Clone-Based Variability Management in the Android Ecosystem

John Businge,* Moses Openja,* Sarah Nadi,† Engineer Bainomugisha,‡ and Thorsten Berger§
*Mbarara University of Science and Technology, Mbarara, Uganda
†Makerere University, Kampala, Uganda
‡University of Alberta, Edmonton, Canada
§Chalmers | University of Gothenburg, Gothenburg, Sweden

***Abstract*—Mobile app developers often need to create variants to account for different customer segments, payment models or functionalities. A common strategy is to clone (or fork) an existing app and then adapt it to new requirements. This form of reuse has been enhanced with the advent of social-coding platforms such as Github, cultivating a more systematic reuse. Different facilities, such as forks, pull requests, and cross-project traceability support clone-based development. Unfortunately, even though, many apps are known to be maintained in many variants, little is known about how practitioners manage variants of mobile apps.**

We present a study that explores clone-based reuse practices for open-source Android apps. We identified and analyzed families of apps that are maintained together and that exist both on the official app store (Google Play) as well as on Github, allowing us to analyze reuse practices in depth. We mined both repositories to identify app families and to study their characteristics, including their variabilities as well as code-propagation practices and maintainer relationships. We found that, indeed, app families exist and that forked app variants fall into the following categories: (i) re-branding and simple customizations, (ii) feature extension, (iii) supporting of the mainline app, and (iv) implementation of different, but related features. Other notable characteristic of the app families we discovered include: (i) 73 % of the app families did not perform any form of code propagation, and (ii) 74 % of the app families we studied do not have common maintainers.

***Index Terms*—software variants, mobile apps, app families, Android, software ecosystems**

I. Introduction

Software reuse is essential to keep up with the pervasiveness of software in our everyday lives. The advent of social coding platforms and version-control systems such as Github and Bitbucket has made large-scale software reuse more systematic by providing different facilities, such as pull requests and cross-project traceability, to allow fork-based development [1]. In the latter, developers realize new features or system variants by forking a repository, making changes on their own fork, and propagating changes back to the repository from which they forked, via pull requests (known as *"upstream" propagation*).

Mobile apps often need to exist in different variants [2], [3], accounting for different users and markets, or non-functional requirements, such as hardware, power consumption, performance or fidelity. As such, these variants typically share common and variable features [4], [5], and need to be maintained in parallel. Unfortunately, despite relatively simple configuration mechanisms, the Android platform does not offer more sophisticated variability management concepts, including those from methodologies such as software product line engineering [3], [6], which advocates integrating all variants into a platform. Instead, when such concepts are not available or applicable, a common form of variability management, known as clone&own [7], is to copy and adapt existing variants, and propagate changes (e.g., new features or bug fixes) to maintain and evolve the variants. In fact, as studies show, there is substantial software reuse in the Android ecosystem [2], [8], [9] through cloning. These observations suggest that, when variants of apps exist and need to be maintained, that the majority is done using clone&own. Yet, the practices applied by mobile-app developers are unknown—hindering the improvement of such practices.

We address this gap with an exploratory study on variant management practices in one of the largest app ecosystems in existence today: Android apps. We focus on apps that are available in the official app store, Google Play, and that host their source code on Github. This allows identifying app families as well as studying the variability management practices in depth. We consider an *app family* as a collection of apps on Github that are maintained together, typically consisting of a *mainline variant* (MLV) and its forked variants (FVs), representing existing variants of the mainline app.

Our study is guided by the following research questions:

RQ1 *What are the characteristics of Android app families?*
We investigate general characteristics of open-source apps that belong to an app family, including the app category they belong to and the pace of development and maintenance of the FVs with respect to the MLV.

RQ2 *How are app families maintained and co-evolved?*
We strive to understand how code is propagated between variants of the same family. For example, are pull requests used as the main propagation technique? Is code only propagated between the FVs and the MLV or are there additional propagations between other FVs?

RQ3 *How diverse are the contributors in Android app families?*
We investigate whether the FVs are typically created and controlled by new developers or whether they are still governed by the MLV developers.

RQ4 *What are the various types of variations or customizations that lead to the creation of an app family?*
Understanding the reasons behind creating a fork of

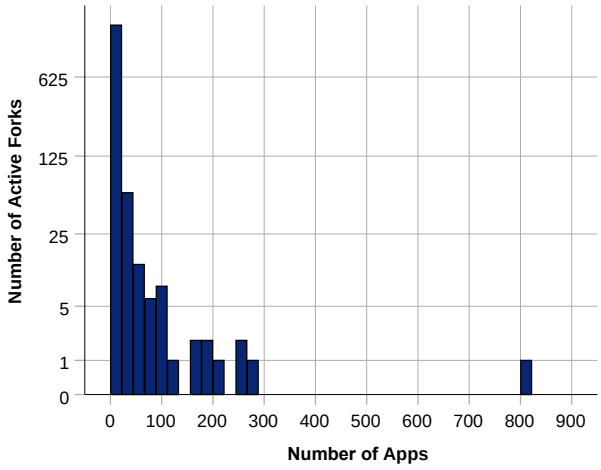

Fig. 1: Number of active forks across the considered 1,890 apps (y-axis in log scale)

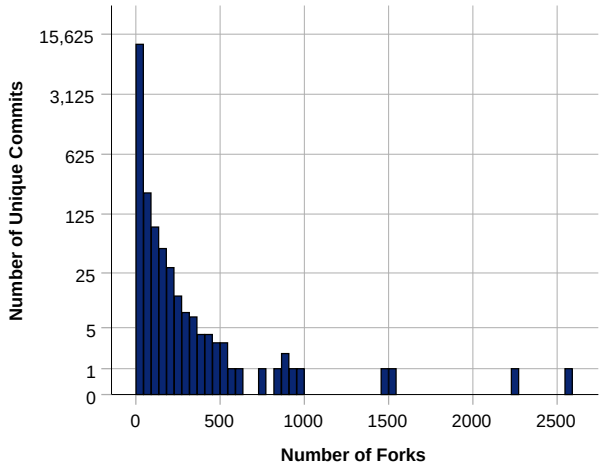

Fig. 2: Number of unique commits (compared to the mainline app) in the 12,356 active forks (y-axis in log scale)

another app can help us understand the types of reuse that occur in the Android ecosystem. For example, does an FV app typically add small functionality to the MLV app or does the FV use the MLV as a building block for an app with a completely different focus?

In our study, we found that app families actually exist, even though, there is little code propagation using typical Github facilities. When there is, then mainly from the MLV to the FVs. To the best of our knowledge, our work is the first to explore variability management practices and app-family characteristics on a social-coding platform such as Github.

We proceed by presenting our study design—especially how we identified app families—in Section II. We present our results for the four research questions in Sections III to VI, first providing the research-question-specific methodology, then the respective results. We discuss threats to validity in Section VII, related work in Section VIII, and conclude in Section IX.

II. Study Design

Our aim is to identify Android app families in order to study their characteristics and development practices.

A. Identification of App Families

Recall that we defined an Android app family as a collection of apps on Github that are maintained together, typically consisting of a MLV and its FVs. To identify all app families that exist on Github and that are also published on Google Play, we first mined the MLVs and their corresponding FVs on Github, then we found the corresponding links of these MLVs and FVs on Google Play. We identified app families as follows.

1) Using Github's Rest API v3 we identified a total of 55,939 repositories with the following criteria: (1) contains the word "Android" in repository name, description or readme.md file; (2) is not a fork; (3) is written in a programming language; (4) has been forked at least twice; and (5) was created no later than 31-12-2017. We used the criteria of having at least two forks to reduce the chance of finding student assignments, which could pollute our results [10].
2) To assure that the repositories we identified are indeed real Android apps, we searched for an `AndroidManifest.xml` file in each repository and, if it existed, we identified the package name indicated in the file. We then looked up the app with the extracted package name from Google Play. This step filtered out more repositories from our list, since some repositories may have no manifest file or might have no corresponding app on Google Play. We obtained 5,865 repositories representing an actual Google Play app.
3) Next, we manually looked at the list of the 5,865 apps to identify any apps sharing the package name, which would mean they are linked to the same app on Google Play. We removed 330 apps that have duplicate package names, leaving 5,535 apps. We speculate that duplicate package names are a result of repositories cloning other apps' source code and including it in their own.
4) From the remaining 5,535, we eliminated apps with less than six commits in their lifetime, according to the median number of commits in Github projects found by prior work [11]. After this preprocessing step, we were left with 4,634 apps.
5) We then eliminated apps without active forks—that is, forks that did not have one single commit after the fork creation date. We were left with 2,423 apps, which have 18,446 active forks altogether. For each fork, we identified commits that are unique to the fork versus those that were pulled from the mainline app after the fork date. We will explain how we identified the unique and pulled commits in Section IV-A. We eliminated apps that did not have at least one fork with a unique commit. This left us with 1,890 apps. Their distribution is shown in Fig. 1. Figure 2 shows the distribution of active forks and their unique commits. The forks that did not have unique commits meant that they only performed cherry-picking and did

TABLE I: Collected metrics that characterize Android app families and variant-management practices, arranged by research question (some metrics used in multiple questions)

Dimension	Metric	Description
Family Size (RQ1)	*Variants*	Number of variants in an app family
Variant Duration (RQ1)	*Duration*	Number of weeks since the earliest commit of any variant (since the first fork date) to latest commit of any variant (duration for an app family).
	ForkVariantBacklog	Number of weeks a given FV is behind its MLV (MLV's last commit date minus the FV's last commit date).
	Inactivity	Number of weeks a given variant has spent without making a commit until the stopping date (31-12-2017). We consider the median statistics to describe all the variants (inactivity for a family).
Code Propagation (RQ2)	$PullRequest_{MLV\text{-}FV}$	Number of closed pull requests from the MLV variant to a given FV in an app family.
	$PullRequest_{FV\text{-}MLV}$	Number of merged pull requests from a given FV to the MLV in an app family.
	$PullRequest_{FV\text{-}FV}$	Number of merged pull requests from one FV to another FV in an app family.
	StartingCommits	Number of common commits between a given FV and the MLV. Count of the MLV commits from the first commit until the given fork date.
	$StartingCommits_{cdLOC}$	Number of changed lines of code of the commits in *StartingCommits*. We use this metrics to calculate the $VariabilityPercentage_{cdLOC}$ discussed later in this table.
	$DirectPullCom_{MLV\text{-}FV}$	Number of common commits between a given FV and the MLV after the fork date that are not $PullRequest_{FV\text{-}MLV}$, $PullRequest_{FV\text{-}MLV}$ or $PullRequest_{FV\text{-}FV}$ (c.f. Section IV-A for detailed explanation).
	$PullRequestsCom_{MLV\text{-}FV}$	Number of commits associated with merged pull requests from the MLV to a given fork (i.e., associated with $PullRequest_{FV\text{-}MLV}$)
	$PullRequestsCom_{FV\text{-}MLV}$	Number of commits associated with merged pull requests from the MLV to the FV (i.e., associated with $PullRequest_{FV\text{-}MLV}$)
	UniqueCom	For a given MLV–FV pair, these are the number of commits that are unique to a each variant.
	$Unique_{cdLOC}$	Changed lines of code of the commits in *UniqueCom*.
	MergedCom	Total number of common commits between the MLV and FV pair after the fork date (i.e., common commits between MLV and FVs excluding *StartingCommits*).
	MergedCdLOC	Number of changed lines of code of the commits in *MergedCom*.
	VariabilityPercentage	Percentage of unique commits in a given FV with respect to the total number of commits in the FV (i.e., *UniqueCom* / (*UniqueCom* +*StartingCommits* + *MergedCom*)*100%. *VariabilityPercentage* is unique for all FVs.
	$VariabilityPercentage_{cdLOC}$	Percentage of changed lines of code of the *UniqueCom* in a given FV with respect to the total number of changed lines of code of the unique commits in the FV (i.e., $Unique_{cdLOC}$ / ($Unique_{cdLOC}$ + $StartingCommits_{cdLOC}$ + *MergedCdLOC*)*100%
Code Authorship (RQ3)	*TotalDevMLV*	Number of developers who contributed to a given MLV
	TotalDevFV	Number of developers who contributed to a given FV
	TotalDevs	*TotalDevMLV* + *TotalDevFV*.
	CommonDevs	Number of common developers between a given pair of MLV and FV in a family.
	CommonDevs %	Percentage of common developers between a given pair of MLV and FV with respect to the *TotalDevs*.
Google Play metadata (RQ4)	*Category*	Google Play app category of a given variant
	Description	Google Play app description of a given variant

not make any modifications in the forked code.

6) One important criteria for an app family is that its variants actually represent different apps. To consider only repositories that have at least one fork with a different package name and are on Google Play, we searched for the `AndroidManifest.xml` in each fork repo like we did for the mainline in Step-2 above. We then searched for the extracted package names of the forks on Google Play only if they were different from the mainline package name. A fork having the same package name as the mainline means that it did not make modifications to the mainline package name, implying that it is not yet been advertised on Google Play. After collecting the forks that are advertised on Google Play, we also performed a manual step to eliminate false positives i.e., two forks advertising the same package names. This would mean that one of the two forks, or both, have copied code that contains a manifest file with a package name advertised on Google Play. We would then look up the descriptions of both forks on both Github and Google Play, and also inspect other information, including the Github developer name and the developer name on Google Play. In some cases, the Google Play app description would have a link to the Github repository. Based on this process, we identified 88 apps that have at least one fork with a distinct associated app on Google Play. The 88 apps have a total of 127 forks. Since the 88 mainline apps and their forks are maintained in parallel on Google Play, we consider them as app families. This set of 88 app families is the final data set we used to answer our RQs.

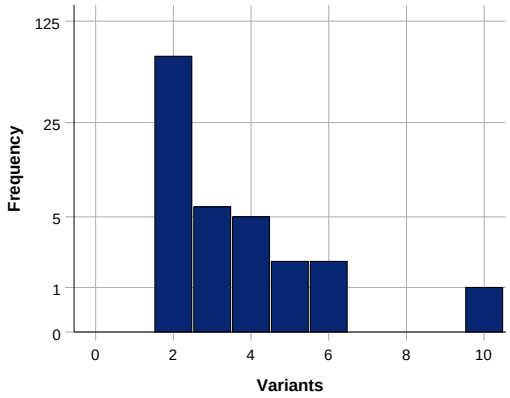

Fig. 3: Number of variants per family (y-axis in log scale)

B. Analysis of App Families

To answer the RQs we defined metrics that help answer each question. Table I provides an overview of all metrics. When reporting the results for each RQ in the remainder, we will explain the respective metrics in detail.

We collected the metrics both from Github and from Google Play using scripts we wrote. For the former, we utilized Github's REST API v3; for the latter, we directly mined from the Google Play website by creating scripts using the jsoup library, which helps parsing websites.

III. App Family Characteristics (RQ1)

With RQ1, we investigated general characteristics of apps that belong to an app family.

A. Methodology

To generally characterize the identified 88 app families, we looked into the sizes of the families (*Variants*) and metadata available on Google Play: the app category and the app description. We were also interested in understanding how up-to-date the FVs were with respect to their MLVs, for which we defined the metric *ForkVariantBacklog*. For each forked variant it determines the number of weeks it is behind its MLV in terms of dates of the respective last commit. A positive value means the FV is ahead of the MLV.

B. Results

The first step in our analysis consisted of examining various descriptive statistics of the Android families we identified. The 88 mainline apps of the considered families are written in 4 programming languages with the majority being written in Java (74), C (8), C++ (4), and Kotlin (2). The 88 mainline apps have a total of 26 Google Play categories. The categories with the highest number of mainline apps include: tools (19), productivity (11), communication (9), education (7), finance (7), and entertainment (5). The 127 fork variants of the 88 mainline apps had a total of 21 categories on Google Play that are not necessarily a subset of the 26 categories of the mainline apps. The highest frequency of fork variant categories include: tools (27), finance (18), productivity (15), communication (14), education (7), and health & fitness (6). From the statistics of the total number of categories presented (i.e., 26 for mainline variants and 21 for fork variants), we observe that variants in the same app family can be listed in different Google Play categories.

TABLE II: App family metrics (defined in Table I)

Metric	Mean	Min	Median	Max
Variants	2.4	2	2	10
Duration	505.6	0	158	2439
ForkVariantBacklog	45.1	-244	32.5	313
Inactivity	233.0	-1473	181	2191
$PullRequest_{MLV\text{-}FV}$	0.2	0	0	10
$PullRequest_{FV\text{-}MLV}$	0.2	0	0	4
$PullRequest_{FV\text{-}FV}$	0.0	0	0	1
TotalDevMLV	30.2	1	7.5	270
TotalDevFV	3.1	1	1	43
TotalDevs	42.2	1	21	272
CommonDevs	1.0	0	0	14
CommonDevs %	6.3	0	0	86
StartingCommits	1411.5	0	463	28924
$StartingCommits_{cdLOC}$	1.05M	0	201K	7117M
MergedCom	364.2	0	0	25758
MergedCdLOC	102K	0	0	4.8M
$PullRequestsCom_{MLV\text{-}FV}$	1.2	0	0	136
$PullRequestsCom_{FV\text{-}MLV}$	15.8	0	0	427
UniqueCom	85.7	1	13	2260
$Unique_{cdLOC}$	114K	20	16K	1.66M
VariabilityPercentage	13.65	0.04	2.76	100
$VariabilityPercentage_{cdLOC}$	20.58	0.00	9.21	100

Figure 3 shows the distribution of the number of variants each app family has. We can observe that the figure is right-skewed, meaning that the majority of the app families has two variants. In Fig. 4, we present the FV backlog with respect to the MLV. We observe that we have: (i) a few cases below the zero line of the y-axis, that is, the FV is ahead of the MLV, (ii) a few cases along the zero line, that is, there is no significant difference between updates of the FVs and MLV, and (iii) the majority of the cases above the zero line, that is, the updates of MLVs are ahead of the FV on Github.

IV. Code Propagation in App Families (RQ2)

With RQ2, we determined how often code is propagated between the variants in a family, using what common practices.

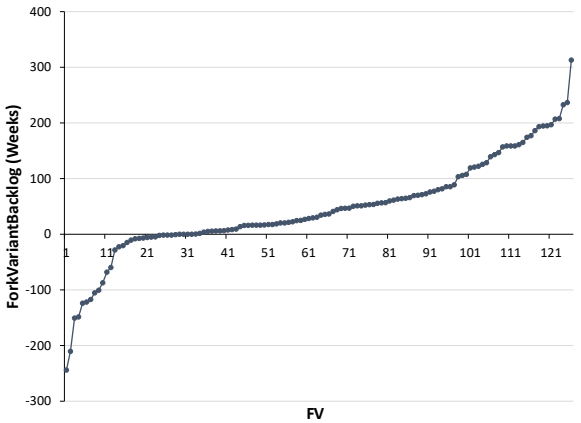

Fig. 4: FV backlog with respect to the MLV: number of weeks the FV is behind the MLV in terms of the last commit dates.

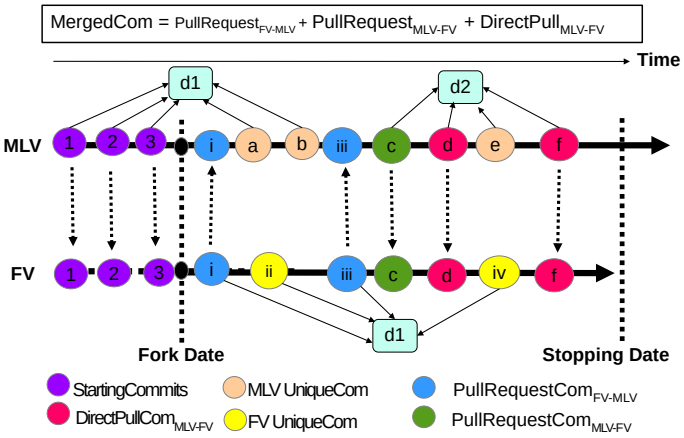

Fig. 5: Illustration of the different types of commits present in a FV and its corresponding MLV.

A. Methodology

In Table I, we outlined a number of metrics that relate to code propagation. We now discuss them in detail. In any variantin a family, there are two categories of commits: variant-specific commits (*UniqueCom*) and common commits. The latter can be further categorized as: (i) the starting commits between MLV and FV that exist at the moment of forking (*StartingCommits*) and (ii) the merged commits that appear since the fork date until the last commit before the stopping date (*MergedCom*).

We now explain how we extracted the metrics *UniqueCom*, *StartingCommits*, and *MergedCom* from GitHub. Figure 5 illustrates the mentioned kinds of commits (and others, explained shortly) for MLV and FV. Recall that all metrics are summarized in Table I.

- *UniqueCom*: In Fig. 5, the *UniqueCom* commits for the MLV are a, b, and e, while those for the FV are ii and iv. To extract these *UniqueCom* for MLV and FV, we collected and compared sets of commits of the MLV and the FV since the fork date to the last commit before the stopping date.
- *StartingCommits*: In Fig. 5, the *StartingCommits* are 1, 2, and 3. These existed in the MLV at the time of the fork creation, which means these are the commits the fork starts with. To extract these commits, we collected all the commits since the *first commit* on MLV until the *fork date*.
- *MergedCom*: In Fig. 5, MLV and FV have the same *MergedCom*: i, iii, c, d, and f. For a given FV and the corresponding MLV, we considered a commit as merged when it appears in both FV and MLV after the fork date.

Looking at *MergedCom* only is not enough, since we also want to understand the direction of the code propagation. On a social coding platform such as Github, there are two ways how code can be propagated: merged pull request (PR) commits or direct pull commits.

- *Merged PR commits*: In a family of app variants, pull requests can be sent from any variant by a *commit author* and received in another variant by a *commit merger*. We can establish three directions from the sender of the PR to its receiver: from the mainline to the fork (*PullRequest*$_{MLV\text{-}FV}$), from the fork to the mainline (*PullRequest*$_{FV\text{-}MLV}$), and from a fork to another fork (*PullRequest*$_{FV\text{-}FV}$). The former two directions are illustrated in Fig. 5. We can see in the figure that only one commit c happens through *PullRequest*$_{MLV\text{-}FV}$, while two commits i and iii happen through *PullRequest*$_{FV\text{-}MLV}$.
- *DirectPullCom*$_{MLV\text{-}FV}$: These are commits that were merged into a variant by being *pulled* from one variant and *pushed* into another variant. This can be achieved in two ways: syncing a fork[1] or merging an upstream repository into a fork.[2] In the latter case, only a subset of commits may actually be merged (cherry picking). Like the pull requests, direct commits may appear in two directions. However, unlike pull requests, it is not possible to identify which variant a direct pull commit came from. This is because of the nature of distributed version-control systems such as git: 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 repo as a result of the fork trying to keep in sync with the new changes in the mainline, we made an assumption that all the direct pull commits we find in a fork are pulled from the mainline variant and pushed into the fork variant. Thus, we defined *DirectPullCom*$_{MLV\text{-}FV}$. Looking at Fig. 5 we can identify the two remaining common commits d and f as *DirectPullCom*$_{MLV\text{-}FV}$.
- *Fork variability percentage*: Here we want to determine how different the FV is from the MLV in terms of the commits/changed lines of code since the fork date. We defined a fork variability percentage for the fork commits as $UniqueCom/(UniqueCom + StartingCommits + MergedCom) \times 100$ and for the fork changed lines of code as $Unique_{cdLOC}/(Unique_{cdLOC} + StartingCommits_{cdLOC} + MergedCdLOC) \times 100$.

B. Results

The results for RQ2 are presented in Figures 6 to 9.

a) Direct pull commits: Figure 6 presents a boxplot showing the distribution of the commits that are cherry-picked from the MLV by the FVs. From the figure, we observe that the median of the cherry-picked commits is at the mark zero. This tells us that the majority of the FVs do not perform cherry picking of commits from the MLV. More specifically, 80 of the 127 FV did not perform any cherry picking of commits since they have been created.

b) PullRequests: Figure 7 shows the distribution of the merged pull requests from MLV to FV in the left histogram, and from FV to MLV in the right histogram. The figure shows that there are a few merged pull requests in both directions. For MLV-FV, we observe a total of 15 merged PRs being sent

[1]https://help.github.com/articles/syncing-a-fork/

[2]https://help.github.com/articles/merging-an-upstream-repository-into-your-fork/

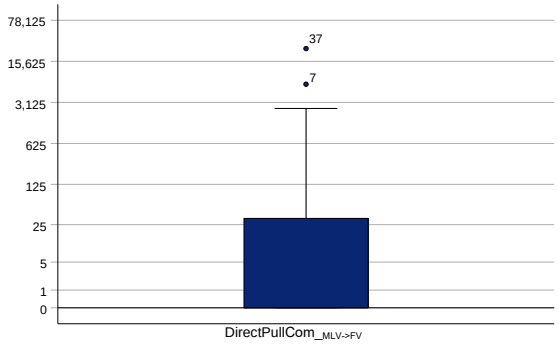

Fig. 6: Distribution of the direct pull commits (*DirectPullCom*$_{MLV\text{-}FV}$) that were cherry picked from the MLV to the FVs (y-axis in log scale)

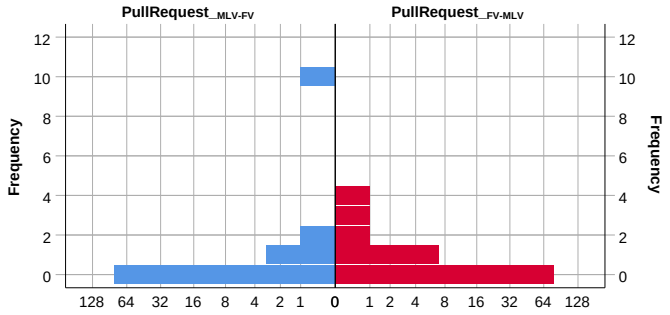

Fig. 7: Distributions of the number of pull requests (MLV-to-FV and FV-to-MLV) that were merged between the FV and the MLV (x-axis in log scale)

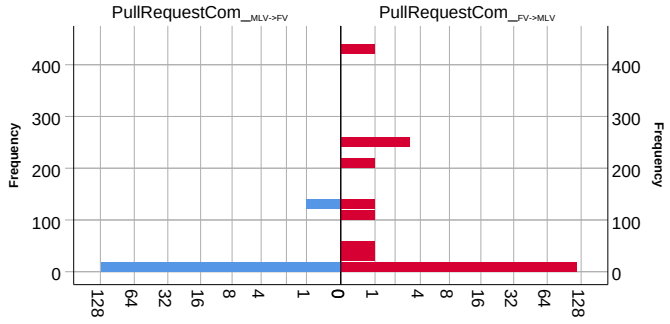

Fig. 8: Distributions of the number of pull request commits (MLV-to-FV and FV-to-MLV) that were sent from the fork variants to the mainline variant (x-axis in log scale)

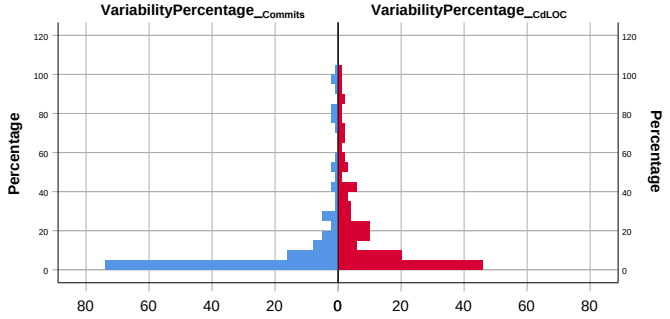

Fig. 9: Distribution of the *VariabilityPercentage* (left histogram) and the *VariabilityPercentage*$_{cdLOC}$ (right histogram)

from five of the 88 MLVs to five of the 125 FVs. For FV-MLV, we observe a total of 16 merged PRs being sent from ten of the 127 FVs to ten of the 88 MLVs.

We were surprised to see PRs being sent from the mainline app to the fork variants, since typically the opposite direction is the more expected route. We decided to investigate this observation more closely. For a given pull request, we identified the developer who initiated the PR and the developer who merged it. In Fig. 5 and in Section V we illustrate how we differentiated the developers in the MLV–FV pair of variants. We discovered that all of the 15 *PullRequestsCom*$_{MLV\text{-}FV}$ were either initiated by a common developer (defined shortly, in Section V), who contributes to both MLV and FV, or by a developer who is only in the FV. In all the 15 *PullRequestsCom*$_{MLV\text{-}FV}$, we did not find any PR that was initiated by a developer who had only authored commits in the MLV. This means that the FV maintainers would go to the MLVs and initiate a PR with the commits they are interested in and send it to the FVs they maintain. The described scenario is another form of cherry-picking we discovered where some developers prefer to use PRs instead of direct pulling of commits described above. For the results of merged PRs from one FV to another FV, we only found one merged PR.

Looking at only the number of PRs does not give the whole picture of the size of the code propagated between the variants. To understand the size of the propagated changes, we also look at the number of commits contained in the merged PRs. Fig. 8 shows the distribution of these commits. We observe that we have large numbers of commits contained in the merged PR of FV-MLV when compared to those of MLV-FV. For example in the right histogram showing $PullRequestCom_{FV-MLV}$ in Fig. 8, we observe that there is one FV that propagated 427 commits to the MLV

c) Fork Variability Percentage: In Fig. 9, we present a stacked histogram of the distribution of the variability indicies of the FVs in relation to their corresponding MLV. The left histogram shows the results of the percentage of FV variability based on the commits and the right histogram shows the corresponding changed lines of code (LOC) of the commits. From the results presented in Fig. 9, we observe that both the distributions are all right-skewed. This means that the majority of the FVs do not differ so much from the MLV in terms of the *UniqueCom* and their corresponding changed LOC. From the statistics of *MergedCom*, we observe that 64 of 88 app families (i.e., 72.7%) have values of zero meaning that the variants in those families did not perform any form of code propagation.

V. Contributor Diversity in App Families (RQ3)

With RQ3, we investigated who contributes to app families, for instance, whether the app variants are typically controlled the MLV developers or by new developers.

A. Methodology

We now describe how we collected data to investigate the commonality of the developers between a MLV and its FVs.

Given two variants in an app family, we define a *common developer* as one who has authored at least one commit in each of the two variants. Recall that in Section IV-A and Fig. 5, we illustrated various types of commits that exist between MLV and FV, and which may result from different code propagation strategies. We can further identify in which variant a given commit was authored. In Fig. 5 we illustrate the commits that were authored in the FV and MLV can be divided as follows:

- *FV Commits*: In a MLV–FV pair, these are commits that were authored in a FV. The FV commits are the sum FV-*UniqueCom* and $PullRequestsCom_{FV\text{-}MLV}$. From Fig. 5, the FV commits are i, ii, iii and iv.
- `MLV Commits`: In a MLV–FV pair, these are commits that were authored in a MLV. The MLV commits are the sum of MLV-*UniqueCom*, $PullRequestsCom_{FV\text{-}MLV}$, and $DirectPullCom_{MLV\text{-}FV}$. From Fig. 5, the MLV Commits are $1, 2, 3, 4, a, b, c, d, e, f$.

For all app families, we collected the *FV Commits* and the corresponding *MLV commits*. We then extracted the developers from the commit details. The commit details we collected included author name, e-mail, login name, and changed lines of code. After collecting the commit details, through manual inspection, we discovered that some contributors of the applications use more than one account for their commits, which causes them to appear as different contributors. To address this issue, we performed name merging to ensure that our data is not polluted with duplicate information that would introduce noise. We merged the details of two contributors into one using the heuristics employed by Businge et. al [12] in a related study of code authorship and fault proneness. Specifically, two contributors are merged into one if (a) they possess the same *login ID*, (b) possess different *login ID* but possess the same *full names*, or (c) possess both different *login ID* and *full names* but have the same *e-mail* prefix (i.e., prior to the email domain name).

In Table I, we outlined the five metrics *TotalDevMLV*, *TotalDevFV*, *TotalDevs*, *CommonDevs*, and *CommonDevs %*), which relate to code authorship. We used these metrics to determine the commonality between the developers of MLV and FV. We illustrate the methodology used to compute the percentage of commonality of developers i.e., the value of *CommonDevs %*, with an example in Fig. 5. The figure comprises a total of three developers: `FV.d1`, who authored commits i,ii,iii, and iv; `MLV.d1`, who authored commits 1, 2, 3, a, and b; and `MLV.d2`, who authored commits c,d,e and f. So, *TotalDevs* = 3. Developers `MLV.d1` and `FV.d1` refer to the same developer who has contributed commits in both the MLV and the FV. This means that we have one common developer but the number of times it appears in Fig. 5 is two (i.e., count(*CommonDevs*) = 2).

In Fig. 5, the percentage of common developers–*CommonDevs %* computed as the $count(CommonDevs)/TotalDevs \times 100$ would be $2/3 \times 100 = 66.7\,\%$.

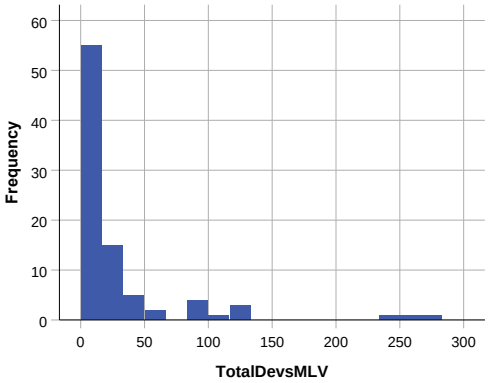

Fig. 10: A histogram showing the distributions of the total developers of the MLV–*TotalDevMLV* for the 88 MLV.

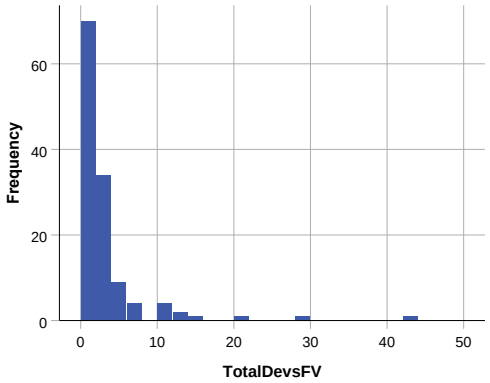

Fig. 11: Distribution of the total numbers of developers of the FV (*TotalDevFV*) for the 127 FVs

B. Results

Figure 10 andFig. 11 show the distribution of *TotalDevMLV* in the 88 MLVs and *TotalDevFV* in the 127 FVs, respectively. As can be seen from the figures, the number of developers in both the MLV and FV are right-skewed (i.e., mean>median), which means that most of the apps (whether MLVs or FVs) are developed and maintained by a few developers. Looking at Fig. 12, we also observe a left-skewed distribution of common developers between the MLVs and the FVs. This indicates that

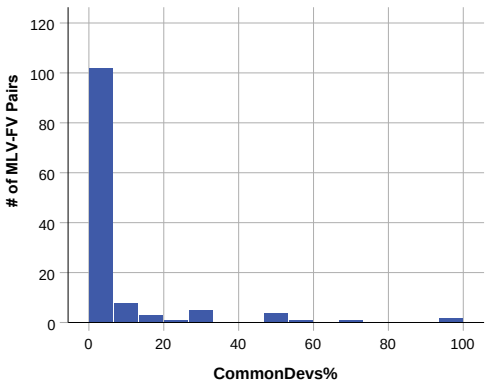

Fig. 12: Distribution of common developers between the MLV and the FVs for the 127 pairs of MLV–FV

there are many MLV-FV pairs that have no common developers. In fact, 94 of the 127 MLV-FV pairs (74 %) do not have any common developers.

VI. Types of Reuse in App Families (RQ4)

With RQ4, we wanted to understand the types of reuse that occur and what kind of variations drive the forking of apps.

A. Methodology

We manually investigated a sample of our 88 identified families. To sample, we first ordered the families by their sizes (number of variants) and then used stratified sampling to select two families from each variant frequency. For each frequency, if more than two families existed, we randomly selected two. If two or less families existed, we analyzed whatever existed.

We obtained eleven families. Specifically, our sample contained one family with ten variants, two families with six variants each, two families with five variants each, two families with four variants each, two families with three variants each, and two families with two variants each.

For each app family, we qualitatively compared the FVs and the MLV by looking at their app descriptions on Google Play and Github, as well as their additions and customizations in the new commits after the fork date on Github. We investigated the app families along the following criteria:

- **Criterion 1: Variant Domains**: The different variants in the app family belong to the same category listed on Gogle Play. These categories comprise: tools, productivity, health & fitness, and so on.
- **Criterion 2: Cherry-picking**: The different FVs in a family perform cherry-picking of commits from the MLV.
- **Criterion 3: Backward propagation**: The FVs in an app family performed backward code propagation to the MLV through pull requests.
- **Criterion 4: Shared developers**: All pairs of MLV–FV in an app family have common developers.
- **Criterion 5: User Competition**: Variants are competing for the same users (explained shortly) on Google Play.
- **Criterion 6: Significant Differences**: The FVs 'significantly' added or changed functionality of the MLV. If the *UniqueCom* commits are just simple customizations of the MLV, then the answer is No. Otherwise, for additions of new features, bug fixes, and so on, it is Yes. We investigated this criterion by first looking at the FV description on Google Play to determine the goal and nature of the app, and then manually inspecting the *UniqueCom* commits of the FV to identify the development activities that occurred.

B. Results

Table III summarizes the results for RQ4, showing the findings for the six criteria for each sampled app family.

We identified four categories of reuse, which we explain in the following four subsections, referring to Table III for additional criteria when needed. Note that some app families fall into more than one category.

1) Re-branding & Customizations: In this category, FVs make simple modifications to the MLV code and are then published on Google Play. Modifications may include changing the user interface, XML/Java package names, logos, server names, and so on. The app families we identified in this category include the MLVs: bitcoin-wallet/bitcoin-wallet, HashEngineering/dash-wallet, opendatakit/collect, DigitalCampus/oppiamobile-android, shadowsocks/shadowsocks-android, and owncloud/android.

In Table III, we see that all the four app families in this reuse category do not have significant differences in the functionality they provide (Criterion 6). On the other hand, there are some differences with respect to Criterion 5 which looks at whether the variants compete on the app store or not. We discuss this aspect further:

- *Competing variants*: In Table III, column–(Criterion5) we present seven families whose variants could possibly be competing for the same users. For example, variants in the app families of the MLVs bitcoin-wallet/bitcoin-wallet, HashEngineering/dash-wallet, and owncloud/android, the variants are possibly competing for the same users on Google Play. For example, the customizations made in the FVs of bitcoin-wallet/bitcoin-wallet and HashEngineering/dash-wallet include changes to the transaction fees. Another example is the FV blaucloud/android of owncloud/android, which made customizations to provide a free app on Google Play, since the MLV is a paid app.
- *Non-Competing variants*: In Table III, column–(Criterion5) we present five families whose variants may not be competing for the same users. For example, in two app families of the MLVs: opendatakit/collect and DigitalCampus/oppiamobile-android, the variants are likely not competing for the same users on Google Play. For example, in one app family, the MLV targets specific user needs in one country, and the FVs reuse the functionality of the MLV by customizing the app to target related user needs in other countries. In a specific example in an app family, the MLV DigitalCampus/oppiamobile-android is a mobile learning application for students of Wits DigitalCampus in South Africa to run training content, quizzes, and video content offline, while the FV CCP-ICT/oppia-mobile-android customized the MLV to deliver the same content to health workers in Nepal. In another example, the MLV opendatakit/collect is a generic data collection app, while the FV anggabayu21/collect is a data-collection app for customizing the MLV functionality to collect specific data for *disaster risk management*. Another FV kobotoolbox/collect customized the MLV to collect data in humanitarian emergencies.

2) Implementation of Different, but Related Features: In this reuse category, FVs in the app families implement different but related features from the MLV. We identify two app families with the MLVs shagr4th/droid48 and k9mail/k-9 in this

TABLE III: Summary of findings for RQ4

Family (MLV)	Reuse Category	Num. of Variants	Criterion 1	Criterion 2	Criterion 3	Criterion 4	Criterion 5	Criterion 6
bitcoin-wallet/bitcoin-wallet	RC	10	Yes	Yes & No	Yes & No	Yes & No	Yes	No
opendatakit/collect	RC	6	No	Yes & No	Yes & No	Yes & No	No	Yes
DigitalCampus/oppia-mobile-android	RC	6	Yes & No	Yes	Yes	Yes	No	No
owncloud/android	SM, FE	5	Yes & No	Yes & No	Yes & No	Yes & No	Yes & No	No
HashEngineering/dash-wallet	RC	5	Yes	No	No	Yes & No	Yes	No
mendhak/Google Playlogger	FE	4	No	Yes & No	No	No	No	Yes
k9mail/k-9	FE, DRF	4	Yes	Yes & No	No	No	Yes	Yes & No
XCSoar/XCSoar	SM, FE	3	Yes	Yes	No	Yes	Yes	Yes & No
shadowsocks/shadowsocks-android	FE, RC	3	Yes	Yes	No	No	Yes	Yes & No
wordpress-mobile/WordPress-Android	FE	2	No	Yes	Yes	No	No	Yes
shagr4th/droid48	DRF	2	Yes	No	No	No	Yes	No

Criterion1 Are the variants in the same domain (Google Play category)?
Criterion2 Have the FVs performed cherry-picking of commits?
Criterion3 Have the FVs performed backward code propagation?
Criterion4 Do the MLV and FVs have common developers?
Criterion5 Is there a possibility of variants competing for users on Google Play?
Criterion6 Are there any significant differences in functionality between MLV and FV?
Yes & No Yes for some variants, No for others
RC Re-branding & Customizations
DRF Implementation of different, but related features
FE Functionality extension
SM Support for MLV

category. A specific example, the FV czodroid/droid48sx is an emulator for the HP 48 SX scientific calculator, which is a modified version of the MLV shagr4th/droid48, an emulator of the HP 48 scientific calculator.

As seen in Table III, all app family variants in this reuse category are in the same domain and also all the variants are likely competing for the same users on Google Play.

3) Functionality Extension: This category of reuse involves FVs that extend the functionality of the MLV. We identified the following app families in this reuse category: mendhak/gpslogger, k9mail/k-9, shadowsocks/shadowsocks, XCSoar/XCSoar, and wordpress-mobile/WordPress-Android. For example, the MLV mendhak/gpslogger is just a very basic GPS tracker app with basic functionality. The FVs dkm/gpslogger and itbeyond/EOTrackMe_Android reuse the MLV's functionality and extend it with additional functionality to perform sophisticated tracking. Another example is the FV micwallace/visualvoicemail, which implements a new visual voice mail feature that is not offered in the MLV k9mail/k-9.

4) Supporting the MLV: We also found FVs that simply offer additional support to the MLV. We identified two app families in this reuse category: owncloud/android and XCSoar/XCSoar. For example, the FV grogg/ownClient is a workaround app to resolve a known bug in Android 4 devices that affects the MLV owncloud/android. Another example is the FV staylo/XCSoar, a testing app used by the developers of the MLV XCSoar/XCSoar to test new features of the MLV. The FV is not intended to be used by normal users.

VII. Threats to validity

Internal Validity. The major threat that could affect the findings are the possible errors as a result of the few manual steps we carried out during our app family data collection. However, during our qualitative analysis of the 11 app families having a total of 50 variants in Section VI, we did not find any MLV or FV that were wrongly linked to Google Play. This gives us confidence in our data collection steps.

Construct Validity. We defined various metrics. Threats to the construct validity are that they are not suited to answer our research questions; that they are not well-defined; or that they are incorrectly calculated. We addressed these threats by: first formulating relevant research questions and then defining the metrics; cross-checking the metrics among the authors, including refining and re-formulating them; and verifying the resulting statistics among the authors. Especially the latter revealed smaller inconsistencies we fixed. Also note that the metrics are defined over different entities. For instance, some metrics are calculated for entire families, while others are calculated for all FVs or apps. We made sure that the entities for which the metrics are calculated are clarified in the definition, for instance, using the formulation "a given FV," which indicates calculation of the metric for individual FVs (so, we provide summary statistics for the respective distribution of the metric over all FVs).

External Validity. A threat is that our results may not generalize to other app families on Google Play. In fact, our scope was focused on open-source apps that are hosted on GitHub. Likely, commercial closed-source app families may be maintained differently, and a dedicated study of such families would be valuable future work, complementing our findings. Another threat is that our app families may be biased towards specific app categories. However, as shown in Section III, our mined apps are well-represented over different app categories.

VIII. Related Work

Only few works study variability management in software ecosystems. Berger et al. [3] study variability mechanisms used in successful software ecosystems. Their focus is on mechanisms that support variability in the whole ecosystem. The authors analyze five ecosystems, including Android, and

identify a spectrum of different mechanisms, related to the target users of the ecosystem. Interestingly, they do not look into clone-based variability management, which, as we show, occurs in ecosystems. In fact, such practices are done in small subsets of the ecosystems, where developers do not use any mechanism offered by the ecosystem platform, such as Android.

Schmid et al. [13] discuss variability ("customization") mechanisms in service platforms, based on a literature review and an industry partner's yard management system. They describe various forms of variability occurring in the platform, and identify static and dynamic variability mechanisms suited for service-oriented platforms.

Seidl et al. [14] introduce an integrated approach to manage variability in space and time in software families using a Hyper Feature Model. The authors' approach allows derivation of concrete software systems from a software product lines or software ecosystems configuring both functionality (features) as well as versions. The authors' goal is to handle the evolution of individual variable assets of the software family by performing the changes on realization assets.

Werber et al. [15] study the application and combination of methods for uncovering variability models from software ecosystems from multi-repository structures in the context of a real-world industrial case study in the health care domain.

All the discussed studies consider ecosystems as collections of individual software projects, managed within their own source code repositories and having references to each. The above authors also discuss variability in the ecosystems and describe the variability forms occurring in those ecosystems. Our study differs from the above work in that it explores clone-based reuse within the Android ecosystem, where smaller app families are developed and maintained in parallel as they cater for different end user requirements or hardware specifications.

The study of Li et al. [2] is related to ours in that it mines Android apps from different market places and presents a vision of carrying out a large-scale, world-wide and time-aware study of reuse practices for automatic assessment of extractive SPL adoption in families of apps. The authors present their initial prototype of app families clustering method. Yet, Li et al.'s study classifies a large number of apps as family members, orders of magnitude more than we actually found. Even though, we were limited to apps hosted on GitHub, the fact that we found only 88 families still indicates that actual (fork-) relationships should be investigated before classifying apps as members of a family.

Finally, Mojica et al. [8] crawl Google Play to study software reuse in mobile apps, finding substantial reuse. Their study indicates that, while these apps benefit from increased productivity, they are also more dependent on the quality of the apps and libraries that they reuse. In comparison, we studied more detailed reuse practices with information mined from both GitHub and Google Play.

IX. Conclusion

We presented an exploratory study of clone-based variability management in the Android Ecosystem. Our main objective was to investigate reuse practices of Android app families in the Android ecosystem on GitHub. We focused on Android app families whose apps appear on Google Play and are, thus, used by end users in practice. We mined and analyzed different properties of the 88 app families we identified from both GitHub and Google Play repositories in order to determine the families' characteristics as well as code-propagation practices and maintainer relationships. In the 88 app families we studied, forked variants are created mainly for: (i) re-branding and simple customizations, (ii) feature extension, (iii) supporting of the mainline app, and (iv) implementation of different, but related features. Surprisingly, we observed that 73 % of the app families did not perform any form of code propagation, and in 74 % there is no single common developer, all variants are maintained by different developers.

Acknowledgment

Sida/BRIGHT (project 317) under the Makerere-Sweden bilateral research programme 2015-2020, NSERC, Vinnova Sweden (project 2016-02804), and the Swedish Research Council Vetenskapsrådet (project 257822902).

References

[1] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *CSCW*, 2012.

[2] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Mining families of android applications for extractive spl adoption," in *SPLC*, 2016.

[3] T. Berger, R. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She, "Variability mechanisms in software ecosystems," *Information & Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.

[4] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.

[5] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: A case study of marlin," in *VaMoS*, 2018.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[7] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.

[8] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large scale empirical study on software reuse in mobile apps," *IEEE Software*, vol. 31, no. 2, pp. 78–86, Mar. 2014.

[9] F. Sattler, A. von Rhein, T. Berger, N. S. Johansson, M. M. Hardø, and S. Apel, "Lifting inter-app data-flow analysis to large app sets," *Automated Software Engineering*, no. 25, pp. 315–346, Jun 2018.

[10] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec 2017.

[11] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *MSR*, 2014.

[12] J. Businge, S. Kawuma, E. Bainomugisha, F. Khomh, and E. Nabaasa, "Code authorship and fault-proneness of open-source android applications: An empirical study," in *PROMISE*, 2017.

[13] K. Schmid, H. Eichelberger, and C. Kröher, "Domain-oriented customization of service platforms: Combining product line engineering and service-oriented computing," *Journal of Universal Computer Science*, vol. 19, no. 2, pp. 233–253, jan 2013.

[14] C. Seidl, I. Schaefer, and U. Assmann, "Integrated management of variability in space and time in software families," in *SPLC*, 2014.

[15] J. H. Werber, A. Katahoire, and M. Price, "Uncovering variability models for software ecosystems from multi-repository structures," in *VaMoS*, 2015.