–fork variant pairs, although the `squash` PR option is more used than the `rebase` pull request option. Finally, by comparing the fork variability percentage, we observed a high percentage difference between the fork variants and

their mainline counterparts, indicated by the higher number of unique commits. These results are consistent across all the variants of the three ecosystems (i.e., Android, JavaScript, and .NET) that we studied. Our findings potentially indicate that the fork variants are being created with the intention of diverging away from the mainline to solve a different problem (i.e., with no intention to sync in any way with the original mainline). Future studies could investigate the motivation behind fork variants' creation and why there is a limited collaboration between mainline and fork variants.

6 Discussion and Implications

The observations from our two research questions have several implications for future research on co-evolution of software families and for respective tool support.

Implications for Identifying Variant Forks. As opposed to previous studies that relied on heuristics applied to GitHub repositories to identify Variant forks, in this study, we ensure that all members of a variant family represent different variants in the marketplace (Google Play, JavaScript, and .NET). Relying on only heuristics applied to GitHub repositories to find variant forks may have false positives (i.e., fork classified as a variant fork, yet it is a social fork). The method for identifying divergent forks can be reused by other researchers interested in studying variant families in other ecosystems, including operating-system packages (e.g., Debian packages [4]) and ecosystems established for other programming languages. In fact, most of the popular programming language today, such as JavaScript `Java`, `PHP`, `.NET`, `Python`, and many more have their own package managers available that host hundreds of thousands of packages. More details on the package managers can be found on Libraries.io which is a platform we have used to identify and extract details about variant families from the JavaScript and .NET ecosystem. Libraries.io references packages from over 37 package managers where one can obtain software families in the different ecosystems.

Implications for Forking Studies. Observation 1–RQ2 and Observation 2–RQ2 suggest that, in our studied divergent forks, direct integration using *git* outside of GitHub is more commonly used than GitHub pull requests. *This implies that simply relying on pull requests to understand code propagation practices in divergent forks is not enough.* Furthermore, it seems that integration using `git rebase` is common, as per Observation 2–RQ2. Rebasing complicates the git history and empirical studies that do not consider rebasing may report skewed, biased and in accurate observations [47]. Thus, in addition to looking beyond pull requests when studying code propagation, *studies must also consider rebased commits.* In this paper, we contribute reusable tooling for identifying these rebased commits.

Implications for Integration Support Tools. Regardless of the integration technique used, our findings based on the variants from the three ecosystems studied suggest that code propagation rarely happens between a fork and its mainline. In our datasets, we observe 35 % of *54 mainline–fork variant pairs*, 21 % of *590 mainline–fork variant pairs*, and 11.5 % of *10,357 mainline–fork variant pairs* that integrated commits using at least one of the commit integration techniques in

the three ecosystems of Android, .NET, and JavaScript, respectively. The lack of integration may be problematic, since the fork variants may rely on the correct functionality of the existing code from the mainline. This means that any bugs that exist in the mainline will also exist in these forks, unless bug fixes are propagated from one variant to the other. However, current integration techniques [32, 34, 37] do not necessarily facilitate finding such bug fixes. For example, code integration using pull requests and `git merge / rebase` may not be the best when integrating changes in variant forks since they involve syncing upstream / downstream all the changes missing in the current branch. Alternatively, cherry picking is probably more suitable for bug fixes since the developer can choose the exact commits they want to integrate. However, GitHub's current setup does not make it easy to identify commits to cherry-pick with out digging through the branch's history to identify relevant changes since the last code integration. As a result of the difficulty of finding commits to cherry-pick, developers may end up fixing the same bugs, which would result in duplicated effort and wasted time. To check if a possible duplication of effort occurs in our data set, we looked at the unique commits of the variants and indeed found that developers independently update files shared by the variants. For example, in the mainline–fork variant pair (`k9mail / k-9, imaeses / k-9`) the shared file `ImapStore.java`[13] has been touched by 15 different developers in 142 commits in the mainline variant while in the fork variant it has been touched by one developer in 9 different commits. It is possible that these developers could be fixing similar bugs existing in these shared artifacts. Moreover, the study of Jang et al. [27] reports that during the parallel maintenance of cloned code, a bug found in one clone can exist in other clones, thus, it needs to be fixed multiple times. Furthermore, as a result of different developers changing shared files, it is possible that these developers do not integrate code because of "**fear of merge conflict**." In relation to this conjecture, several studies have reported that merging diverged code between repositories is very laborious as a result of merge conflicts [6, 38, 49, 54–57]. To this end, it would be interesting for future research to interview the developers of our forks (and further forks) to determine whether the lack of support for cherry picking bug fixes or specific functionality does indeed contribute to the lack of code propagation. In that case, developing a patch recommendation tool that can inform developers of possible interesting changes as soon as they are introduced in one variant and recommend them to other variants in a family can help save developers' efforts. The recent work by Ren et al. [51] that focused on providing the mainline with facilities to explore non-integrated changes in forks to find opportunities for reuse is one step towards this direction. Our work opens up more opportunities for applying such tools since, as mentioned above with respect to identifying divergent forks, we provide a technique for identifying such forks by combining information from GitHub and the ecosystem's main delivery platform as well as we mention various other ecosystems where a similar strategy can be adopted. Finally, the limited sharing of changes can give rise to quality issues. We did not specifically investigate the propagation of test cases, which might not be propagated as well. Developing techniques for propagating test cases within families could significantly enhance the quality of variants within families. The potential of test-case propagation has recently been pointed out in a preliminary study by Mukelabai et al. [40].

[13] src/com/fsck/k9/mail/store/ImapStore.java. Same path for both mainline and fork.

Implications for Future Research: Our work is the first to perform a large-scale empirical study on the practices used to manage software families within software ecosystems. Our results give rise to the following open research questions that could be addressed as follow up studies to further understand the evolution of such families.

1. *More than two variants in a family:* In the results of RQ1, we showed that there are quite a number of families that had a *FamilySize* of more than two variants (i.e., mainline with two ore more fork variants). However, in this study we only concentrated on the practices used to manage mainline-fork pairs. For example, we did not look at fork-fork pairs in a given family or looking at the holistic evolution the families that have more than two variants. It would be interesting to extend the study to those families study the evolution of the family.
2. *Variant dependencies:* In RQ1, we observed that in some variant pairs in all the three ecosystems, one of the mainline or fork variant in the pair has more dependencies than the other. This implies that the variant that has more dependencies implements new functionality relating to the extra dependencies that are missing in the counterpart. It would be interesting to investigate what / why the new functionality is missing in the counterpart variant. Another interesting research relating to dependencies would be to investigate if there are some variants in a family that have updated their code to depend on new releases of the common dependencies, while other variants in the same family are still dependent on the old releases of the dependencies. Updating code to implement a new release of a dependency may involve fixing incompatibilities, especially if the new release of the dependency involves a breaking change. To avoid effort duplication, a tool could be developed that could help in transplanting patches (related to the incompatibility fixes), to other variants in the family that have not yet migrated their code to the new API-breaking change of the release of the common dependency.
3. *Limited sharing of changes in unique commits:* In RQ2 we have observed that there is limited sharing of the changes in the unique commits between the mainline–fork variant pairs in the three ecosystems. We hypothesized that one of the possible reasons could be the variants diverging from each other to solve different problems. We also stated that fork variants could be created to support a new technology, serve different community, target different content, to support a frozen feature in the mainline. Fork variants created for the above reasons are likely to have little to share with their mainline variants. It would be interesting to carry out a study involving mixed methods of quantitative and user studies to verify our hypothesis.
4. *Impediments in co-evolving variants in software families:* Like in the study of Robles and González-Barahona [52], in our dataset we also observed that some mainline–fork variant pairs continue to co-exist, while others one of the variants in the pair is abandoned as the other continues to evolve. A Follow-up study can be conducted to investigate the impediments to co-evolving these variants. Inspirations can be leveraged from the studies of the co-evolution of Eclipse platform and its third-party plug-ins [9, 12–16, 30].

7 Related Work

We discuss related work on (i) variant forking and on (ii) code propagation in forked projects, as well as we discuss (iii) general studies on forking.

7.1 Variant Forking

To understand the variants in our variant families, RQ2 explored the reasons forks were created. While there are existing studies on variant forks, most of these were done in the pre-GitHub days of `SourceForge`, before the advent of social coding environments [35, 43–45, 52, 60]. These studies reported controversial perceptions around variant forks in the pre-GitHub days [19, 22, 24, 42, 44, 50]. However, Zhou et al. [63] recently report that these perceptions have changed with the advent of GitHub. In the Pre-GitHub days, variant forks were frequently considered as risky to projects, since they could fragment a community and lead to confusion of developers and users. Jiang et al. [28] state that, although forking is controversial in the traditional open source software (OSS) community, it is encouraged and is a built-in feature in GitHub. The authors further report that developers carry out social forking to submit pull requests, fix bugs, add new features, and keep copies. Zhou et al. [63] also report that most variant forks start as social forks. Robles and González-Barahona [52] comprehensively study a carefully filtered list of 220 potential forks of different projects that were referenced on Wikipedia. The authors assume that a fork is significant if a reference to it appears in the English Wikipedia. They found that technical reasons and discontinuation of the original project were the most common reasons for creating variant forks, accounting for 27.3% and 20% respectively. More recently, Zhou et al. [63] interviewed 18 developers of variant forks on GitHub to understand reasons for forking in more modern social coding environments that explicitly support forking. The authors report that the motivations they observed align with the above prior studies.

All the above works studied forks for any type of project, not limited to a specific technological space (e.g., web applications or mobile apps). Our paper is different in that it focuses on Android apps, triangulating data from both GitHub and Google Play to study real-world apps. Specifically, we study variant reuse practices in RQ2 and, different from both studies ([63] and [52]), we investigate additional phenomena, such as code propagation with RQ3.

Another difference between the current study and the study of Zhou et al. [63] is the heuristics the two studies employ to determine variant forks. Zhou et al. [63] classify forks on GitHub as variant forks using the following heuristics: (i) contain the phrase “fork of” in the description, (ii) received at least three external pull requests, (iii) have at least 100 unique commits, (iv) have at least one year of development, and (v) have changed their name. In our work, we use the external validation of a fork being listed on Google Play under a different package name, and we use the description there to verify that this app is indeed a variant of the mainline.

7.2 Code Propagation Practices

There are only a few studies that investigated code integration between a given repository and its forks. Stanciulescu et al. [57] studied forking on GitHub using a case study of Marlin, an open source firmware for 3D printers. The authors observed that many forked variants share their changes with the mainline. However, their work does not differentiate between social and variant forks. Thus, we do not know whether this observed prevalent code propagation is simply due to the fact that these are social forks created with the main goal of contributing back to the original project [62]. In our current paper, we are interested only in variant forks. Recently, Zhou et al. [63] observed that only 16% of of their 15,306 studied variant forks ever syncronized or merged changes with their mainline repository. However, based on their discussed threats to validity, it seems that the authors relied only on common commit IDs to identify shared commits. As we explained in Section 2, there are several integration techniques that result in propagated commits having different commit IDs. Thus, relying only on the commit ID may result in missing other shared commits. To mitigate this problem, our work identifies integrated commits that preserve the commit ID as well as those that may have been integrated using techniques that change the commit ID. Another study on code propagation practices work of Kononenko et al [31]. The authors considered three types of commit integration: GitHub merge, cherry-pick merge and commit squashing. In comparison to our study, we only do not study commit squashing but we look at other techniques the authors did not consider like: GitHub rebase and squash pull requests as well as git merge and rebase

Code propagation practices do not necessarily have to be in the context of forks. For example, German et al. [26] investigated how Linux uses Git. The authors stated that code changes are variant to track because of the proliferation of code repositories and because developers modify ("rebase") and filter ("cherry-pick") the history of these changes to streamline their integration into the repositories of other developers. To this end, the authors presented a method *continousMining* that crawls all known git repositories of a project multiple times a day to record and analyze all change-sets of a project. They authors state that *continousMining* not only yields a complete git history, but also catches phenomena that are variant to study such as rebasing and cherry-picking. While we do not continuously capture the "live" history of a software project, we are able to capture rebased and cherry-picked commits in the context of forked projects by relying on the commit meta data, after a thorough investigation of how this meta data changes depending on the propagation strategy .

7.3 Other Studies About Forking.

Gamalielsson and Lundell [25] studied the long term sustainability of Open Source software communities in Open Source software projects involving a fork. The authors study was based on `LibreOffice` project, which is a fork from the `OpenOffice` project. They wanted to understand how Open Source software communities were affected by a forking. The authors undertook an analysis of the `LibreOffice` project and the related `OpenOffice` and `Apache OpenOffice` projects by reviewing documented project information and a quantitative analysis of project repository data as well

as a first hand experiences from contributors in the `LibreOffice` community. Their results strongly suggested a long-term sustainable `LibreOffice` community that had no signs of stagnation in the `LibreOffice` project 33 months after the fork. They also reported that good practice with respect to governance of Open Source software projects is perceived by community members as a fundamental challenge for establishing sustainable communities. Nyman [42] interviewed developers to understand their views on forking. His findings from the interviews differentiate *good forks*, which are those that (i) revive abandoned programs, (ii) experiment with and customize existing programs, or (iii) minimize tyranny and resolve disputes by allowing involved parties to develop their own versions of the program, vs. *bad forks*, which are those that (i) create confusion among users or (ii) add extra work among developers (including both duplication of efforts and increased work if attempting to maintain compatibility).

8 Threats to Validity

Internal Validity. We identify four issues that could threaten the internal validity of our results: (1) In Section 3.1, the heuristics used for app family data identification in Steps 2 & 6 resulted in mismatch in the mapping of some the forks on GitHub and Google Play. We mitigated the threat by carrying out a through manual analysis in Section 3.1–Step 7 and discarded the mismatched apps. Some of the steps we carried out during Android variant's data collection are manual, and any errors in those could affect our results. (2) Although we did not observe any cases where the developer changed the message in cherry-picked commits, we acknowledge that our algorithm will not be able to identify such cases; instead, our algorithm will identify them as unique commits in the respective variants. (3) We also acknowledge that our tool chain may miss some commits that are integrated using more than one integration technique. For example in Section 3.3.3, we presented the *unclassified merged pull requests*, which were listed on the GitHub API as merged yet they were not merged in the master branch. We discovered that the pull requests were integrated in a different branch other than the mainline but had all failed the build integration tests. To this end, when integrating commits from a fork→mainline, as a "best practice", developers may wish to first integrate the commits into a different branch (say staging branch) perform and integration test and then later integrate them into the master. However, following the "best practice" we have explained, if the developer first integrates into the development branch using one commit integration technique. Thereafter the developer may wish integrate the same commits into the master using a different technique that changes the original integrator's metadata (for example cherry-picking). In that case, our toolchain will miss such commits. (4) In Section 2.2, we also stated that our scripts are not able to identify the integrated commits if the integrator uses git commands that rewrite the commit history. However, like we stated in Section 3.3.3, we believe that the practice of rewriting contributions from the community is likely to be rare with experienced developers, since rewriting changes commit authorship. (5) In Step 6 of Section 3.1, we eliminated all Android mainlines that did not at least one fork having a different package name on Google Play store. This means that we eliminate fork variants that were created for different markets other than Google play. However, unlike Google play where one can use an app's package name as a

unique ID on Google play, other markets, such as `anzhi, apkmirror, appsapk` do not implement this strategy which means we cannot easily identify the correct app for a given GitHub repository. Therefore, we intentionally focus only on Android apps that are distributed on Google play store, which limits the number of Android families we are able to identify.

Construct Validity. The calculation of variability percentage of the fork variants treats commits the same way irrespective of the number of files touched. For example, a commit that has touched 100 files is treated the same as one that has just touched on file. While this may be misleading, the measure provides some indication of unique development activity.

External Validity. We analyzed only 54 Android mainline–fork variant pairs while there exists millions of android applications on Google Play and other Android markets, which means that our results might not be representative of all the Android applications. However, we also analyze mainline–fork variant pairs from two other ecosystems that also show similar results and behavior.

9 Conclusion

We presented a large-scale exploratory study on reuse and maintenance practices via code propagation between variant forks and their mainline counterparts in software ecosystems. Our subject ecosystems cover different technological spaces: `Android`, `JavaScript`, and `.NET`. As part of our study, we designed a systematic method to identify real variant forks as well.

We identified and analyzed families of variants that are maintained together and that exist both on the official package distribution platforms (Google play, nuget, and npm) as well as on GitHub, allowing us to analyze reuse practices in depth. For variants in a given ecosystem, we mined from both sources of information—from GitHub and the package distribution site—to study their characteristics, including their variations, and code-propagation practices. In the Android ecosystem we identified *38 software families* with a total of *54 mainline–fork variant pairs*, in the .NET ecosystem *526 software families* with *590 mainline–fork variant pairs*, and in the JavaScript ecosystem *8,837 JavaScript software families* with *10,357 mainline–fork variant pairs*. We provide a toolchain for analyzing code integration between any mainline-fork variant pair. Regardless of the integration technique used, our findings suggest that code integration rarely happens between a fork and its mainline. In our study, in the Android ecosystem, we observed only 19 of the 54 (35 %) that integrated commits using at least one of the commit integration techniques we discussed. In the .NET ecosystem, we observed a total of 126 of the *590 mainline–fork variant pairs* pairs (21 %) that that integrated commits using at least one of the commit integration techniques. In the JavaScript ecosystem, we observe a total of 1,189 of the *10,357 mainline–fork variant pairs* pairs (11.5 %) that integrated commits using at least one of the commit integration techniques.

Overall, we analyzed variant forks on GitHub for two main reasons: (1) many previous studies focused on social forks, (2) the few studies on variant forks are conducted in the pre-GitHub days of `SourceForge`. In the future, it would be interesting to investigate a middle ground between the variant forks and social forks. For example, one could investigate if the practices observed in the variant forks are different from those of social forks.

Acknowledgements We thank Serge Demeyer for comments on earlier drafts of this work.

John Businge's work is supported by the FWO-Vlaanderen and F.R.S.-FNRS via the EOS project 30446992 SECO-ASSIST. Thorsten Berger's work is supported by Swedish research council and Wallenberg Academy. Sarah Nadi's research was undertaken, in part, thanks to funding from the Canada Research Chairs Program.

References

1. "online appendix". https://github.com/johnxu21/emse2020
2. 2020 GitHub, I.: "about pull request merges". https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-request-merges
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
4. Berger, T., Pfeiffer, R., Tartler, R., Dienst, S., Czarnecki, K., Wasowski, A., She, S.: Variability mechanisms in software ecosystems. Information & Software Technology **56**(11), 1520–1535 (2014)
5. Berger, T., Steghöfer, J.P., Ziadi, T., Robin, J., Martinez, J.: The state of adoption and the challenges of systematic variability management in industry. Empirical Software Engineering **25**, 1755–1797 (2020)
6. Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Proactive detection of collaboration conflicts. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, p. 168–178. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2025113.2025139. URL `https://doi.org/10.1145/2025113.2025139`
7. Businge, J., Decan, A., Zerouali, A., Mens, T., Demeyer, S.: An empirical investigation of forks as variants in the npm package distribution. In: M. Papadakis, M. Cordy (eds.) Proceedings of the 19th Belgium-Netherlands Software Evolution Workshop, BENEVOL 2020, Luxembourg, December 3-4, 2020, *CEUR Workshop Proceedings*, vol. 2912. CEUR-WS.org (2020). URL `http://ceur-ws.org/Vol-2912/./paper1.pdf`
8. Businge, J., Kawuma, S., Bainomugisha, E., Khomh, F., Nabaasa, E.: Code authorship and fault-proneness of open-source android applications: An empirical study. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, pp. 33–42. ACM, New York, NY, USA (2017). DOI 10.1145/3127005.3127009. URL `http://doi.acm.org/10.1145/3127005.3127009`
9. Businge, J., Kawuma, S., Openja, M., Bainomugisha, E., Serebrenik, A.: How stable are eclipse application framework internal interfaces? In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 117–127 (2019). DOI 10.1109/SANER.2019.8668018
10. Businge, J., Openja, M., Kavaler, D., Bainomugisha, E., Khomh, F., Filkov, V.: Studying android app popularity by cross-linking github and google play store. In: SANER (2019)
11. Businge, J., Openja, M., Nadi, S., Bainomugisha, E., Berger, T.: 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, pp. 625–634 (2018)
12. Businge, J., Serebrenik, A., van den Brand, M.: Compatibility prediction of eclipse third-party plug-ins in new eclipse releases. In: 12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012, pp. 164–173 (2012)
13. Businge, J., Serebrenik, A., van den Brand, M.: Survival of eclipse third-party plug-ins. In: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, pp. 368–377 (2012). DOI 10.1109/ICSM.2012.6405295
14. Businge, J., Serebrenik, A., van den Brand, M.: Analyzing the eclipse API usage: Putting the developer in the loop. In: 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013, pp. 37–46 (2013)
15. Businge, J., Serebrenik, A., van den Brand, M.G.J.: An empirical study of the evolution of Eclipse third-party plug-ins. In: EVOL-IWPSE'10, pp. 63–72. ACM (2010)

16. Businge, J., Serebrenik, A., van den Brand, M.G.J.: Eclipse API usage: the good and the bad. Softw. Qual. J. **23**(1), 107–141 (2015). DOI 10.1007/s11219-013-9221-3. URL https://doi.org/10.1007/s11219-013-9221-3
17. Chacon, S., Straub, B.: "git tools - rewriting history". https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History (2014)
18. Chacon, S., Straub, B.: Pro Git. Apress (2014)
19. Chua, B.B.: A survey paper on open source forking motivation reasons and challenges. In: R.A. Alias, P.S. Ling, S. Bahri, P. Finnegan, C.L. Sia (eds.) 21st Pacific Asia Conference on Information Systems, PACIS 2017, Langkawi, Malaysia, July 16-20, 2017, p. 75 (2017)
20. Czarnecki, K.: Overview of generative software development. In: J.P. Banâtre, P. Fradet, J.L. Giavitto, O. Michel (eds.) Unconventional Programming Paradigms, pp. 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
21. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Softw. Engg. **24**(1), 381–416 (2019). DOI 10.1007/s10664-017-9589-y. URL https://doi.org/10.1007/s10664-017-9589-y
22. Dixion, J.: "different kinds of open source forks – salad, dinner, and fish". https://jamesdixon.wordpress.com/2009/05/13/different-kinds-of-open-source-forks-salad-dinner-and-fish/ (2009)
23. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An exploratory study of cloning in industrial software product lines. In: CSMR (2013)
24. Ernst, N.A., Easterbrook, S.M., Mylopoulos, J.: Code forking in open-source software: a requirements perspective. ArXiv **abs/1004.2889** (2010)
25. Gamalielsson, J., Lundell, B.: Sustainability of open source software communities beyond a fork: How and why has the libreoffice project evolved? Journal of Systems and Software **89**, 128 – 145 (2014). DOI https://doi.org/10.1016/j.jss.2013.11.1077. URL http://www.sciencedirect.com/science/article/pii/S0164121213002744
26. German, D.M., Adams, B., Hassan, A.E.: Continuously mining distributed version control systems: An empirical study of how linux uses git. Empirical Softw. Engg. **21**(1), 260–299 (2016)
27. Jang, J., Agrawal, A., Brumley, D.: Redebug: Finding unpatched code clones in entire OS distributions. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, pp. 48–62. IEEE Computer Society (2012). DOI 10.1109/SP.2012.13. URL https://doi.org/10.1109/SP.2012.13
28. Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L.: Why and how developers fork what from whom in github. Empirical Softw. Engg. **22**(1), 547–578 (2017). DOI 10.1007/s10664-016-9436-6. URL https://doi.org/10.1007/s10664-016-9436-6
29. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: MSR (2014)
30. Kawuma, S., Businge, J., Bainomugisha, E.: Can we find stable alternatives for unstable eclipse interfaces? In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1–10 (2016). DOI 10.1109/ICPC.2016.7503716
31. Kononenko, O., Rose, T., Baysal, O., Godfrey, M., Theisen, D., de Water, B.: Studying pull request merges: A case study of shopify's active merchant. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, p. 124–133. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3183519.3183542. URL https://doi.org/10.1145/3183519.3183542
32. Krueger, J., Berger, T.: Activities and costs of re-engineering cloned variants into an integrated platform. In: 14th International Working Conference on Variability Modelling of Software-intensive Systems (VaMoS) (2020)
33. Krueger, J., Berger, T.: An empirical analysis of the costs of clone- and platform-oriented software reuse. In: 28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) (2020)
34. Krueger, J., Mahmood, W., Berger, T.: Promote-pl: A round-trip engineering process model for adopting and evolving product lines. In: 24th ACM International Systems and Software Product Line Conference (SPLC) (2020)
35. Laurent, A.S.: Understanding Open Source and Free Software Licensing . O'Reilly Media (2008)
36. Li, L., Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Mining families of android applications for extractive spl adoption. In: SPLC (2016)
37. Lillack, M., Stanciulescu, S., Hedman, W., Berger, T., Wasowski, A.: Intention-based integration of software variants. In: 41st International Conference on Software Engineering (ICSE) (2019)

38. Mahmood, W., Chagama, M., Berger, T., Hebig, R.: Causes of merge conflicts: A case study of elasticsearch. In: 14th International Working Conference on Variability Modelling of Software-intensive Systems (VaMoS) (2020)
39. Mojica, I.J., Adams, B., Nagappan, M., Dienst, S., Berger, T., Hassan, A.E.: A large scale empirical study on software reuse in mobile apps. IEEE Software **31**(2), 78–86 (2014)
40. Mukelabai, M., Berger, T., Borba, P.: Semi-automated test-case propagation in fork ecosystems. In: 43rd International Conference on Software Engineering, New Ideas and Emerging Results track (ICSE/NIER) (2021)
41. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017)
42. Nyman, L.: Hackers on forking. In: Proceedings of The International Symposium on Open Collaboration, p. 1–10 (2014)
43. Nyman, L., Lindman, J.: Code forking, governance, and sustainability in open source software. Technology Innovation Management Review **3**, 7–12 (2013)
44. Nyman, L., Mikkonen, T.: To fork or not to fork: Fork motivations in sourceforge projects. In: Open Source Systems: Grounding Research, pp. 259–268 (2011)
45. Nyman, L., Mikkonen, T., Lindman, J., Fougère, M.: Perspectives on code forking and sustainability in open source software. In: Open Source Systems: Long-Term Sustainability, pp. 274–279 (2012)
46. Openja, M., Adams, B., Khomh, F.: Analysis of modern release engineering topics : – a large-scale study using stackoverflow –. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 104–114 (2020). DOI 10.1109/ICSME46990.2020.00020
47. Paixão, M., Maia, P.: Rebasing in code review considered harmful: A large-scale empirical investigation. 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM) pp. 45–55 (2019)
48. Parnas, D.L.: On the design and development of program families. IEEE Trans. Softw. Eng. **2**(1), 1–9 (1976). DOI 10.1109/TSE.1976.233797. URL https://doi.org/10.1109/TSE.1976.233797
49. Perry, D.E., Siy, H.P., Votta, L.G.: Parallel changes in large-scale software development: An observational case study. ACM Trans. Softw. Eng. Methodol. **10**(3), 308–337 (2001). DOI 10.1145/383876.383878. URL https://doi.org/10.1145/383876.383878
50. Raymond, E.S.: The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary. O'Reilly Media, Inc. (2001)
51. Ren, L., Zhou, S., Kästner, C.: Poster: Forks insight: Providing an overview of github forks. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 179–180 (2018)
52. Robles, G., González-Barahona, J.M.: A comprehensive study of software forks: Dates, reasons and outcomes. In: Open Source Systems: Long-Term Sustainability, pp. 1–14 (2012)
53. Sattler, F., von Rhein, A., Berger, T., Johansson, N.S., Hardø, M.M., Apel, S.: Lifting inter-app data-flow analysis to large app sets. Automated Software Engineering **25**, 315–346 (2018)
54. Silva, L.D., Borba, P., Mahmood, W., Berger, T., Moisakis, J.: Detecting semantic conflicts via automated behavior change detection. In: 36th IEEE International Conference on Software Maintenance and Evolution (ICSME) (2020)
55. Sousa, M., Dillig, I., Lahiri, S.K.: Verified three-way program merge. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). DOI 10.1145/3276535. URL https://doi.org/10.1145/3276535
56. de Souza, C.R.B., Redmiles, D., Dourish, P.: "breaking the code", moving between private and public work in collaborative software development. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, GROUP '03, p. 105–114. Association for Computing Machinery, New York, NY, USA (2003). DOI 10.1145/958160.958177. URL https://doi.org/10.1145/958160.958177
57. Stanciulescu, S., Schulze, S., Wasowski, A.: Forked and integrated variants in an open-source firmware project. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), ICSME '15 (2015)
58. Sung, C., Lahiri, S.K., Kaufman, M., Choudhury, P., Wang, C.: Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '20, p. 172–181. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3377813.3381362. URL https://doi.org/10.1145/3377813.3381362

59. Vandehey, S.: Rebase and merge. https://cloudfour.com/thinks/squashing-your-pull-requests/ (2019)
60. Viseur, R.: Forks impacts and motivations in free and open source projects. International Journal of Advanced Computer Science and Applications - IJACSA **3**(2) (2012)
61. Zhou, S., Stănciulescu, c., Leßenich, O., Xiong, Y., Wasowski, A., Kästner, C.: Identifying features in forks. In: Proceedings of the 40th International Conference on Software Engineering, p. 105–116 (2018)
62. Zhou, S., Vasilescu, B., Kästner, C.: What the fork: A study of inefficient and efficient forking practices in social coding. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p. 350–361 (2019)
63. Zhou, S., Vasilescu, B., Kästner, C.: How has forking changed in the last 20 years? a study of hard forks on github. In: Proceedings of the 42nd International Conference on Software Engineering (2020). Accepted