2016 IEEE International Conference on Software Maintenance and Evolution

Why are Commits being Reverted?

A Comparative Study of Industrial and Open Source Projects

Junji Shimagaki*, Yasutaka Kamei†, Shane McIntosh‡, David Pursehouse*, Naoyasu Ubayashi†
*Sony Mobile Communications Inc., Japan; {junji.shimagaki,david.pursehouse}@sonymobile.com
†Kyushu University, Japan; {kamei,ubayashi}@ait.kyushu-u.ac.jp
‡McGill University, Canada; shane.mcintosh@mcgill.ca

***Abstract*—Software development is a cyclic process of integrating new features while introducing and fixing defects. During development, commits that modify source code files are uploaded to version control systems. Occasionally, these commits need to be reverted, i.e., the code changes need to be completely backed out of the software project. While one can often speculate about the purpose of reverted commits (e.g., the commit may have caused integration or build problems), little empirical evidence exists to substantiate such claims. The goal of this paper is to better understand why commits are reverted in large software systems. To that end, we quantitatively and qualitatively study two proprietary and four open source projects to measure: (1) the proportion of commits that are reverted, (2) the amount of time that commits that are eventually reverted linger within a codebase, and (3) the most frequent reasons why commits are reverted. Our results show that 1%-5% of the commits in the studied systems are reverted. Those commits that are eventually reverted linger within the studied codebases for 1-35 days (median). Furthermore, we identify 13 common reasons for reverting commits, and observe that the frequency of reverted commits of each reason varies broadly from project to project. A complementary qualitative analysis suggests that many reverted commits could have been avoided with better team communication and change awareness. Our findings made Sony Mobile's stakeholders aware that internally reverted commits can be reduced by paying more attention to their own changes. On the other hand, externally reverted commits could be minimized only if external stakeholders are involved to improve inter-company communication or requirements elicitation.**

I. Introduction

While code changes are developed with noble intentions (e.g., to fix a defect or add a new feature), they often inadvertently introduce new defects in the process. Indeed, the mere act of altering a codebase increases the risk of introducing new defects. Case in point, Graves *et al.* [5] find that recently modified code tends to be more defect-prone than older code. Moreover, Nagappan *et al.* [13] find that the amount of change that a component has undergone is a highly effective predictor of where post-release defects will appear.

Since precise debugging is a difficult process, a popular (albeit coarse-grained) approach to recovering from a defect-introducing change is to *revert* it, i.e., undo the change that introduced the defect. Code changes are typically reverted using Version Control Systems (VCSs). For example, Git, Mercurial, and Subversion provide a built-in `revert` sub-command. Otherwise, the Unix `patch` utility, which is used to apply a set of tracked changes to a set of directories or files, offers a `-R` flag to reverse those changes.

Despite the pervasiveness of the `revert` command, little is known about how it is used in practice. A preliminary study by Yoon and Myers [20] shows that 75% of developers feel that a backtracking tool is necessary. Moreover, Codoban *et al.* [3] show that developers often use history tracking tools (like VCSs) to back out problematic changes by reverting to previous, known-to-be working system states.

In this paper, we set out to better understand why commits are reverted in large software systems. First, we perform a quantitative empirical analysis of 3,144 *reverting commits* (i.e., commits that revert another commit) and 2,958 *reverted commits* (i.e., commits that have been reverted by a reverting commit) from two proprietary projects at Sony Mobile and four open source projects.[1] We complement our quantitative analysis with a qualitative one to arrive at a deeper understanding of revert use at Sony Mobile. Overall, our empirical analyses address the following three research questions:

RQ1: What percentage of commits are being reverted?
At most, 5% of commits are reverted in the studied proprietary projects, while only 1% of commits are reverted in the studied open source projects.

RQ2: How long do commits that are eventually reverted linger within a codebase?
In the studied proprietary projects, reverted commits linger for 24 - 35 days, while in the studied open source projects, reverted commits linger for 1 - 12 days.

RQ3: Why are commits being reverted?
We observe 13 reasons for reverting a commit. The distribution of reverted commits per reason varies broadly among the studied projects. For example, reasons involving external parties (e.g., requirement change by end-users, customers, or remote teams) are dominant in the proprietary projects, whereas those involving only internal parties (e.g., coding mistakes) are dominant in the open source projects.

The main contributions of this paper are:

- An empirically-grounded insight into the nature of reverted commits in large software projects that we derive from our quantitative analysis.
- The definition of a classification scheme that describes the reasons for which commits are being reverted.

[1]Our results are shared in https://github.com/yiu31802/icsme2016.

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.83

TABLE I
SUMMARY OF PROJECT DATA.

Project Name	Software	Length	# Commits	Tag from	Tag to
Sony Mobile X	Embedded software for smartphones	6 months	17,537	N/A (*)	N/A (*)
Sony Mobile Y		6 months	10,674	N/A (*)	N/A (*)
Android L	OS for mobile products	9 months	75,452	4.4.2_r2.0.1	5.0.2_r3
Android M		9 months	67,445	5.0.2_r3	6.0.1_r3
Gerrit	Web based code review tool	7 years	7,151	(initial)	v2.11
git-repo	Command-line based git tool	6 years	677	(initial)	v1.12.32

(*) Redacted for confidentiality reasons.

Our quantitative findings make the Sony Mobile stakeholders aware that there are reverted commits that can be avoided by more carefully verifying patches before integration. However, the majority of reverted commits are a symptom of miscommunication in Sony Mobile's development environment, which involves internal and external stakeholders. Plans have been laid out to improve internal and external communication, which will likely yield a lower rate of problematic commits.

Paper organization. The remainder of the paper is organized as follows. Section II describes the design of our empirical study, while Sections III and IV present the results. Section V presents the results of the complementary qualitative analysis that we performed with stakeholders at Sony Mobile. Section VI discloses the threats to the validity of the study. Section VII situates this paper with respect to the related work. Finally, Section VIII draws conclusions.

II. EMPIRICAL STUDY SETUP

In this section, we describe the studied systems and explain our approach to identifying reverted and reverting commits.

A. Studied Systems

Table I provides the overview of the studied systems. The two proprietary software projects X and Y are developed and maintained by Sony Mobile. These software systems run on a mobile handset that is produced by Sony Mobile, which contains a third-party chipset that is produced by Qualcomm Inc. and is often embedded in Android devices. The software systems consist of several components, which are different from one another in terms of the extent of the involvement of external stakeholders. For example, there are original applications like *Movie Creator*,[2] which are primarily developed in-house by Sony Mobile. On the other hand, there are Android components like *frameworks/base*,[3] which are primarily developed by the Android team (= Google Inc.), and Qualcomm components like *device/qcom/common*,[4] which are primarily developed by the Qualcomm team. Those *external* components are also modified by the Sony Mobile's teams. Thus there is a challenge to produce new patches there to minimize source code conflicts, as all parties make development progress. Despite its relatively large number of commits, the Sony Mobile development period is the shortest among the studied systems, mainly due to heavy market share competition that has taken root in the mobile handset industry.

The studied Android projects are mainly developed by Google, but also welcome contributions from the community-at-large. Along with the recent popularity of the Android platform (especially on mobile handset devices), each new release provides plenty of new features, spanning from performance improvement to new hardware support. For instance, the Android Lollipop release added support for televisions.[5] Therefore, the studied Android projects have the largest numbers of commits among the studied systems, despite from relatively short analyzed timespans.

The Gerrit (a web-based code review tool[6]) and git-repo (a tool to interface with projects that are composed of a collection of Git repositories[7]) projects are mainly developed by a community of open source contributors. Unlike the other studied projects, these two projects have much smaller development scale in terms of the numbers of commits. Thus, we target the entire development history of these projects.

B. Identifying Reverting and Reverted Commits

In order to find reverting commits, we scan `git log` messages with the following regular expression:

```
^Revert \".*This reverts commit ([0-9a-f]{40}).*
```

which searches for the fixed string pattern that Git uses to mark revert commits. The pattern includes a reference to the SHA-1 ID of the commit that is being reverted (`[0-9a-f]{40}`). After identifying reverting commits, we find the reverted commits by searching for the SHA-1 IDs that are referenced in the reverting commits.

A single reverted commit can be reverted by several reverting commits. Such a situation is often observed when a software project follows a complex branching strategy. For example, in the case of the Android project, the commit that is associated with review `70850`[8] was reverted by the commit that is associated with review `73884`[9] because of a software defect: *"... continued complaints about not being able to generate bug reports and surfaceflinger crashes..."* The same commit was also reverted by the commit that is associated with review `5832702`[10] on the Android Lollipop release branch. Eventually, when the two branches are merged into the master branch, we observe both reverting commits. In our study, we consider each commit independently, because the reasons for reverting a commit depends on the content of each branch.

III. QUANTITATIVE STUDY RESULTS

In this section, we present the results of our first quantitative study with respect to our two research questions. For each research question, we discuss its motivation, present our approach to addressing it, and present our observations.

[2]http://www.sonymobile.co.jp/myxperia/app/moviecreator/
[3]https://android.googlesource.com/platform/frameworks/base
[4]https://source.codeaurora.org/quic/la/device/qcom/common/
[5]https://www.android.com/intl/en_us/versions/lollipop-5-0/
[6]https://www.gerritcodereview.com/
[7]https://code.google.com/p/git-repo/
[8]https://android-review.googlesource.com/#/c/70850/
[9]https://android-review.googlesource.com/#/c/73884/
[10]https://android.googlesource.com/platform%2Fexternal%2Fsepolicy/+/5832702

RQ1: What percentage of commits are being reverted?

Motivation. The development effort that was spent to produce commits that are reverted is wasted. As a first step toward understanding the nature of reverted commits, we would like to know: (1) what proportion of commits are reverted and (2) whether there is a difference in the proportion of reverted commits among the studied systems.

Approach. We calculate the percentage of reverted and reverting commits in each studied system. To do so, we collect the commit logs of the analyzed periods, i.e., between the 'Tag from' and 'Tag to' columns in Table I. From the collected commits, we identify the reverted and reverting commits using the approach that we describe in Section II-B.

Results. Figure 1 shows the percentage of reverted and reverting commits in each of the studied systems. Overall, 1%-5% of the commits are reverted. The proportions of reverted commits in the Sony Mobile projects are largest at 3% and 5% among the studied systems. While the Android Marshmallow project shows that 2% of commits are reverted, Android Lollipop project has a slightly lower percentage of commits that are reverted at 1%. Both the Gerrit and git-repo projects have the lowest proportions at 0.6% and 1%, respectively.

We find that Sony Mobile projects have larger proportions of reverted commits than the four studied OSS projects. We suspect that the result is likely because Sony Mobile has a heavy dependency on external components, as discussed in Section II-A. For example, one might suspect that some delivered commits from Android are not necessary for the Sony Mobile product, or some built-in features delivered from Qualcomm may be in conflict with Sony Mobile's in-house features. We will discuss what the case is for the Sony Mobile's context and will compare the proportions of such reasons among the studied systems by identifying reasons for reverting commits in Section V.

Overall, 1%-5% of the commits in the studied systems are reverted. Larger proportions of Sony Mobile commits are reverted than the four OSS projects.

RQ2: How long do commits that are eventually reverted linger within a codebase?

Motivation. In RQ1, we find that up to 5% of commits are reverting commits and up to 5% of commits are reverted commits in the studied systems. Although those commits are reversed, they exist for a time within the studied system. Commits that are eventually reverted may cause problems while they linger within the codebase, potentially impeding development progress. A prior study also suggests that early detection of problematic commits, e.g., build breaking commits, can save developer's time [18]. Hence, in this research question, we study the amount of time that reverted commits linger within a codebase.

Approach. We extract the commit date of both the reverted and reverting commits. Then, we compute the number of days between each pair of reverted and reverting commits.

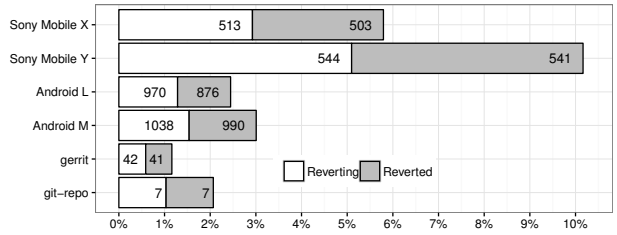

Fig. 1. Percentages of reverting and reverted commits for each studied system represented by the bar lengths. The numbers of reverting and reverted commits are shown inside each bar.

TABLE II
STATISTICS FOR THE NUMBER OF DAYS UNTIL COMMITS WERE REVERTED.

Project	1st Qu.	Med.	Mean	3rd Qu.	Max.
Sony Mobile X	9	35	62	91	769
Sony Mobile Y	10	24	46	55	834
Android L	0	1	17	5	1,190
Android M	0	2	18	8	1,180
Gerrit	2	8	72	103	811
git-repo	2	12	59	62	271

Results. Table II shows the number of days between the reverted and reverting commits for each of the studied systems. Reverted commits linger the largest in the Sony Mobile projects, with projects X and Y having median values of 35 and 24 days respectively. On the other hand, the reverted commits of the Android projects linger for the least time, with median values of 1 and 2 days.

Figure 2 shows the overall distribution of the days between reverted and reverting commits. While the distribution in the Sony Mobile projects is relatively flat, the distribution in other OSS projects (Android projects and Gerrit) is right-skewed.

Similar to RQ1, we find that Sony Mobile projects show different trends with respect to the studied OSS projects. As discussed in Section II-A, one might suspect that the results are explained by Sony Mobile's projects characteristics — software development must be rapidly carried out; even if some code is temporary-made or will be later replaced with an official solution, possibly by partner companies, the entire software operation at Sony Mobile cannot wait. Thus, such temporary-made commits may remain in the software system for a larger time depending on the date of officially-made fixes. We elaborate on this further in Section V.

Commits that are eventually reverted linger within the studied codebases for 1-35 days (median). Similar to RQ1, Sony Mobile projects show different trends than the OSS projects, having a wider distribution of days between reverted and reverting commits.

IV. QUALITATIVE STUDY RESULTS

In this section, we present the results of our qualitative research question (RQ3) to clarify why commits are reverted. We manually classify the reverted commits of the six studied systems. We further confirm our classification with the actual

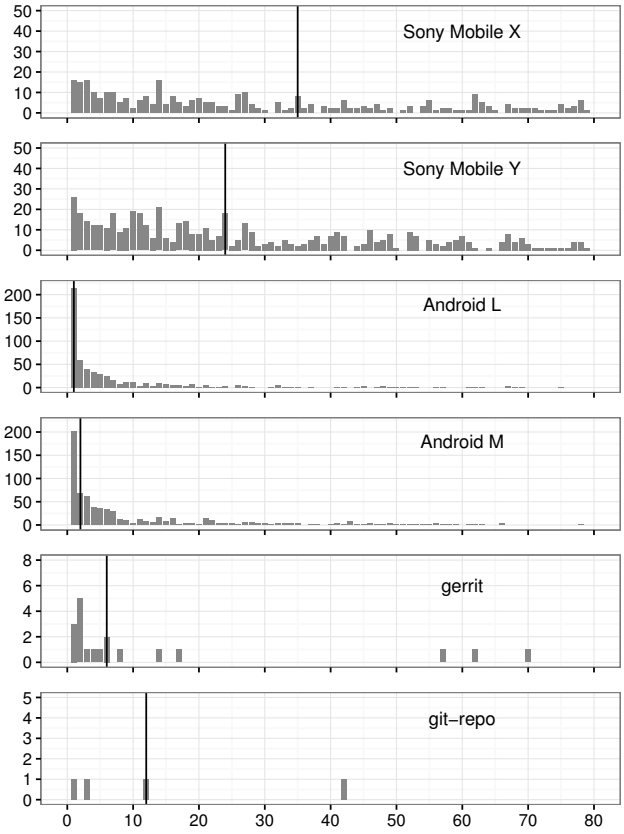

Fig. 2. The number of days that reverted commits linger within the studied systems. The vertical black line indicates the median value.

software engineers at Sony Mobile and by community responsible members of the Gerrit and git-repo projects.

RQ3: Why are commits being reverted?

Motivation. Our analysis of reverting commits in Section III shows that there are substantial differences among the studied projects. We also would like to know if there are differences in the reasons for reverting commits in the studied systems.

Approach. For each reverting commit in the studied Sony Mobile projects, we manually inspect information that is related to it in several data sources (e.g., commit logs and review comments). From this analysis, we arrive at a common set of reasons that commits are reverted, which we use to produce a reverting commit classification scheme (see Table IV). In order to produce our classification scheme, we perform the four iterations of classification that are described below. We apply the scheme to the dataset of our 6 studied systems.

Classification method

Iteration 1. As a first step, we start the classification for reverting commits in the X project of Sony Mobile. This project was chosen because (1) the first author has first-hand experience in its development and (2) the project has adopted a process for carefully documenting reverting commits.

The first author manually inspects the commit messages of all of the reverting commits of the X project and identifies the reasons. While the message of many commits are well-described, some commit messages cannot be classified due to insufficient detail. We mark such reverting commits as the `unknown`, and revisit them in later iterations.

We begin with the categories that were defined in Tao *et al.*'s study of rejected patches [19]. We create a new category when we encounter reverting commits that were motivated by reasons that do not already appear in our classification scheme.

Iteration 2. We classify the data of the other studied systems using the classification scheme that we produced during Iteration 1. First, we use the classification scheme from project X to classify reverting commits in project Y at Sony Mobile. Then, we use the scheme to classify reverting commits in the two Android projects. Finally, we use the scheme to classify reverting commits in the Gerrit and the git-repo projects. Note that this classification step is also performed by manually inspecting all of the messages of reverting commits. The first and fourth author's first-hand knowledge of the studied systems helped to effectively classify these reverting commits.

We then held a post-classification discussion to decide whether the classification scheme needed to be altered. This discussion focused on splitting up the categories that have a very large or very small number of reverting commits. Often, these categories were defined too broadly or precisely, and needed to be redefined accordingly. In cases when the classification scheme was altered, the first author updated the classification for the two Sony Mobile projects and two Android projects, and the fourth author updates the classification for the Gerrit and git-repo projects.

Iteration 3. To verify our classification, we survey stakeholders who were involved in the reverting/reverted commits in the Sony Mobile Y project. We chose the Y project because it is a more recent project than project X. Thus, development decisions would still be relatively fresh in stakeholders' minds.

By analyzing the commit logs, we identified 91 developers as candidates for our survey. To minimize the impact on Sony Mobile development, we randomly selected 30 candidates for our survey. We asked each each candidate to answer the following question: *"Could you explain the background of the reverting/reverted commit A?"* The answers are used to verify our interpretation and to update the classification scheme.

Iteration 4. Some reverting commits could not be classified solely based on the commit message. For example, we cannot classify the commits of which the author omits the reverting rationale. In order to classify as many reverting commits as possible, we analyze the code review history in the Gerrit repositories of each project. This code review history is used to provide additional context, which often helps us to identify the revert rationale. In addition to enforcing code review policies on software projects, Gerrit explicitly links code review records to integrated commits in the project's VCS.

Results for Classification

Table III shows the commits we classify in each iteration.

Iteration 1. For the 513 reverting commits of the X project of Sony Mobile, we begin our classification using Tao *et al.*'s categories of rejected patches in open source projects [19]. We adopt four of the reasons for patch rejection for our study, i.e., (I1) `compilation error`, (I2) `incomplete fix`, (I3) `introducing bugs`, and (I4) `wrong direction`. We extend the classification to include eight additional reasons, i.e., (I5) `mis-operation of delivery`, (I6) `delivery process`, (E1) `temporary workaround`, `requirement change`, `dependency`, (E3) `refactoring`, (E6) `unmatched baseline`, and (E7) `investigation`.

Iteration 2. After applying the classification to the other projects, the post-classification discussion identified two additional reasons for reverting commits. First, we notice that `requirement change` is too abstract and there are too many reverting commits that fit its definition. When we examine the commits that were reverted due to `requirement change`, we find that many of them should be classified as `dependency` (e.g., due to the changes in external libraries).

Second, we notice that commits that were reverted because of (I3) `introducing bugs` have a large variance in the dates until commits are reverted. Most of such commits are reverted within a few days whereas some are only reverted months later. We observe that the main reason is caused by who discovers the issue(s). It takes a longer time to propagate software defects to end-users and other team members than to internal testing teams. Thus, we make a reason (E5) `user's feedback` to distinguish between (I3) `introducing defect` depending on the distance between the bug discovery and the change author.

Table III shows the outcome of iteration 2. 113 commits in the Sony Mobile X project are updated. 98 commits that were previously classified as `requirement change` are reclassified to `dependency`, and 15 commits previously classified as (I3) `introducing bugs` are reclassified to (E5) `user's feedback`. Commits in all other projects are newly classified in iteration 2. Commit messages in the Android projects tend lack lack detail. Thus, 762 and 766 commits classified as `unknown` in the Android Lollipop and Marshmallow projects respectively. The Android developers also find this problematic, as mentioned in one commit: *"A: I can't remember what the original reason for the revert was... — B: this is why you should add a comment on any revert ... :-P"*.[11] All reverts in the Gerrit and git-repo projects are verified by the fourth author, who has experience with the projects.

Iteration 3. We received survey responses from 20 of the 30 candidate stakeholders (67% response rate). These respondents were involved with 81 of the reverting commits in project Y (15% of the reverting commits). More specifically, the maximum number of reverting commits that any one particular developer was involved with is 20, while the median is 2.

TABLE III
CLASSIFICATION EVOLUTION OVER TIME.

Project	Iter. 1	Iter. 2	Iter. 3	Iter. 4	?	Clas. / Total
Sony Mobile X	+406 −0	+113 −113	+63 −63	+0 −0	107	406 / 513
Sony Mobile Y	-	+405 −0	+66 −66	+9 −0	130	414 / 544
Android L	-	+208 −0	+7 −7	+58 −0	704	266 / 970
Android M	-	+272 −0	+17 −17	+66 −0	700	338 / 1038
gerrit	-	+42 −0	+14 −14	+0 −0	0	42 / 42
git-repo	-	+7 −0	+3 −3	+0 −0	0	7 / 7

The stakeholder responses match our interpretation of 71 reverting commits out of the total of 81 reverting commits. We therefore believe that our classification is reliable for the Sony Mobile projects. That said, for the other 10 reverting commits, their answers provide even deeper insight into the commits that were reverted because of a `requirement change`. For example, we obtain the following comments in our survey:

A. *"The requirements changed after discussion of the design of other components with their teams. Our code needed to adapt to reflect these requirement changes."*
B. *"The feature [...] was dropped."*

Although responses A and B seem similar, the fundamental reasons for reverting differ. Case A is caused by awareness of the constraints that the design of other software component put on project Y development. To account for this, we introduce a new category (E3) `obsolete solution`. On the other hand, case B does not contain any inconsistency in the implementation. However, the previously required features are no longer necessary. To account for such cases, we introduce a new category (E2) `unnecessary feature`.

Due to a high degree of similarity, we merge the `dependency` category with (E3) `obsolete solution`. Furthermore, we remove the `requirement change` category because it is covered by (E2) `unnecessary feature` and (E3) `obsolete solution`.

In iteration 3, all commits that were previously classified as `requirement change` are reclassified according to the new scheme. Table III shows that the Sony Mobile X and Y projects are the most impacted by iteration 3, with 63 and 66 commits being reclassified, respectively. Comparatively, only 3 to 17 commits are impacted in the other studied systems.

Iteration 4. To minimize the amount of reverted commits that are classified as `unknown`, we analyze the code review information. In the Sony Mobile projects, the code review information allows us to classify an additional 9 commits of the Y project. On the other hand, no additional reverting commits could be classified in project X. Many of the reverting commits in the Android projects have transparent review discussions.[12]

[11] https://android-review.googlesource.com/#/c/99879/

[12] For example: http://android-review.googlesource.com/109372

TABLE IV
CLASSIFICATION SCHEME.

	ID	Reasons	Characteristics	Example of comments	URL
Internal parties	I1	Compilation error	The original commit introduced compilation error of the system.	*Breaking some build... will look in to it later.* *It breaks cross compilation with x86_64.*	◊/96813 ◊/100739
	I2	Incomplete fix	The original commit partially resolves the issue but is imperfect.	*Revert this patch to give us more time to investigate.* *... more testing to make sure OTA updates aren't broken...*	◊/78673 ◊/78634
	I3	Introducing bugs	The original commit introduced side-effect bugs and/or failed in later performed test.	*This is causing runtime restarts on flo/deb when uninstalling some APKs* *This causes YouTube to crash on launch.*	◊/69276 ◊/81200
	I4	Wrong direction	The committer misunderstood how to resolve the issue.	*This was a mistaken attempt to fix bug* *The right resolution is to fix it in framework or HAL*	◊/66024 ◊/159751
	I5	*Mis-operation of delivery	Attributes of the commit contain incorrect information, or fault in code submission processes.	*Wrong bug id listed on comment* *commit 467a680 is present in lollipop-cts-dev but not lollipop-release.*	△1 ◊/160150
	I6	*Delivery process	The commit needs to move away for branching strategies.	*i plan on reverting this after cherrypicked it internally.* *I think we can re-land this change.*	◊/110164 ◊/111254
External parties	E1	*Temporary workaround	Temporary and locally made fixes are reverted when official fixes are made.	*libvpx is now fixed.* *The underlying issue has been fixed*	◊/103509 ◊/102618
	E2	*Unnecessary feature	The original commit was once valid but later become unnecessary because of requirement change.	*Feature no longer needed;* *This CL did not ship with diamond-release, but diamond-mr1*	△2 △3
	E3	*Obsolete solution	The solution designed by and composed of the original commit is no longer valid.	*code needs re-implementation due to dependency* *We've committed a better fix*	N/A △4
	E4	*Refactor	Restructuring involving other components.	*The users of public getSpi have been migrated to getCurrentSpi*	△5
	E5	*User feedback	A broad sense of users, including customers, live-users find issues and submit an issue report.	*The original fix seems to have led to boot failures in QA* *some unexpected cellbroadcast message come in*	△6 △7
	E6	*Unmatched baseline	Features in the external codebase which are harmful are reverted.	*We are not using this feature but it conflicts with our feature.*	N/A
	E7	*Investigation	Limiting software features or printing debug log to test focused feature is done in practice. Such commits are unnecessary in the release software.	*temporary revert while we investigate* *Reverting debug message prints.*	△8 △9

* Newly classified reasons in this study from the prior study by Tao *et al* [19] N/A Mainly observed in commits of Sony Mobile

◊ Link to corresponding Android commit by adding a prefix https://android-review.googlesource.com/ . E.g., https://android-review.googlesource.com/96813

△ Link to corresponding Android commit by adding a prefix https://android.googlesource.com/ with corresponding suffix (1-9):
1. platform/frameworks/base/+/1a07846, 2. platform/frameworks/base/+/69e6501, 3. platform/frameworks/base/+/6c0b5b3, 4. platform/external/sqlite/+/0266b37, 5. platform/libcore/+/834660d, 6. platform/system/vold/+/223fd1c, 7. platform/packages//apps/CellBroadcastReceiver/+/08454ad, 8. platform/frameworks/av/+/5dcaebb, 9. platform/cts/+/950feb3

Thus, we were able to reclassify 58 and 66 commits of the Lollipop and Marshmallow projects, respectively.

Summary. Table IV provides an overview of the 13 reasons that make up our final classification scheme. 6 of the 13 reasons are due to internal parties (e.g., commit authors, code reviewers). 7 of the 13 reasons are due to external parties (e.g., other development teams, externally developed codebases, clients).

Among the commits that were reverted due to internal causes, most are assumed to have a large impact on software quality and development efficiency. For example, (I1) `compilation error` is crucial for developers because the latest source code does not cleanly build. In the previous study, many developers also mentioned that compilation error is a decisive reason for rejecting a patch [19]. Except for I5 and I6, which are caused by lack of negligence but do not really impact software behavior, we believe commits due to internal causes are in general having a large impact.

On the other hand, reverted commits that were reverted due to external causes, by definition, do not directly introduce problems per se. Instead, due to a change in dependent components or software requirements, those reverted commits should not persist in the system any longer, otherwise breaking the coherency in the software implementation (i.e., E1, E3, E4 and E6 in Table IV) or in the software requirements (i.e., E2, E5). Although this type of impact is not as direct as the impact originated of internal causes, we argue that commits that were reverted for external reasons indeed some impact, albeit but from a higher level of software design.

Of the 3,104 extracted reverting commits from the studied systems, 1,473 reverting commits (47%) could be classified according to our 13-reason classification scheme.

Results — Sony Mobile

Figure 3 shows the classification result for revert reasons in the X and Y projects at Sony Mobile. From the category perspective, internal reasons account for 20% and 14% in the X and Y projects, respectively. Within the internal parties category, (I3) `Introducing bugs` accounts for the largest share, whereas (I2) `Incomplete fix` accounts for a smaller share. This agrees with the intuition that revert operations are used to quickly (but coarsely) fix defects. Note that (I5) `Mis-operation in delivery` has the second largest share, which implies that consistency in internal development data, e.g., bug ticket status, is not only important for researchers [1], but also for practitioners as well.

Within the external reasons, two categories share the largest proportions: (E1) `Temporary workaround` and (E3) `Obsolete solution`. `E3` is quite an important observation because there are frequent Sony Mobile codebase updates that are imported from upstream branches of Android and Qualcomm. Our data shows that many developers often need to redesign solutions due to API updates and architectural restructuring on upstream branches. `E1` also happens frequently because of similar reasons as in `E3`—while internal developers are waiting for an issue fix by the developers on the upstream side, internal developers make temporary commits to allow internal development to proceed. For instance, one function call, which upstream developers are responsible for, was inducing crashes of the Sony Mobile system. In this case,

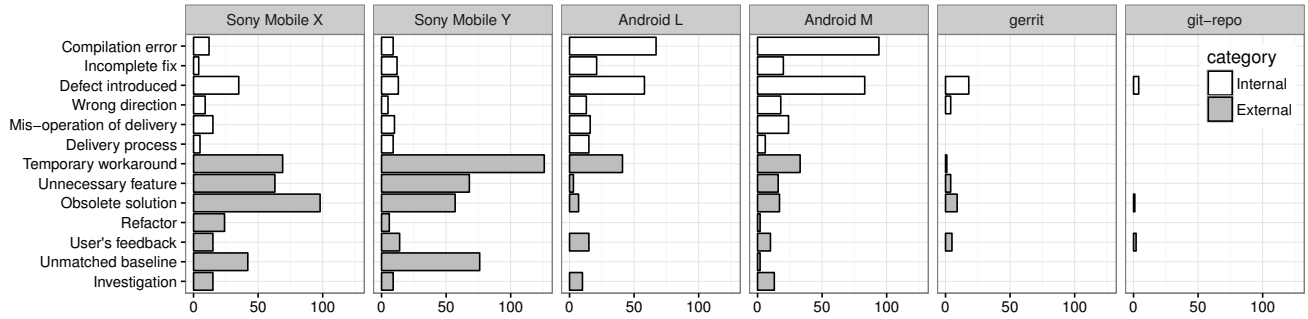

Fig. 3. Distributions of revert reasons per studied systems.

an internal developer needed to temporarily revert the change that introduced the function call until the issue was resolved.

(E2) `Unnecessary feature` accounts for the third-most reverting commits. We observe that it is often due to the fact that the studied system has customers that span the globe. This leads to a turbulent set of requirements that can be rapidly changed or suddenly dropped. On the other hand, (E6) `Unmatched baseline` accounts for a large share. Even though the studied system has a policy to retain the upstream code as much as possible, some commits cannot co-exist with internal commits. For instance, if the upstream commits implement features that are harmful to Sony Mobile internal features, such upstream features may need to be reverted.

In the Sony Mobile projects, due to dependencies on other parties, 326 and 356 reverted commits (80% and 86%) are due to external factors.

Results — Android

Figure 3 shows the classification result for reverting commits in the studied Android projects. From the category perspective, internal reasons account for 71% and 72% of the reverting commits in the Lollipop and Marshmallow Android projects, respectively. Since RQ2 shows that reverting commits tend to appear within a few days for the Android project, we suspect that internal reasons should dominate the reverting rationale. The relatively low proportion of external reasons for reverting commits is also expected due to the fact that most of the commits in the Android project are developed within the community, as opposed to the Sony Mobile projects, where a large amount of the codebase is inherited from external sources. The internal nature of development can also make requirements elicitation and code design easier than involving external vendors.

Among the internal reasons for reverting commits in Android, the dominant reasons are (I1) `Compilation error` and (I3) `Introducing bugs`. Interestingly, both top categories are also observed to be the most frequent reasons for patch rejection in prior work [19].

Unlike the Sony Mobile projects, internal reasons for reverting commits are more prevalent than external ones in Android projects. This is likely because of the in-house nature of the majority of Android development.

Results — Gerrit and git-repo

Figure 3 shows the classification result for reverting commits in the studied OSS projects. Although reverting commits are much less frequent in the OSS projects, there are still interesting trends that occur. For example, similar to Android, (I3) `Introducing bugs` accounts for the lion's share of the reverting commits in the studied OSS systems. Several revert commits claim that the submitted code caused test failures, e.g., *"... Appears to have broken the acceptance test suite..."*[13] implying that insufficient verification was not performed before integration. The second-most prevalent reason for reverting commits, (E3) `Obsolete solution`, is often caused by removal of a feature that is no longer supported after downgrading to an earlier version of a dependency library, as mentioned in the commit message:[14] *"... This change is not compatible with SSHD 0.9.0 which is being brought into master by merges from stable-2.9 ..."* This particular feature removal was implicated in y of the 41 reverting commits.

Similar to the studied Android projects, internal reasons for reverting are more prevalent than external ones.

V. Discussion

In this section, we perform a qualitative analysis of the results of our empirical study in Section III and IV. The main aim of our qualitative analysis is to answer question: *"Why are our results from the Sony Mobile context so different from the other studied projects?"*

Approach. We issued a developer questionnaire at Sony Mobile to identify the potential reasons for the discrepancies between Sony Mobile and the other studied systems. The questionnaire consists of the following open-ended questions that roughly mirror our research questions:

- Why does Sony Mobile have more reverted commits? (RQ1)
- Why do reverted commits persist longer at Sony Mobile? (RQ2)
- Should Sony Mobile teams try to avoid reverting commits? (RQ3)

[13]https://gerrit-review.googlesource.com/#/c/45911/
[14]https://gerrit-review.googlesource.com/#/c/62290/

TABLE V
MEDIAN DAYS UNTIL COMMIT IS REVERTED.

	I1	I2	I3	I4	I5	I6	E1	E2	E3	E4	E5	E6	E7
X	2	12	6	9	1	6	43	41	82	63	83	48	20
Y	1	8	4	11	2	17	21	50	35	90	37	29	7
L	0	1	1	3	0	1	7	1	67	-	32	-	19
M	1	1	1	5	1	4	6	28	28	56	22	117	26

We collect answers from the developers and triangulate our findings with the results of Section III and IV to arrive at concrete implications to each question.

A. Why does Sony Mobile have more reverted commits?

In RQ1, we observe that the proportions of reverting and reverted commits in the Sony Mobile projects are substantially higher than those of the other studied projects. In RQ3, on the other hand, we clarified that the majority of reverting commits at Sony Mobile are due to factors that are outside of the control of the Sony Mobile team (i.e., external reasons). In the following, we discuss two implicated causes, which are indicated by the feedback of Sony Mobile developers, and also supported by the quantitative data analysis of Section III.

(A-1) A strong dependency on an externally-developed codebase increases the rate of reverted commits. This implication is best depicted by the response of a software architect, whose project relies on components that are developed by Qualcomm:

> *"Our codebase is built upon Google and Qualcomm code. They often update their code in ways that impact our internal code in unexpected ways. This often requires redesigning our code, i.e., our initial commits must be reverted."*

Due to the fact that Sony Mobile's code is built upon an externally developed (and rapidly changing) stack, internally developed code often ends up in conflict (or worse, incompatible) with the underlying dependencies when the externally developed code is updated. This agrees with the implication from our previous study showing that quality of the code is strongly related with the existence of external code integration [17]. In such cases, the internal code needs to be redesigned to resolve the conflict—the internal commits are reverted due to `E3`. In the Sony Mobile X and Y projects, 89% and 78% of reverted commits belonging to category `E3` are caused by incompatible updates to the externally maintained components of the development stack. In the Android projects, on the other hand, we did not find similar reasons for revert commits, likely because of the project's relative independence from external dependencies (with the obvious exception of the Linux kernel). Herbsleb and Grinter [6] found that face-to-face communication indeed helps to avoid mis-communications, which Sony Mobile's stakeholders may have lack of when working with external organizations. Therefore, we suspect that other such projects with dependencies on external codebases may also suffer from high rates of reverted commits.

(A-2) Reverting commits is necessary part of the development process for large systems with rapidly changing external dependencies. One senior requirements analyst explains the unique aspects of the development process at Sony Mobile:

> *"Even if a crash is due to a bug in a vendor's proprietary code, we need to manipulate our program flow to avoid using that module, while still keeping the functionality. To do so, we need to make temporary workarounds that are reverted when a vendor fix is delivered."*

In our classification scheme, such temporary workarounds are categorized as `E1`. As the requirements analyst states, there is a development process to handle such urgent cases when temporary fixes must be delivered quicker than the official fix is made (e.g., when system testing is scheduled to occur before the vendor delivers a fix). In the Sony Mobile projects, 76%-96% of the revert commits due to `E1` involve external parties. Considering that `E1` accounts for most reverting commits in projects X and Y, this unique development process is likely a factor that increases the number of reverted commits.

> *Workarounds for problems in external components of the Sony Mobile development stack account for 17% to 31% of the reverted commits.*

B. Why do reverted commits persist longer at Sony Mobile?

In RQ2, we observe that reverting a commit in Sony Mobile tends to take a longer time than in the other studied software projects. Table V shows the median days until commits are reverted according to our classification scheme. We find that reverted commits that are due to external factors linger within the codebase for longer than reverted commits that are due to internal factors. Below, we discuss why this might be the case.

(B-1) Requirement change in a later phase is expected in new product development. A software developer explains why his enhancement code recently needed to be reverted:

> *"Some features are dropped as per the request of customers or product managers."*

As the studied project is a software project for a new smartphone product, new features and applications are developed based on volatile requirements that product managers and other stakeholders collect from end-users. These features need live-testing to make sure that their performance and quality is sufficient for all devices. In other words, there are features which do not meet the quality criteria for the new product, e.g., a feature negatively affects memory usage, CPU utilization, or energy consumption, which have a direct impact on customer satisfaction. In such cases, these corresponding commits may be reverted (categorized as `E2`), but this kind of decision normally takes time. Considering that `E2` accounts for 9% and 11% of the reverting commits of projects X and Y, respectively, `E2` is likely to raise the total amount of time that reverted commits linger in the Sony Mobile codebase.

(B-2) Temporary workarounds live longer than anticipated if the partner organization needs time to address the underlying cause. As discussed in A-2, Sony Mobile has a development process that relies heavily on external parties. The same requirements analyst addresses the question about why reverting commits at Sony Mobile tend to take a long time:

> *"We request the external parties to fix bugs. Their delivery generally takes time because they have their own triage,*

QA, and development processes before they can release the official fix to us."

Naturally, inter-organizational communication is slow for many reasons, e.g., time zone difference [6, 7]. On top of that, even after the official fix is delivered, the Sony Mobile quality assurance and development teams need to be very careful when replacing the temporary workaround with the official one, because such software defects can have a serious impact the customer experience. As a consequence, temporary workarounds can take a long time to be reverted.

Risk of severe regression causes temporary workarounds to linger for a long time before being reverted.

C. Should Sony Mobile teams try to avoid reverting commits?

In RQ3, we classify reverting commits into two broad categories: those related to internal factors and those related to external factors. The stakeholder questionnaire reveals additional information about each type of reverting commit.

(C-1) Revert commits that are related to internal factors can be avoided by being more careful. There are indeed commits that are caused by developer mistakes. For example, a developer mentions why he failed to find a compilation error before integrating changes:

"Increasing supported devices makes it hard for developers to verify that a code change will work in all product variants. Occasionally, insufficient testing and incomplete commits slip through verification and need to be reverted."

According to Table V, commits that are reverted due to internal factors are often addressed quickly. It is an indication that developers might have submitted commits without complete testing, as expressed by the developer above. A similar situation is also observed among software engineers at Mozilla, where growth of the codebase inflated conflict [18]. Although all of the studied systems employ the code review practices using Gerrit [14], it does not mandate minimum review quality criteria. Previous studies showed that lax code review practices can impact software quality [9, 10, 17] and design quality [11]. With the help of thorough code review process and more complete pre-integration automated testing, these problematic commits can be detected and fixed before they land on official development and release branches.

(C-2) Reverting commits are a symptom of the development process. Several developers mention that reverting commits are not all always a bad thing:

"Reverting commits is often unexpected. However, it is beyond our scope to avoid such reverting commits due to requirements or external component changes. Our current development process, which results in some amount of revert commits, is not all bad."

We find that the majority of reverted commits at Sony Mobile are due to external factors. It is rarely possible to avoid these reverting commits.

Commit that are reverted for internal reasons can be addressed by adding more complete automated tests. On the other hand, commits that are reverted due to external factors may just be the "cost of entry" for projects with rapidly evolving external dependencies in their development stack.

VI. Threats to Validity

Construct validity. We used two data sources to classify reasons for reverting commits: commit log and code review discussion. Although the sources that we used are described well enough to understand the context behind them, there is no guarantee of the correctness of the information. To minimize the risk in the Sony Mobile dataset, we conducted an e-mail survey and interviews. We made sure that the stakeholder responses matched our interpretation of 71 reverting commits. We also exclude commits with a lack of information from the Android projects and leveraged the expert knowledge of the fourth author, who is a maintainer of the other two projects.

Internal validity. We manually inspected revert commits to build a classification scheme for the reasons for reverting a commit. The process of building this classification scheme may be subjective. To minimize this threat, we surveyed the previous studies [12, 19] and bootstrapped the process using the schemes provided by them. Furthermore, two authors discussed the resulted scheme and surveyed 20 stakeholders who were involved in either the reverting or reverted commits.

External validity. We analyze two proprietary projects at Sony Mobile and four open source projects. Our selection of subjects may introduce bias. To mitigate the risk, we select projects of various size, with different development cultures.

VII. Related Work

In this section, we discuss the related work with respect to patch rework and qualitative analysis of software repositories.

Patch rework. In software development, rework takes various forms. For example, before a patch is checked into the main software repository, patches can be rejected in the code review. Both Beller *et al.* [2] and Mäntylä and Lassenius [12] investigated for what reasons commits are encouraged to be reworked in the code review process. Counter-intuitively, they found that indeed functional defects are found but more *evolvable* defects that worsen the maintainability of the code are found. Tao *et al.* [19] manually inspected 300 rejected patches to investigate reasons for their rejections in Eclipse and Mozilla projects. They produced a comprehensive list of reasons for patch rejection. Their results suggest that the most decisive reasons for patch rejection are patches being incomplete, or the existence of a preferred alternative implementation. These reasons for rejection can be used to rework the patch to produce one that will be deemed acceptable. In this study, we find that temporary workarounds, which are intentionally reworked later, are common at Sony Mobile.

After the code is committed to the main software repository, there are cases where rework is still needed. Shihab *et al.* [16]

studied the characteristics of software defects that are re-opened after being fixed. Based on software metrics that capture the likelihood of being re-opened in the future, they showed that prediction of re-opened bugs can be achieved while the best indicators vary depending on the studied project. Souza *et al.* [18] studied how the backout rate changes over time during a large change to the software development process at Mozilla. They found that the rate of reverted commits (i.e., the backout rate) increases after employing rapid release scheduling, i.e., shortening the software release cycle. In our study, we find that development context impacts the rate of reverted commits, with external factors leading to the majority of reverted commits at Sony Mobile.

Qualitative analysis of software repositories. In this paper, we set out to better understand why commits are reverted in various development settings. To do so, we qualitatively analyze commits using manual classification of reverted commits and surveying software engineers at Sony Mobile. There are many other studies with the similar approaches to classify software repository data, which is otherwise hard to characterize quantitatively. For example, Gousios *et al.* [4] surveyed 749 developers in GitHub community to understand the GitHub pull-based development model from the integrator's perspective. According to their study, most of the pull requests that system integrators receive are bug fixes. Also, pull requests of bug fixes are more easily accepted than those of other types, e.g., refactorings or enhancements.

Hindle *et al.* [8] manually classify large commits. They compared ordinary (i.e., 'small') commits with large commits, and found that small commits are primarily bug fixes, whereas large commits tend to involve broad architectural changes.

Saha *et al.* [15] studied how long-lived bugs are different from short-lived bugs. Through quantitative and qualitative analyses on large open source systems, they observe that long-lived do not always imply large-scale software fixes.

Aranda and Venolia [1] surveyed developers at Microsoft to study different scenarios of the life of software bugs. They interviewed 26 engineers at Microsoft. Their results suggest that even small bugs are accompanied by many non-technical factors, which do not appear in bug repositories, e.g., social relationships and organizational matters.

Table VI provides an overview of the related work in terms of three dimensions that are within the scope of this paper. In addition to the target phenomenon, Table VI shows: (1) the proportion of events that are targeted by the paper (Prop.), (2) how long do the events start and end (Time), and (3) the reasons why the events happen (Clas.). For example, Shihab *et al.* [16] contrasts the number of re-opened bugs with the number of bug reports. Therefore, the column Prop. is checked. However, Shihab *et al.* do not study how this proportion changes over time nor why certain bug reports are re-opened. Thus, the Time and Clas. columns are not checked. In short, while previous work studies several phenomena in software repositories, this paper focuses on reverted (and reverting) commits, covering three dimensions that provide insights for better understanding why commits are reverted in large systems.

TABLE VI
SYNTHESIS OF RELATED WORK.

Paper	Prop.	Time	Clas.	Target phenomenon
Beller *et al.* [2]			✓	Defects in code review
Mäntylä and Lassenius [12]			✓	Defects in code review
Tao *et al.* [19]			✓	Rejected patches
Shihab *et al.* [16]	✓			Re-opened bugs
Souza *et al.* [18]	✓	✓		Early and late backouts
Gousios *et al.* [4]	✓		✓	GitHub pull requests
Hindle *et al.* [8]	✓		✓	Large commits
Saha *et al.* [15]	✓	✓	✓	Long lived bugs
Aranda and Venolia [1]			✓	Bug fixing coordinations
Our study	✓	✓	✓	Reverted commits

Abbreviation: Prop. = Proportion, Clas. = Classification

VIII. CONCLUSION

In this paper, we analyze the proportion of commits that are reverted and the amount of time that such commits linger within a codebase. We find that reverted commits are most prominent in the Sony Mobile projects, with revert rates of 3%-5%. By way of comparison, the studied open source projects have revert rates of roughly 1%. These reverted commits linger within the studied codebases for 1-35 days.

We also qualitatively study the reasons for why commits are reverted. We manually inspect 3,144 reverting commits and identify 13 common reasons for reverting commits. The frequency of reverted commits of each reason varies depending on the studied project. For example, in the Sony Mobile setting, the most prominent reasons for reverting commits involve external stakeholders, e.g., features being dropped due to volatile customer requirements. On the other hand, in the other studied projects, the most prominent reasons for reverting commits involve internal stakeholders, e.g., introducing defects due to insufficient testing.

Our findings have made Sony Mobile stakeholders aware that more testing before integration can yield a lower rate of problematic commits. However, the unique development processes involving external stakeholders poses unique challenges for Sony Mobile stakeholders, who must avoid external miscommunication. Our findings have inspired stakeholders at Sony Mobile to improve the development process involving the external parties.

Repeatability. To foster replication and extension of our work, we share our final classification results for 4 OSS projects at https://github.com/yiu31802/icsme2016.

Acknowledgment. This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306. The authors would like to thank anonymous stakeholders at Sony Mobile for participating in the survey. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Sony Mobile and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of Sony Mobile's products.

REFERENCES

[1] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2009, pp. 298–308.

[2] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2014, pp. 202–211.

[3] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: A study on why and how developers examine it," in *Proc. of the Int'l Conf. on Software Maint. and Evolution*, 2015, pp. 1–10.

[4] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 358–368.

[5] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[6] J. D. Herbsleb and R. E. Grinter, "Architectures, coordination, and distance: Conway's law and beyond," *IEEE Software*, pp. 63–70, 1999.

[7] J. Herbsleb and R. Grinter, "Splitting the organization and integrating the code: Conway's law revisited," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 1999, pp. 85–95.

[8] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2008, pp. 99–108.

[9] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects," in *Proc. of the Working Conf. on Mining Software Repositories (MSR)*, 2013, pp. 192–201.

[10] S. McIntosh, Y. Kamei, B. Adams, and A. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, pp. 1–44, 2015.

[11] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects," in *Proc. of the Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2015, pp. 171–180.

[12] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Trans. Software Engineering*, pp. 430–448, 2009.

[13] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, 2005, pp. 284–292.

[14] G. C. R. O. S. Project, "Gerrit code review - a quick introduction," https://gerrit-documentation.storage.googleapis.com/Documentation/2.12/intro-quick.html.

[15] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *Proc. of the Int'l Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 144–153.

[16] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K. Matsumoto, "Predicting re-opened bugs: A case study on the Eclipse project," in *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, 2010, pp. 249–258.

[17] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "A study of the quality-impacting practices of modern code review at Sony Mobile," in *Proc. of the Int'l Conf. on Software Engineering (ICSE), Software Engineering in Practice (SEIP)*, 2016, pp. 212–221.

[18] R. Souza, C. Chavez, and R. A. Bittencourt, "Rapid releases and patch backouts: A software analytics approach," *IEEE Software*, vol. 32, no. 2, pp. 89–96, Mar 2015.

[19] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *Proc. of the Int'l Conf. on Software Maintenance and Evolution (ICSME)*, 2014, pp. 271–280.

[20] Y. Yoon and B. A. Myers, "An exploratory study of backtracking strategies used by developers," in *Proc. of the Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2012, pp. 138–144.