Exploring trends and practices of forks in open-source software repositories

Mahsa Hadian
Polytechnique Montreal
Montreal, QC, Canada
mahsa.hadian@polymtl.ca

Soude Ghari
Polytechnique Montreal
Montreal, QC, Canada
soude.ghari@polymtl.ca

Marios Fokaefs*
Polytechnique Montreal
Montreal, QC, Canada
marios.fokaefs@polymtl.ca

Scott Brisson
University of Toronto
Toronto, ON, Canada
scott.brisson@mail.utoronto.ca

Ehsan Noei
University of Toronto
Toronto, ON, Canada
e.noei@utoronto.ca

Kelly Lyons
University of Toronto
Toronto, ON, Canada
kelly.lyons@utoronto.ca

Bram Adams
Queen's University
Kingston, ON, Canada
bram.adams@queensu.ca

Shurui Zhou
University of Toronto
Toronto, ON, Canada
shuruiz@ece.utoronto.ca

ABSTRACT

Forking a software repository is a popular and recommended practice among developers. A fork is a copy of the original repository that can evolve independently from the parent repository, allowing developers to experiment with a code base or test new features without the danger of affecting the original project. A fork can result in changes that are pushed back to the original project or even evolve into an independent project. Some projects tend to be forked extensively to the point where their forks are also forked and form families of projects. In this work, we explore the motivation, the practices and the culture of forking open-source software repositories. In particular, we study how forks evolve compared to the parent repository, how they are related to pull requests, how they contribute back to the parent, and how dependencies, in terms of libraries or external modules defined in a build script, are shared or differ within project families. Finally, we relate our findings with how communication and collaboration occurs within software families.

CCS CONCEPTS

• **Software and its engineering** → **Open source model**; *Software design engineering*; **Software libraries and repositories**; **Software development process management**.

KEYWORDS

mining software repositories, open source software, forks, software development processes, version control, dependencies, collaborative software development

*Currently, Pr Fokaefs is a faculty member at York University, Toronto, Canada

CASCON'22, November 15–17, 2022, Toronto, Canada

ACM Reference Format:
Mahsa Hadian, Soude Ghari, Marios Fokaefs, Scott Brisson, Ehsan Noei, Kelly Lyons, Bram Adams, and Shurui Zhou. 2022. Exploring trends and practices of forks in open-source software repositories. In *Proceedings of CASCON'22*. ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Forking is an open source practice that, until the early 2000s, was interpreted as a negative phenomenon: "There is strong social pressure against forking projects. It does not happen except under plea of dire necessity, with much public self-justification, and with a renaming" [12]. Typically, a subset of a given community would disagree about the development plan, progress or team composition of a given project, copy (the current state of) the code base elsewhere and continue separately from the initial community. Notable examples of such "hard" forks [20] are the forking of XEmacs from the main GNU Emacs in 1991 (both communities never merged again), or the EGCS fork from GCC in 1997 (successfully merged again into GCC 2.95). Today, the meaning of fork has softened slightly, thanks to the popularity of GitHub, and is mostly interpreted as "social" forking [20], which simply makes a copy of a (parent) repository in order to make modifications (new features, bug fixes or pure experimentation) to the code base that can then be proposed back to the parent (through a "pull request").

The notion of social forks goes much deeper than this. German et al. [6] note that modern forks basically are distributed branches of a code base that together form an ecosystem ("super-repository") around their parent, i.e., forks maintain traceability to their parent. Brisson et al. [2] went a step further, identifying the crucial role of social relationships between the contributors of a parent repository, its forks, and the forks' own forks to the extent that these forks essentially form a "family" of projects communicating directly through pull requests (PRs), issues, and mentions. Like any family, there might be small cliques that interact more closely than others, or some family members might break off from the family altogether.

It is not clear whether the motivational differences between hard and social forks are also reflected in different development practices. At a minimum, we are interested in understanding the degree of activity in both types of forks, and how long forks survive in either case. Do hard forks really have a much more difficult time succeeding, and, conversely, do social forks lead to substantial contributions? To what extent do such contributions tackle more invasive changes, such as updates of existing third party dependencies or the addition of new dependencies, and how receptive is the parent repository to such changes? Finally, does parent-fork communication (e.g., issues and comments) help determine the type (hard/social) and fate (success/failure) of forks?

To better understand the practices followed by social and hard forks, this paper presents an empirical study that analyzes 75 GitHub families of Java projects that include a total of 3,405 repositories (parents and forks). We selected these projects using criteria concerning their development activity (age in days, number of commits), the intensity of the communication (size of team, number of issues), the presence of build files (to specify dependencies) and others. We then set out to answer the following research questions:

RQ1) How do forks evolve alongside their parents?

RQ2) How are dependencies maintained and evolved between forks and their parents?

RQ3) How does communication between repositories, also within the context of families, relate to repository evolution?

The rest of the paper is organised as follows. Section 2 describes the data collection process and how the study was organised. In Section 3, we present and discuss the results of our study. Section 4 provides an overview of related literature. Finally, Section 5 concludes this work.

2 STUDY SETUP

2.1 Data Collection - Repositories

GitHub provides accessibility to its internal data storage through an API[1]. This API provides access to a rich collection of information about developers and repositories, and also provides valuable opportunities to understand forking behavior. As an extra check, we also used the GHTorrent data dump as a complementary data source for the same dataset [7]. In practice, the official API corresponds to the live current picture of GitHub, while the GHTorrent data corresponds to a static, more cumulative picture of the same data. In addition, GHTorrent is more complete with respect to communication and collaboration data. We aimed for the studied projects to exist in both sources to have as complete a dataset as possible. Zhou et al. [20] provide a dataset of 15,306 hard forks identified in GHTorrent's data dump from June 2019. Starting from the same data dump, we performed the following 10 steps to curate our dataset.

Identify and retrieve hard forks: We cross-referenced the 15,306 hard forks identified in the replication package of [20] with our GHTorrent data dump. Because the unique GHTorrent repository IDs were not provided, we cross-referenced the repository name and owner name in order to identify the hard forks in our dataset. Of 15,306 hard forks, 15,260 were identified and retained.

Identify parents from hard forks: The parents are needed to identify the entire software family for each repository. We recursively iterated through the data dump to find the parent (baseline repository) for each hard fork identified in the previous step. Five (5) hard forks were removed because the parents could not be identified, six (6) hard forks were removed because they were not forked using the GitHub forking mechanism (i.e., the fork button), but their code base was manually copied in a new repository, so their parents were not identifiable on GitHub which meant we could not link the evolution histories between parents and forks. Finally, one hard fork was removed because GHTorrent flagged the project as deleted, resulting in 12,985 parents with 15,248 hard forks.

Remove non-Java repositories: We opted to focus on Java projects since it is one of the most popular software languages, as identified by the TIOBE index [16], especially in the open source domain, and it possesses important properties we need for our analysis, namely build files for dependencies. Focusing on a single language can also help remove any variation in findings caused by this factor. Thus, considering only Java projects (repositories with language set to Java), we kept 1,038 parents with 1,201 hard forks.

Construct software families: Using the 1,038 parents that resulted from the previous step, we recursively iterated through each parent's forks and the forks of their forks, as identified in the GHTorrent data dump, resulting in 1,038 families composed of 367,353 repositories with 1,201 hard forks. During this process, we also kept track of which repositories were forks of the hard forks, resulting in 23,805 repositories within the 367,353 repositories that are forks of hard forks.

Remove pure clones: We define pure clones as forks that have been forked from a parent on GitHub, but contain no other change (in terms of commits). For the purpose of our study we removed pure clones among the children found in the previous step. This eliminated 289,501 repositories, resulting in 1,038 families composed of 77,852 repositories.

Remove personal repositories: Many repositories on GitHub are personal repositories, meaning they only have a single contributor. Since we are interested in the collaborative aspect of software development, we removed such repositories, resulting in 970 families composed of 15,725 repositories.

Remove inaccessible fork repositories: As we mentioned previously, parent repositories were identified using the GitHub API, but their children (i.e., forks) were gathered and evaluated using the GHTorrent dump, which contains more cumulative data and helped us to further filter repositories with low activity. As an extra check, we used the URLs in GHTorrent to call the GitHub API and confirm that our dataset contains "live" repositories, meaning not deleted or archived. As a result, we removed children from our dataset that were not accessible via the GitHub API. During this step, we also removed families whose hard forks may have been removed, bringing our dataset to 794 families composed of 11,162 repositories.

Remove repositories with no issues or PRs: Since we are interested in studying communication among repositories, we included a repository only if it used the collaborative tools on GitHub (i.e., issues and pull requests), resulting in 752 families composed of 7,976 repositories.

Remove non-software repositories with no build files: Many repositories on GitHub are not used for software development. These can include personal repositories, repositories that correspond to

[1] https://developer.github.com/v3/

web sites, or even documentation and manuscript repositories [9]. We distinguished these repositories from actual software repositories using the classification system similar to [9]. We manually inspected each repository and categorized each one as "software" or "other". Repositories categorized as "software" remained in our sample. In addition, given that our study is concerned with aspects of collaboration and communication in software development, where one analyzes artifacts in natural language, we also removed repositories whose dominant language was not English. This was done in tandem with identifying software repositories by inspecting the majority language used in the commit messages, pull requests, and issues. From these projects, we considered only those that had a build file, from which we could extract dependencies. The build files we considered were Gradle[2] and Maven[3], which are popular among Java projects. Thus, considering only English Java software repositories that have a build script, we kept 83 families composed of 3,857 repositories.

Remove small families: We removed families composed of fewer than 10 repositories (parent and forks) in order to reduce the risk of over-fitting [15]. We also removed families whose hard fork was removed during the previous filtration step, any duplicate repositories, and repositories that could not be accessed through GitHub's API. This resulted in a final dataset of 75 families composed of 3,405 repositories with 176 hard forks, with each family composed of at least 10 repositories with at least one hard fork.

2.2 Data Collection - Metrics

After finalizing the dataset with respect to parent repositories and their children, we gathered the following metrics necessary for answering our research questions (see Table 1):

Development Metrics: For **RQ1**, we are interested in studying the activity that takes place in forks. For that reason, we gathered data for all commits (including the commit messages) for all repositories. We also registered the date of creation for all forks and we calculated the age of each repository as the number of days from the creation day until 2019-06-01[4], which is the last date we considered. We also gathered data about the PRs (pull requests) for each repository. We only consider closed PRs. For Github, a closed PR is either a merged PR or a rejected PR. We do not distinguish between the two, because either of them show the intention of the fork to contribute back to the parent repository, which brings them closer to the "social" type of a fork. We did not consider open PRs because their status is unclear and we could not be sure about their progress (a PR can be retired by the submitter).

Dependencies: For **RQ2**, we are interested in how dependencies are maintained and evolve between forks and their parents. As described in Section 2.1, repositories in our dataset either use a Gradle (build.gradle) or Maven (pom.xml) build file that describes the project's dependencies and other resources needed by the project. In general, a Maven dependency is defined as a triplet of a `groupId`, which is the unique identifier of the general project that the library belongs to, an `artifactId`, which is the id of the specific library and a `version`. Normally, the `version` corresponds to a number or a version ID, but in some cases a variable may be used, which instructs Maven to download the latest version available in the project's repository.

Gradle is not always consistent in structure between projects because it provides different notations for specifying dependencies including a string notation and a map notation. Therefore, a combination of automatic parsing and manual inspection was used to extract dependencies from `dependencies` and `plugins` blocks of `build.gradle` files. In addition, the Gradle build file may include the declaration of dependencies on local binaries or file dependencies (i.e., a .jar file usually specified within a lib folder). For this case, we extracted the jar files with the corresponding full names. We used a python script to parse and identify the `artifactId`, `groupId`, and `version` declaration described within the dependencies and plugins blocks of the build file. The parsing resulted in a set of triplets that specified the library, the module, and the version of the module, similarly to Maven dependencies.

Communication Metrics: Communication metrics capture the social activity in a repository, including pull requests, comments on PRs, issues, users and followers among others that we are interested in analyzing for **RQ3**. Each metric was mined or calculated from GHTorrent, was measured relative to each repository, and can be identified via its suffix:

- _repo: Metrics measured amongst users within the same repository. For example, *issue_repo* is the number of issues reported in the repository by users who have *write access* to that repository.
- _family: Metrics measured amongst users in the same family. For example, *issue_family* is the number of reported issues by users exclusively with write access to another repository in the same family.
- _outside: Metrics measured from users not in the family. For example, *issue_outside* of reported issues by users with no write access to any repository in the same family.

3 RESULTS AND DISCUSSION

The classification of forks as social or hard within a family is the first step towards answering the research questions we posed in the beginning. Zhou et al. [20] propose a set of heuristics to differentiate between hard and social forks. These heuristics include the age of the repository, popularity (as measured by the number of stars in GitHub), number of merged or unmerged pull requests, whether the forks have received external pull requests, whether the forks have changed the name of the repositories, among others. Additionally, they interviewed several hard fork owners and found that many owners did not intend to create a hard fork and branch out the development, but after certain events or inconveniences (e.g., lack of responses from the upstream, disagreement between fork owner and upstream maintainers), these forks gradually evolved from social forks to hard forks and stopped interacting with the original projects.

In our work, we focus only on code contribution metrics, such as number of commits and number of merged commits by pull requests, to differentiate between hard and social forks, under the assumption that these represent a less subjective way to capture the nature of a fork and potentially the developers' intent. According

[2]https://gradle.org/
[3]https://maven.apache.org/
[4]This date was selected to remove bias of very "young" repositories of less than a year of age

Table 1: List of metrics mined from the repositories and considered in this study.

Category	Metric	Description
Age	age	Total age (in days).
Forks	depth	Total #forks away from the parent.
	forks	Total #forks.
	forks_family	Total #forks that have participated in a PR or issue within the family.
Commits	commit_count	Total #commits present in the repository.
	unique_commits	Total # of unique commits, present in the fork, but not in the parent
	commit_author	Username that made the commit
	commit_message	Message of the commit
Dependencies	dependencies	List of unique dependencies.
	jaccard	Jaccard distance between the sets of dependencies of the fork and of the parent
Users	users_repo	Total #users with write access to the repository.
	users_also_in_family	Total #users with write access to the repository and one other familial repository.
Followers	followers_repo	Total #followers of users_repo coming from the same repository.
	followers_family	Total #followers of users_repo coming exclusively from the family.
	followers_outside	Total #followers of users_repo coming from outside the family.
Pull Requests (PRs)	pr_repo	Total #PRs within the same repository (through branching)
	pr_family	Total #PRs within the same family (through forking)
PR Comments	pr_repo_comments	Total #comments from pull request discussions in pr_repo.
	pr_family_comments	Total #comments from pull request discussions in pr_family.
	pr_repo_code_comments	Total #comments from pull request commit discussions in pr_repo.
	pr_family_code_comments	Total #comments from pull request commit discussions in pr_family.
PR Mentions	pr_mentioned_repo	Total #PR mentions of users in users_repo.
	pr_mentioned_family	Total #PR mentions of users exclusively in the family.
	pr_mentioned_outside	Total #PR mentions of users outside the family.
Issues	issue_repo	Total #issues reported by users in users_repo.
	issue_family	Total #issues reported by users exclusively in the family.
	issue_outside	Total #issues reported by users outside the family.
Issue Comments	issue_comments_repo	Total #issue comments from issue_repo.
	issue_comments_family	Total #issue comments from issue_family.
	issue_comments_outside	Total #issue comments from issue_outside
Issue Mentions	issue_mentioned_repo	Total #issue mentions of users in users_repo.
	issue_mentioned_family	Total #issue mentions of users exclusively in the family.
	issue_mentioned_outside	Total #issue mentions of users outside the family.
Issue Events	issue_closed	Total #issues closed.
	issue_subscribed	Total #issues subscribed.
	issue_unsubscribed	Total #issues unsubscribed.
	issue_reopened	Total #issues reopened.
	issue_assigned	Total #issues assigned.
	issue_referenced	Total #issues referenced.

to our definition, a social fork initiates changes destined for the parent repository. These changes are represented by commits in the fork that are then proposed to the parent through pull requests. However, as people have different expectations of hard forks (which form a new community) and social forks (which are still part of the original community), it is necessary to define a threshold on the contribution of forks back to the parent as the portion of fork commits included in pull requests to the parent, and better identify these two types of fork.

Zhou et al. [20] defined 30% as the threshold between social forks and hard forks, yet without rigorous justification of this decision. Therefore, in this work, we investigate different merge thresholds and examine the impact of choosing a fixed threshold between hard and social forks in these kind of studies. This first analysis aims at investigating the distribution of social and hard forks in our data set and also how this distribution changes when we consider a different threshold for the ratio of merged pull requests. The objective is not to propose a unique or evaluated method to classify forks. In order to answer our research questions, we need to make this classification and we present how the process of defining a classification (i.e., selecting the merge threshold) can affect the outcomes of a similar study.

In order to calculate the proportion of merged *unique* commits from a fork to its parent, we define a unique commit as a commit that has originated only from the fork. In practice, when a fork is created from a parent repository, all the commits from the parent are automatically copied in the fork's history as well. Therefore, we compared the commits, based on their SHA IDs[5] between a fork and its respective parent to identify the commits that are unique only in the fork. However, commits that have been merged from the fork to the parent through a pull request are present in both the fork and the parent, hence, along with purely unique commits, we also add merged commits in this count as long as they originate from the particular fork (as mentioned in the corresponding pull request metadata). Finally, the percentage of merged commits is calculated as the ratio of merged over total unique commits in the fork.

Based on this ratio, we considered a number of different thresholds to examine the impact of choosing a fixed threshold in the analyses between social and hard forks. More specifically, we identify a fork as social if at least 1, 25%, 50%, 75% or all of its unique commits are merged back to its parent. For the rest of this work, we answer the various research questions separately for each of these thresholds and we discuss the results.

Table 2 shows the distribution of hard and social forks in our dataset for the different thresholds, with the proportion of hard forks increasing with threshold. In simple terms, as the threshold becomes stricter, from requiring just 1 merged commit to 100% merged commits, forks need to become much more coupled to the parent repository to be considered a social fork.

We also compared our classification with that by Zhou et al. [20]. If we consider the latter as the ground truth, the last three columns in Table 2 show the precision, recall, and accuracy measures per threshold. The best precision is achieved when we consider all commits being merged to the parent. In reality, from the forks contained in the examined dataset only 5% are identified as hard forks by Zhou et al. [20].

[5]Commits were also compared based on commit message and author name to account for rebasing issues, but no difference was found in the studied dataset.

Table 2: Fork types in our dataset by threshold compared to that in Zhou et al. [20]

	Our Dataset		Compared to [20]		
Commits	#Hard	#Social	Precision	Recall	Accuracy
1	919	1,648	44.37	7.31	64.78
25%	1,100	1,467	56.95	7.85	58.15
50%	1,256	1,311	73.51	7.32	41.76
75%	1,520	1,047	66.22	7.99	52.26
ALL	1,634	933	78.14	7.24	37.32

3.1 RQ1: Fork Evolution Analysis

Motivation: By definition, forks constitute an independent copy of the parent repository in the sense that changes happening in the forks are not automatically reflected back to the parent. If this is the case for the entire lifetime of the fork, then we can talk about hard forks. When changes are attempted to be explicitly pushed to the parent (through a pull request) in a more or less systematic manner (cf. the threshold in the previous section), we can talk about social forks. It is important to note that for our study the intention of the fork developer is the important part and not necessarily whether the contribution will be accepted or rejected.

Based on this, it is natural to expect that the two types of forks may demonstrate differences when it comes to their lifespan as well as their development activity. One intuitive hypothesis would be that, in general, social forks may have a shorter lifespan with a denser development activity; this is the case when a social fork contributes a small "patch" back to the parent, whose development is done over a short but intensive period with lots of commits. However, other social forks may have a longer lifespan, remaining active for a longer time, and contributing changes in spurts, i.e., contributing multiple patches, back to the parent. On the other hand, hard forks may correspond either to projects that have started to evolve independently and follow their own development progress, or repositories that were forked (perhaps for social reasons), but were slowly abandoned. Our intention is to recover such patterns, study them in the context of our entire dataset and within a per-family context and answer the question if there are significant differences in development activity between hard and social forks.

An interesting factor in the activity of forks within the context of the same family is the "depth of the family tree". Brisson et al. [2] discuss how forking is not a sequential process, but it can result in complex tree-like structures, which implies that repositories within the same family may be forked not directly from the parent of the family, but from other forks. Therefore, one question that remains to be investigated is whether the depth of a fork in the family tree has an effect on its development activity and practices.

To respond to **RQ1**, we examined the following two null hypotheses:

- $RQ1.H1_0$**:** *Activity is similar across social and hard forks.*

- $RQ1.H2_0$: *Activity does not vary along the depth of the family tree.*

Approach: The important metrics to test these hypotheses are mainly the number of commits and the age of a repository. In order to control project activity for project age, we calculated the metric of *"activity density"*, which simply shows the average number of commits per day of activity. In practice, this is a normalization of activity and it helps to differentiate two repositories with, for example, heavy activity (large number of commits), but a short and a long lifetime each. The density was used as a continuous outcome to study the relationship with other metrics of the repositories.

To validate our hypotheses statistically, we first checked for normality of the activity metric using the Shapiro-Wilk test ($\alpha = 0.05$), whose null hypothesis states that the data distribution under study is normal. Since this hypothesis was rejected with $p < 2.2e-16$, we can conclude that the activity density metric values were not normally distributed, hence we opted for non-parametric tests, i.e., Mann-Whitney for $RQ1.H1_0$ and Kruskal-Wallis for $RQ1.H2_0$ (comparing activity across 5 depth levels). In both cases, we used $\alpha = 0.05$. For the Kruskal-Wallis test, which is an omnibus test, we used the Dunn post-hoc tests in case $RQ1.H2_0$ is rejected, since this would allow to find the individual depth levels with significantly more or less activity.

Results: **There is a significant difference in activity between hard and social forks.** Testing our first hypothesis with the Mann-Whitney test between the activity density and the type of the fork (hard or social) rejected the null hypothesis ($p < 0.05$) for all 5 thresholds, when examining the entire dataset, i.e., all forks merged in a single dataset without accounting for individual families.

Activity varies significantly along with depth of the family tree. We obtained this result for our second hypothesis through a Kruskal-Wallis test on activity density for different depths of a fork in the repository family tree. We could not reject the null hypothesis (with $p = 0.1661$). One important reason for this could be the substantial imbalance of the dataset with respect to the depth of each fork. More specifically, out of the 3405 forks studied in total, 2964 were immediate forks of the root of the family tree, 323 were forks of forks and the other 72 were deeper in family tree up to a depth of 5. For this reason, we repeated the analysis to compare the activity between forks of depth 1 and forks of depth more than 1. For this analysis, we performed a Mann–Whitney U test. In this case, with $p = 0.028$, we can reject the null hypothesis, thus activity differs between forks of depth 1 and those deeper in the family tree.

The proportion of families with significant differences in activity between social and hard forks varies from 4.1% to 13.7%. Continuing on the per-family analyses, we performed a series of Mann-Whitney U tests to study if there are indeed any emerging patterns with respect to activity density and if these patterns are different between social and hard forks within the same family. Table 3 shows the numbers of families where different patterns of activity ("Activity" columns) were observed between social and hard forks. We performed the test for all considered thresholds of merged commits between forks and baselines. In some families, depending on the threshold, only one type of fork (either social or hard) occurs, as shown by the number of families with a single type of fork in the last column of Table 3. Examining the results more carefully, we found that 14 out of 75 families have a significant difference in activity between hard and social forks in at least one of the 5 thresholds. However, these families account for about 49% of our entire dataset. As a result, they seem to be responsible for the global outcome when all forks are merged into a single dataset. On the other hand, 50.7% (threshold "1 commit") to 85.3% (threshold "ALL") of families do not exhibit significant differences in activity. However, these families have an average of 28 forks per family accounting for 51% of the total number of repositories in our dataset. While it is evident that the size of the family has an impact on the results, we can argue that in larger families with more forking activity, the development intensity seems to differ between hard and social forks consistently across different merge thresholds to distinguish between the two types.

	Activity		Dependencies		
	$p < 0.05$	$p > 0.05$	$p < 0.05$	$p > 0.05$	Single type families
1 com.	10	63	8	65	2
25%	3	70	6	67	2
50%	4	67	6	65	4
75%	8	61	5	64	6
ALL	6	62	5	63	7

Table 3: Differences in Activity (based on activity density) and Dependencies (based on Jaccard distance) per family between social and hard forks for different pull request thresholds.

3.2 RQ2: Dependency Analysis

Motivation: The premise behind analyzing dependencies to study differences between forks and parents is that dependencies, along with code and documentation, can be used to accurately describe the purpose and the functionality of a project. Even in the presence of general-purpose dependencies, like logging, authentication, a certain number of dependencies are project or domain-specific, clearly indicating functionality. In the context of project families, we can easily deduce some drift in functionality and purpose between forks and parents by simply comparing the sets of dependencies. Unlike dependencies, code may require cumbersome and expensive comparisons to find differences, while documentation, when available, requires equally complicated natural language processing, with the associated shortcomings.

In this work, we focus on dependencies and more specifically on the presence or absence of dependencies between forks and parents, not version updates of existing dependencies. This is because dependency updates are a very common change, especially between forks and parents, but do not necessarily contribute to a drift in functionality. In addition, as mentioned before, build files, where dependencies are explicitly specified, usually leave version as a variable, exactly because it changes often and in most cases the latest version is the one required.

To respond to **RQ2**, we examined the following two null hypotheses:

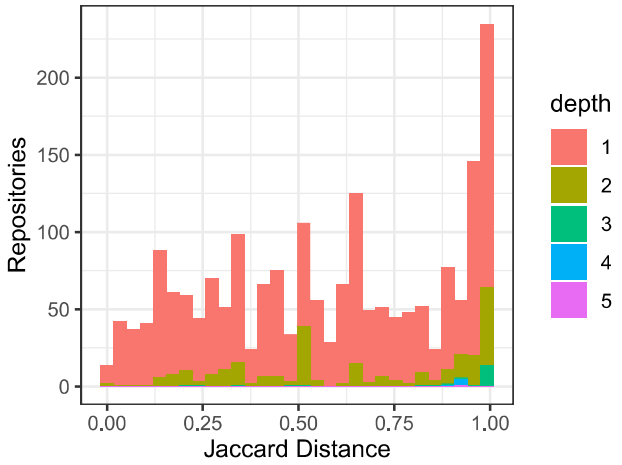

Figure 1: Frequency of repositories per Jaccard distance.

- $RQ2.H1_0$**:** *Dependency sets are similar across social and hard forks.*
- $RQ2.H2_0$**:** *Dependency sets do not vary along the depth of the family tree.*

Approach: To capture differences in dependencies, we considered the set of dependencies for every repository as declared in a build file (`pom.xml` or `build.gradle`). We excluded the version information from each dependency to avoid version updates to be considered as different dependencies (cf. motivation). We then calculated the difference in the dependency sets between the parent and each fork in terms of the Jaccard distance between the two sets.

To validate our hypotheses for RQ2, we followed a similar approach to that for RQ1. The Shapiro-Wilk test ($\alpha = 0.05$) showed that the Jaccard distance data is not normally distributed ($p < 2.2e-16$). Therefore, similar non-parametric tests were applied, Mann-Whitney for $RQ2.H1_0$ and Kruskal-Wallis for $RQ2.H2_0$ (comparing activity across 5 depth levels) with $\alpha = 0.05$. In case $RQ2.H2_0$ is rejected, we applied post-hoc Tukey tests [18] to see if the Jaccard distance is significantly different between different depths.

Results: **Deeper forks tend to have a larger probability of changing dependencies.** Only 1,361 repositories out of a total of 3,330 studied repositories (excluding parent repositories) had the exact same set of dependencies as their respective parent. Furthermore, Figure 1 shows the number of repositories with specific Jaccard distances. For every distance, the plot also includes the different proportions according to the depth of the repository. Depth 1 corresponds to forks of the parent, depth 2 corresponds to forks of forks and so on. We can observe that as the depth increases the probability for greater deviations from the set of dependencies of the parent also increases, i.e., deeper forks tend to have a larger probability of changing dependencies.

We studied in detail the relationship between the Jaccard distance and the depth in the family tree to identify the degree to which the deviation in dependencies is stronger as we go deeper. Our dataset contained forks up to a depth of 5 (where 0 is the depth of the parent repository). The results for the Kruskal-Wallis test showed that the Jaccard distance can become significantly higher as we go deeper in the family tree. In addition, the post-hoc Tukey tests show that the Jaccard distance is significantly different between level 1 and each deeper level (i.e., 1-2, 1-3, 1-4), but there is not much difference between other levels. From this, we can deduce that starting from depth 2 the set of dependencies has deviated enough from the parent that the Jaccard distances among all the deeper forks are significantly different.

Between 7.2% and 10.9% of families show a significantly higher deviation of dependencies in social forks compared to hard forks. To study if the type of fork plays a role in the deviation of dependencies, we again performed a series of Mann-Whitney U tests per family of repositories to see if the Jaccard distances per type of fork come from different distributions. As shown in Table 3 (column Dependencies), this hypothesis was confirmed only for a few families for every threshold of commits. Nevertheless, when we considered all studied repositories, outside the context of families, the corresponding Mann-Whitney U tests for every threshold confirmed the hypothesis that the Jaccard distances produce different distributions for social and for hard forks, respectively. We also found that 9 out of 75 families had significantly different dependency sets between hard and social forks, according to Jaccard distance. These families accounted for about one third (34.8%) of the dataset with an average of 131 forks between them. On the other hand, at least 89% of families did not show significant difference in their dependency sets between hard and social forks. These families accounted for about 65% of our dataset, but with an average of 33 forks per repository. Once again, the difference in dependency sets seems to be more prominent for larger families with more forking activity.

3.3 RQ3: Communication and Evolution Analysis

Motivation: As fundamentally collaborative activities, software projects have a pronounced social aspect that involves significant amounts of direct or indirect communication among developers. Based on the definitions for hard and social forks, we hypothesize that the quantity and the means of communication would be different for the two types of forks. Moreover, we explore if communication differs between the different evolution patterns, as expressed by activity density and differences in dependencies between forks and parent projects.

To respond to **RQ3**, we examined the following three null hypotheses:

- $RQ3.H1_0$*: Communication is the same between hard and social forks.*
- $RQ3.H2_0$*: Communication is the same between different levels of activity density.*
- $RQ3.H3_0$*: Communication is the same between different levels of Jaccard distance, representing the difference between forks and parents with respect to dependency sets.*

Approach: Since communication cannot be encapsulated in a single value and rather it consists of multiple parameters, the hypotheses were not checked with simple statistical tests that would accept or reject the null hypothesis. Rather, we trained different

Table 4: Significant communication metrics across all thresholds.

Metric	ALL	75%	50%	25%	1 Commit
(Intercept)	-0.63***	-3.71***	-3.82***	-4.086***	2.52***
forks	0.052**	-0.005**	0**	-0.004*	0.05
users_repo	0.55***	0.41***	0.42***	0.32***	0.86***
users_also_in_family	-0.61***	-0.39***	-0.38***	-0.35***	-0.85***
followers_repo	0	0.050	0.020	0.053	-0.04
followers_family	0*	-0.01*	-0.03***	-0.06***	-0.01***
followers_outside	0	0	0	0	0*
pr_repo	0	0.01	0.009	0.01	0.02
pr_family	0	0	0	0	-0.01***
pr_repo_comments	0	0	0	0.01**	-0.01
pr_family_comments	0*	0*	0*	-0.00	0**
pr_repo_code_comments	0.01	0	0.01	-0.02	0.28
pr_repo_code_comments	0.02***	0*	0.01***	0	0.01
issue_repo	0.10	-0.05	-0.06	-0.05	-0.49*
issue_family	1.71	0.55**	0.55**	0.91***	12.39
issue_outside	-0.09**	-0.01*	-0.01*	-0.01*	-0.01
issue_assigned	0.11	-0.02	-0.03	-0.21	-4.99*
R^2	0.084	0.132	0.149	0.154	0.155

regression models with the communication metrics as the predictors, and the respective parameter for each of the above hypothesis: logistic (hard vs social) for $H1_0$, activity density for $H2_0$, and Jaccard distance for $H3_0$. The models will tell us how accurately the communication metrics can predict, or in our case explain, the outcomes. If there exist statistically significant ($p < 0.05$) predictors and the regression model is well-fitted (according to R^2), we assume that there is significant difference in communication between the different outcomes. The logistic regression model was trained for all five thresholds studied for hard and social forks.

Results: **Evidence was found that certain communication elements (users, followers, issues) are different between social and hard forks.** Table 4 shows the results for the logistic regression models with the type of fork as outcome and the communication metrics as input. The significance for each communication metric is shown with respect to its p-value: $*$ => $p \leq 0.05$, $**$ => $p \leq 0.01$, $***$ => $p \leq 0.001$.

As it can be seen by R^2 for each threshold, communication metrics cannot predict the type of fork with high accuracy. However, we can see that the model fitness progressively increases as we relax the threshold. While this may be due to data availability or outcome imbalance, as it was discussed in Table 2 the balance with respect to the outcome changes uniformly between the threshold. As a result we can understand that the prediction is better when we have more social forks. This conclusion is intuitive as social forks tend to be more active in terms of communication, especially within the same family, as we will discuss next. However, a more microscopic analysis is needed to further confirm this finding.

With respect to significant communication predictors, we can also see certain patterns. For example, it can be seen that the number of users and the number of common users within the family are both significant predictors across all thresholds. However, the respective trends between the two metrics are opposite. The more users we have in a fork, the higher the log-odds that the fork is a hard fork, while the more users within the family, the higher the log-odds for a social fork. Again, this is an intuitive finding, given that in social forks users tend to be active in at least two forks, the social fork and the parent. A similar observation can be made for followers within the family, where in social forks users tend to follow multiple repositories within the same family.

An interesting pattern can be observed with respect to issues. While the number of issues within the family are an important predictor for the type of fork, but conversely to users and followers, a higher number of issues within the family implies higher log-odds for a hard fork. In the contrary, more issues by users outside to family imply higher log-odds for a social fork. We speculate that this finding may be circumstantial based on how the repositories are used. In any case, both metrics are neither very significant nor consistently significant across all thresholds. Finally, there seems to be no or little connection between pull requests and the type of fork.

Table 5: Coefficients of communication metrics for Activity Density and Jaccard Distance in linear regression models.

Communication metrics	Activity Density	Jaccard
(Intercept)	1.83E-02*	3.25E-01***
forks	1.05E-04	4.66E-04
user_repo	-6.59E-03	2.34E-02***
user_also_in_family	2.37E-02***	-2.43E-02***
followers_repo	1.55E-02***	9.91E-03**
followers_family	-6.39E-05	-1.03E-03**
followers_outside	-1.69E-06	6.10E-06***
pr_repo	4.09E-03***	-7.78E-04
pr_family	2.19E-03***	-2.72E-04
pr_repo_comments	-1.33E-04	-8.39E-04
pr_family_comments	3.22E-04***	-7.43E-06
pr_repo_code_comments	9.90E-04	-2.61E-03
pr_family_code_comments	-4.98E-04*	-1.25E-03***
issue_repo	1.22E-02*	6.43E-03
issue_family	-2.40E-02	-4.18E-02
issue_outside	-2.17E-02	7.73E-03
issues_assigned_repo	2.71E-03	0.24048
R^2	0.4731	0.02446

High communication metrics are correlated with high development activity in terms of commits. Table 5 shows the results of the linear regression models with the activity and Jaccard distance as the outputs. One first observation is that the model between communication and activity had $R^2 = 0.4731$ showing a good fit and a potential relationship between activity density and communication. More specifically, we can see that repositories with a high number of users within the family of the fork, followers, pull requests of the fork and of the family and issues also have high activity with respect to daily commits. Unlike the type of fork, pull requests and some of their respective social metrics are found to be good predictors of activity density.

Communication metrics are not highly correlated with differences in the dependency sets between parent repositories and forks. The fitness of the linear regression model with the Jaccard distance as the output was $R^2 = 0.02446$, which may imply that there is no strong relationship between communication and the purpose of the repository as manifested by its dependencies.

Even so, we can see that high values for repository specific communication metrics, like number of users and number of followers, imply high Jaccard distance and consequently greater deviation of the fork's dependencies from the parent. Conversely, high values of family related communication metrics, like users, followers and commit comments in pull requests, imply lower Jaccard distance and more similar dependency sets between the fork and the parent. Given that, as we have shown in **RQ2**, low Jaccard distance is correlated with social forks, it makes sense that higher communication activity within the broader family of the fork implies social forks.

Overall, we can confirm that communication metrics can be used to indicate differences between the type of the fork (hard vs social) or between different levels of activity density (in terms of daily commits). Therefore, we can reject the null hypotheses $RQ3.H1_0$ and $RQ3.H2_0$. However, our analysis could not identify a strong correlation between communication metrics and the difference in the dependency sets (based on the Jaccard distance). Therefore, we could not reject the null hypothesis $RQ3.H3_0$.

4 RELATED WORK

4.1 Forks and Pull Requests

Forks and the practice of forking have been of interest to software engineering researchers for over 20 years [20]. Several studies have attempted to categorize forks by type. Zhou et al. [20] differentiate between "hard" and "social" forks. Social forks are frequently used to enable contributions from developers who are external to a project. Independent or "hard" forks [20] often result in competing development activities and significantly different project directions.

Rastogi and Nagappan [11], identify three types of GitHub forks: contributing, independent, and inactive. Contributing forks are similar to social forks in [20] in that are used to integrate changes into the forked project (baseline) via pull requests (PRs) whereas, independent forks do not issue pull requests and have internal commits that differ from those in the baseline project [11] (akin to hard forks in [20]). Inactive forks do not issue pull requests and do not have any commits. Contributing forks are further classified into those which are mostly used to fix bugs and those which are used to add new features or functionality, and those that are used to change configurations [8, 14].

Jiang et al. [8] investigated why and how developers fork what from whom in GitHub. They observed that a common reason developers fork a repository is to send pull requests, fix bugs, and add new functionality [10, 14].

Robles and González-Barahona [13] found that a minority of forks are merged back into the baseline repository. Moreover, the number of forks that integrate code from similar or parent projects is extremely low.

A study on the topic of clone-based variability management demonstrated that practices such as clone-and-own is broadly used in the Android ecosystem [3]. The study explored Android apps that can be accessed through the official app store as well as Google Play. A total of 88 Android application families were analyzed. They found that: 1) the propagation of code from one variant to another one is not more common in the applications, and 2) the number of parent forks that propagate code through pull requests is very low.

4.2 Communication

There is substantial research on the significance of communication in GitHub. Tsay et al. [17] explore metrics that relate to PR acceptance, including the number of comments, and the social connections between the contributor and project manager. Zhang et al. [19] explore the effects of `@mentions` on processing PRs, including reducing the delay of developer collaboration. Bissyandé et al. [1] conduct a large scale study on GitHub issues, including how they relate to repository success. Destefanis et al. [5] look specifically at issue comments, and explore how sentiment differs between users and project contributors. Dabbish et al. [4] make the case that GitHub contains a rich set of social inferences.

The notion of a software family is introduced in [2] in order to analyze developer interaction within repositories, among repositories within the same family, and among families. The study found that interactions from developers in the same software family share a relationship with repository stars. Their results suggest that a software family is an interesting concept for investigating developer contributions.

5 CONCLUSION

The work presented in this paper aimed at studying the current practices and trends of forking in open-source software repositories. The study explored how forks evolve with respect to their parent repository and how this forked evolution may manifest itself in differences between forks of the same family of repositories with respect to the purpose of the fork, the development activity, the functionality of the software and the communication intensity between developers. Our results showed that within repository families we can differentiate between hard and social forks and that these two types of forks show different patterns with respect to the development activity, the evolution of dependencies and the social interactions between the developers. We have also shown that differences in development practices are also correlated with communication activity.

The main objective of the study was to show that we can easily characterize a fork, understand its purpose, and potentially predict its evolution with respect to its parent, by observing measurable and pragmatic indexes such as the number of commits, the number users, the size and depth of the repository family, among others. The practical usefulness of this finding is the increase in developer awareness of software projects and the increased ability to onboard new members in development activities. By focusing on a small set of specific indexes and metrics and having a general awareness of the family of a fork, a developer, new or old, can understand a lot about the evolution of the project.

Although our study was extensive in terms of the number of repositories and families of repositories it considered, at this stage its nature was mostly observational. In the future, we plan to focus deeper on specific families and repositories to identify case studies that can provide explanations and potentially more specific patterns that would further justify our findings. Personally contacting developers of the studied repositories, as in the case of Zhou et al. [20], would further confirm the value and practicality of our findings.

REFERENCES

[1] Tegawendé F Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. 2013. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 188–197.

[2] Scott Brisson, Ehsan Noei, and Kelly Lyons. 2020. We Are Family: Analyzing Communication in GitHub Software Repositories and Their Forks. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 59–69.

[3] John Businge, Moses Openja, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-based variability management in the android ecosystem. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 625–634.

[4] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. ACM, New York, NY, USA, 1277–1286.

[5] Giuseppe Destefanis, Marco Ortu, David Bowes, Michele Marchesi, and Roberto Tonelli. 2018. On measuring affects of github issues' commenters. In *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*. 14–19.

[6] Daniel M. German, Bram Adams, and Ahmed E. Hassan. 2015. Continuously Mining the Use of Distributed Version Control Systems: An empirical study of how Linux uses git. *Empirical Software Engineering* 21, 1 (2015), 260–299.

[7] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 12–21.

[8] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22, 1 (2017), 547–578.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[10] Linus Nyman and Tommi Mikkonen. 2011. To fork or not to fork: Fork motivations in SourceForge projects. *International Journal of Open Source Software and Processes (IJOSSP)* 3, 3 (2011), 1–9.

[11] Ayushi Rastogi and Nachiappan Nagappan. 2016. Forking and the Sustainability of the Developer Community Participation–An Empirical Investigation on Outcomes and Reasons. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 102–111.

[12] Eric S. Raymond. 1998. Homesteading the Noosphere. *First Monday* 3, 10 (Oct. 1998). https://doi.org/10.5210/fm.v3i10.621

[13] Gregorio Robles and Jesús M González-Barahona. 2012. A comprehensive study of software forks: Dates, reasons and outcomes. In *IFIP International Conference on Open Source Systems*. Springer, 1–14.

[14] Stefan Stanciulescu, Sandro Schulze, and Andrzej Wasowski. 2015. Forked and integrated variants in an open-source firmware project. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160.

[15] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* 43, 1 (2017), 1–18.

[16] TIOBE. 2020. *TIOBE Index for August 2020.* https://www.tiobe.com/tiobe-index/

[17] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 356–366.

[18] John W Tukey. 1949. Comparing individual means in the analysis of variance. *Biometrics* (1949), 99–114.

[19] Y. Zhang, G. Yin, Y. Yu, and H. Wang. 2014. A Exploratory Study of @-Mention in GitHub's Pull-Requests. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. 343–350.

[20] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.