Will My Patch Make It? And How Fast?

Case Study on the Linux Kernel

Yujuan Jiang, Bram Adams
MCIS, Polytechnique Montréal, Canada
{yujuan.jiang,bram.adams}@polymtl.ca

Daniel M. German
University of Victoria, Canada
dmg@uvic.ca

Abstract—**The Linux kernel follows an extremely distributed reviewing and integration process supported by 130 developer mailing lists and a hierarchy of dozens of Git repositories for version control. Since not every patch can make it and of those that do, some patches require a lot more reviewing and integration effort than others, developers, reviewers and integrators need support for estimating which patches are worthwhile to spend effort on and which ones do not stand a chance. This paper cross-links and analyzes eight years of patch reviews from the kernel mailing lists and committed patches from the Git repository to understand which patches are accepted and how long it takes those patches to get to the end user. We found that 33% of the patches makes it into a Linux release, and that most of them need 3 to 6 months for this. Furthermore, that patches developed by more experienced developers are more easily accepted and faster reviewed and integrated. Additionally, reviewing time is impacted by submission time, the number of affected subsystems by the patch and the number of requested reviewers.**

I. Introduction

Integration of code changes into a project's main repository is an open source developer's ultimate goal, since it marks the first step towards inclusion in an official product release. An open source project like the Linux kernel, for example, integrates between 8,000 and 12,000 patches in a new release, contributed by more than 1,000 developers [1]. Those patches only represent the "lucky few". Studies on Apache and other open source systems have shown how only 40% of the patches considered for integration eventually succeed [2], [3], [4].

One of the major reasons for the relatively low success rate of integration is the complexity of this process. The patches first need to pass a gate-keeper who performs a review of the code [2], [5], [6], before the code is merged by an integrator (e.g., release engineer) into the corresponding branch of the open source project [7], [8], [9]. Code reviews fail when a patch does not implement a relevant, working feature or bug fix, or when the project's development guidelines are not followed [10]. The actual integration (merging) fails when the patch interacts incorrectly with other patches or the merging process creates too many merge conflicts [11]. In case of integration issues, the developer needs to go back to the drawing board and try to integrate the code again. In the worst case, a patch will be rejected time and time again until the developer eventually gives up.

As a result, the integration process looks like a black box to most developers, with unpredictable outcome. Everyone knows the stories of disgruntled developers, even experienced ones, whose changes did not make it after putting months of work into them (e.g., [12], [13]). Even major projects like the Google Android mobile platform have problems getting their Linux kernel modifications integrated into the official kernel version [11]. Yet, determining up front whether a patch will make it, and how long it will take, is a grey area. Research on code reviews has shown how small patches [6], [4] sent by experienced developers [2] are more likely to be accepted by the reviewers, but it is not clear if these characteristics play the same role during the actual integration of the patch with other patches. Similarly, the impact of these characteristics on the time it takes to get a patch into a release is unclear.

This paper studies the relation of patch characteristics with (1) the probability of acceptance into an official release and (2) the time between submitting a patch for review and acceptance. We also analyze if these relations change over time. Our empirical analysis is based on eight years of patch review data and version control data from the Linux kernel project, which is a 20 year-old, popular open source system containing more than 15 MLOC of source code. We address the following research questions:

RQ1) What percentage of submitted patches has been integrated successfully, and how much time did it take?
Around 33% of patches are accepted. Reviewing time has been dropping down to 1–3 months, while integration time steadily has been increasing towards 1–3 months, bringing the total time to 3–6 months.

RQ2) What kind of patch is accepted more likely?
Developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance, while patch characteristics and submission time do not.

RQ3) What kind of patch is accepted faster?
Reviewing time is impacted by submission time, the number of affected subsystems, the number of suggested reviewers and developer experience, while integration time is impacted by the same attributes as patch acceptance.

First, we provide background about the Linux kernel integration process (Section II), followed by an explanation of our case study methodology (Section III). Section IV presents the results of our case study, followed by a discussion of threats to validity (Section V). We finish the paper with related work (Section VI) and the conclusion (Section VII).

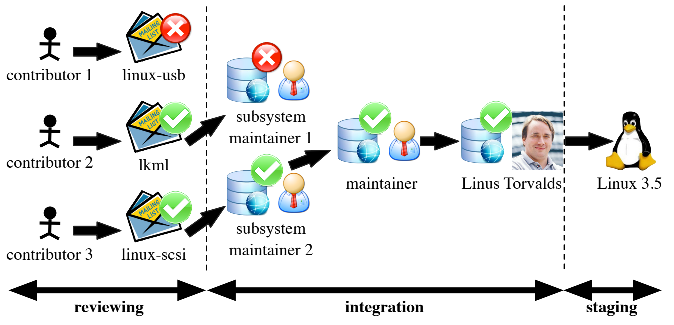


Fig. 1. Linux kernel development process. Contributor 1's patch is rejected during reviewing, and contributor 2's patch is rejected during integration, but contributor 3's patch makes it into the next Linux release (version 3.5).

II. Background

The Linux kernel open source project started out as a one-man project by Linus Torvalds in 1991, but quickly exploded into one of the hallmark projects of open source development. Since early on, kernel development is managed by a strict hierarchy of experienced kernel developers under the leadership of Linus Torvalds, who has the final decision about incorporating a source code patch into the kernel. Figure 1 illustrates the lifecycle of a kernel patch [10].

A developer first needs to send a request for comments (RFC) to a kernel subsystem's mailing list to get input on a new idea for a feature or bug fix. After fleshing out the design, the developer implements it and sends the resulting patch or patch set (series of collaborating patches) to the subsystem's mailing list and/or the global Linux Kernel Mailing List (LKML). Through email discussion, the subsystem maintainer and other experts review the patch. The developer then needs to incorporate any feedback and re-submit his patch (set), otherwise the patch does not get through the reviewing stage.

Once all reviewers are happy, the subsystem maintainer commits the final patch to his Git repository. Other maintainers, developers and beta-testers tracking the maintainer's repository now become aware of the patch and can provide additional reviews. This will also expose integration conflicts, i.e., other patches that break because of the new patch. Again, the developer needs to act on these conflicts to avoid not getting through integration. If the kernel maintainers are sufficiently confident about the patch, Linus Torvalds might consider incorporating it into the official Linux kernel release. This only happens during a release's "merge window", a period of roughly two weeks following the previous release. Afterwards, small bug fixes are still possible until the next release occurs (every 2 or 3 months [1]).

This paper uses the terminology of Figure 1 to denote the duration of each major phase. **Reviewing time** is the time from a developer's patch submission until the patch's commit by a subsystem maintainer. **Integration time** is the time from a patch's commit by a subsystem maintainer until its merge into Linus Torvalds' repository. Finally, **staging time** is the time from a patch's merge into Linus Torvalds' repository until the next kernel release.

III. Methodology

In order to address our three research questions, we studied the acceptance rate of patches submitted by email, as well as their reviewing, integration and staging time. We now explain the different steps used for our study.

A. Data Extraction

Since the Linux kernel integration activities (Figure 1) are spread out across mailing lists and Git repositories, we need to mine both data sources, then try to match the patches inside the emails to commits inside the repositories.

There is one global mailing list (LKML), and 130 more specialized kernel-related lists. These mailing lists are archived online [14] in the form of textual mbox files. We obtained access to the mbox files of 2005 until 2012, then used the MailMiner tool [15] to process these files into a relational database, with tables containing the email messages and their metadata. We did not analyze email attachments, since the Linux kernel developer guidelines dictate that all patches are inlined into the email body(i.e., attachments are not reviewed). Unfortunately, we were not able to find reliable heuristics that link threads related to (different versions of) the same patch to each other, since no official guidelines exist regarding the title of such email threads.

The source code repository data is better structured. Since the 2.6.12 kernel (June 26, 2005), the kernel uses the Git distributed source control system where each developer and maintainer has a copy of the whole project repository. Since we only need to analyze which patches make it into an official release, we only have to clone and mine Linus Torvalds' repository [16]. This repository contains all commit information of accepted patches from June 26, 2005 to December 31, 2012, including information like the original commit date by a subsystem maintainer and the merge date into Linus Torvalds' repository. Note that we do not have information about patches that made it to a maintainer's repository, but never to Linus Torvalds' repository (the second example in Figure 1).

B. Linking the Patches in Emails to Git Commits

To obtain information on which patches are accepted by Linus Torvalds, we had to link the patches in the mailing list emails to the commits in Linus Torvalds' Git repository. Similar to Bird et al. [17], we did not directly link a full patch to a Git commit, because of "cherry-picking". This is a common integration activity where an integrator only picks the interesting parts of a patch and ignores the rest. Large patches risk not being merged completely, or maybe not in one Git commit. Hence, we split a patch into one or more chunks, with each chunk containing all changes of the patch to one file. If a patch undergoes multiple versions in one email thread (we cannot track versions across threads), we tried to link each patch version to the Git repository.

The actual linking between chunks and commits is based on checksum matching. After splitting into chunks, we filter out the unchanged code lines of a submitted chunk, remove all white space and capitalization, then concatenate all lines into one line. After prepending the relative path name of the file changed by the chunk, we calculate the MD5 checksum. We perform the white space, capitalization and concatenation to deal with small changes done to a chunk before merging, and the path name prepending to avoid false positive matches with similar changes to other files. We perform the same procedure to the commits in the Git repository (after splitting them into chunks), then match the MD5 checksums of submitted chunks and Git chunks. We only link to the closest Git chunk in time that follows the submitted chunk. Overall, 47% of the chunks in the mailing lists could be mapped to Git commits.

We found that patch chunks, i.e., all changes in a patch to one particular file, are a good compromise between granularity and patch identity. We evaluated the recall of the chunk linking mechanism on a sample of 3,000 email threads from the linux-tips mailing list, which contains the actual Git commit identifier for an accepted patch. Our linking approach had a recall of around 75%. We then performed a random sampling of 100 emails (50% linked and 50% not linked by our linking mechanism) across all mailing lists, to obtain a confidence interval for the mechanism's precision, with length 10% and a 95% confidence level. We found that our technique had a precision of $100\% \pm 10\%$. These numbers provide us confidence that our linking mechanism is sufficiently accurate.

Finally, to map back from the chunk level to the patch level, we analyzed each patch's chunks and considered a patch to be accepted if at least one of its chunks was mapped to a Git commit. The patch's reviewing, integration and staging time are the corresponding times for the first chunk that was accepted by Linus Torvalds. This makes sense, since one accepted chunk is enough for a developer to know that his work will appear (at least partially) in the next release.Using the last accepted chunk would also lead to more outliers. Of the 348,184 Git commits in Linus Torvalds' repository, 256,284 were actual patch submissions (i.e., non-merge commits authored by other people than the committer), and we could map 190,931 of those commits (74.5%) to a patch submission email.

C. Measuring Patch Characteristics

In order to study the relation between patch characteristics and (time to) acceptance, we need to define a list of relevant characteristics. Initially, we based ourselves on a list of guidelines published by the Linux Foundation to support kernel developers in getting their patches accepted [10]. Some of these guidelines, such as whether or not a developer has used the "checkpatch.pl" tool before submitting her patch to a mailing list, could not be measured easily. Others, such as whether the patch is small enough or was sent to the right mailing list, were straightforward to measure. We extended this set of metrics with additional metrics, such as whether a patch fixes a bug or proposes a new feature.

Table I shows all metrics that we used during our study, including the type of metric, the data source from which we calculated it and a short description. They are grouped into five dimensions, and an additional group of dependent metrics. Most of the metrics are straightforward to understand, hence we only discuss some of them in more detail. We consider a review to be each email in a thread that replies to an email with a patch, but only until the next patch in that thread or the end of the thread.

Developers sometimes want one or more specific persons to review their patches, often the maintainer of a subsystem. To (try to) achieve this, they add those persons' email addresses as the CC addressees of the email (in order of importance). We measure the number of CC-ed people, and also check whether the first CC-ed reviewer indeed committed chunks of the patch.

Since the time in between Linux releases varies between 2 and 3 months, the merge window can also vary in time. As Linus Torvalds determines by himself when the merge window ends, we could not identify the exact dates. Instead, we divided the period in between two subsequent releases into 4 periods ("quarters"), and report for each patch the `rel_quarter` in which the patch was submitted to a mailing list for review.

To identify the most specific mailing list for a particular patch, we analyzed all patches of each mailing list, and counted the number of patches modifying each subsystem. We assigned the most changed subsystem to each mailing list. A subsystem here corresponds to a subdirectory in the kernel code base, such as "kernel" and "net/ipv4". We did not go deeper than 2 levels of subdirectories, since otherwise the subdirectories became too specific.

To tag a patch as being a bug fix, we used a traditional, simple heuristic: We searched in the patch's log message for the words "bug" and/or "fix" (case-insensitive). More advanced techniques exist [18], and we plan to experiment with these in future work.

Furthermore, to identify if a patch is part of a patch set, we used the naming convention suggested by the Linux Foundation [10]. Messages that are part of a patch set should contain the term "PATCH" (case-insensitive) and a marker of the form "5/20", which means that this email is patch 5 out of 20. We check all subsequent threads of the same author within one hour to see if other messages contain "PATCH" and "20". Using a 1-hour window for each email suffices since manual browsing of patches showed that patch set emails follow each other very closely. We cannot check for threads with exactly the same subject, since different patches in a patch set typically have specific subjects for that patch. If all patches of a patch set end up in the same thread, we consider the reviews in the thread to be shared by all patches in the set.

Finally, we only calculate the reviewing, integration and staging time for fully accepted patches. Patches that failed reviewing or went through reviewing, but not through integration, do not have any of these three metrics. For the patches that made it to Linus Torvalds, we measured the shortest time from the initial maintainer commit until a merge commit performed by Torvalds that brings the patch into his repository.

D. Data Analysis

To address the three research questions, we performed empirical analysis on the metrics that we collected, and also built decision tree models. A decision tree is a tree where

TABLE I

Overview of the metrics and dimensions used. The superscript after the name is the research question in which it was used.

	metric name	type	source	explanation
Experience	msg_exp^{2,3}	numeric	mbox	Number of patches sent by author in earlier threads.
	commit_exp^{2,3}	numeric	git	Number of accepted Git commits thus far by the author.
Email	year/month/week/day^{1,3}	nominal	email	Year/Month/Week of the year/Day of the week on which patch was sent.
	nr_ccs^{2,3}	numeric	email	Number of people CCed by email.
	msg_length^{2,3}	numeric	email	Number of lines of email text excluding patch lines.
	rel_quarter^{2,3}	nominal	email	Quarter of release window in which patch is submitted.
	thr_first^{2,3}	boolean	thread	Is this email the first one of the current thread?
	first_patch^{2,3}	boolean	thread	Is this first patch in thread?
	thr_volume^{2,3}	numeric	thread	Number of email messages between start of current thread and current patch.
	thr_part^{2,3}	numeric	thread	Number of people participating in current thread until current patch.
	thr_time^{2,3}	numeric	thread	Discussion time (seconds) between start of current thread and patch.
	right_venue^{2,3}	boolean	patch	Is the patch sent to the most specific mailing list for the changed subsystem?
	lkml_first^{2,3}	boolean	thread	Was the first email sent to the general-purpose LKML list?
Review	nr_reviewers^{2,3}	numeric	thread	Number of different people sending review messages.
	nr_reviews^{2,3}	numeric	thread	Number of review messages.
	response_time^{2,3}	numeric	thread	Time in seconds from patch to first review message.
	first_response_time^{2,3}	thread	patch	Time in seconds from first patch in thread to first review message.
Patch	bug_fix^{2,3}	boolean	patch	Is patch a bug fix?
	chunks_in^{3}	numeric	git	#chunks in patch accepted by Linus Torvalds.
	chunks_out^{3}	numeric	git	#chunks in patch rejected by Linus Torvalds.
	size^{2,3}	numeric	patch	Patch churn (sum of added and removed lines).
	spread^{2,3}	numeric	patch	Number of files changed by patch.
	spread_subsys^{3}	numeric	patch	Number of subsystems changed by patch.
	nth_try^{2,3}	numeric	thread	What version of this patch are we (relative to this thread)?
	commit_sub^{2,3}	numeric	git	Number of previously accepted patches modifying the changed subsystem (prior churn).
	patch_set^{2,3}	boolean	email	Is this patch part of a larger patch set?
Commit	committer^{3}	numeric	git	Number of different committers for the chunks of this patch.
	cc_is_rev1^{3}	boolean	git	Is name of committer same as name of first CC-ed reviewer?
	cc_is_rev^{3}	numeric	git	Number of chunks where the name of committer is same as name of first CC-ed reviewer.
Dependent	**accepted**^{1,2}	boolean	git	Was this patch accepted by Linus Torvalds?
	reviewing_time^{1,3}	numeric	email/git	Time between patch submission and first commit by a kernel maintainer.
	integration_time^{1,3}	numeric	git	Time between first commit by a kernel maintainer to acceptance by Linus Torvalds.
	staging_time^{1,3}	numeric	git	Time between acceptance by Linus Torvalds and the next release.
	total_time^{1,3}	numeric	email/git	Time between patch submission and the release in which the patch appeared.
	missed^{2}	numeric	email	Number of missed kernel releases since patch submission for an accepted patch.

every node is a condition such as "size>50?" and one takes a different path based on the evaluation of the condition for a particular patch. The leaves of the tree correspond to a classification such as "accept" or "reject". We provide more explanation when discussing the individual research questions.

In order to improve the performance of the models and to make the values easier to interpret, we discretize the reviewing and integration time variables. For example, it does not really make a difference whether a patch will take 13 or 14 days to review, but it does make a huge difference if a patch will take a month instead of a day. After analyzing histograms of the data, we decided to discretize the time data into the following bins: "instantly", "within_hour", "within_day", "within_week", "within_month", "within_quarter", "within_half_year", "within_year" and "took_ages".

IV. Case Study Result

This section presents the results of our three research questions. For each question, we present its motivation, the analysis approach and a discussion of our findings.

RQ1: What percentage of submitted patches has been integrated successfully, and how much time did it take?

Motivation: Existing reports about Linux kernel development show that in between 8,000 and 12,000 patches are accepted into each current kernel release, authored by more than 1,000 developers [1]. Although this illustrates the huge scale of development for the Linux kernel, not much is known about the success rate of patches submitted to the kernel. How many patches are actually submitted, and how many eventually make it? Of those who make it, how much time do they

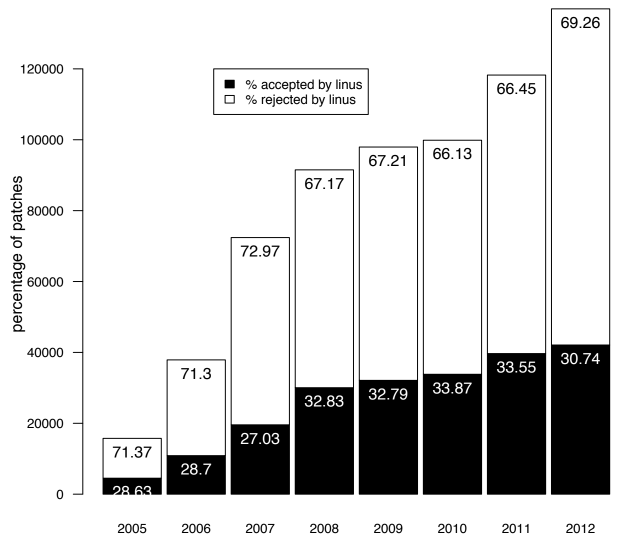


Fig. 2. Number of accepted patches. The numbers in the bars correspond to the percentage of accepted or rejected patches.

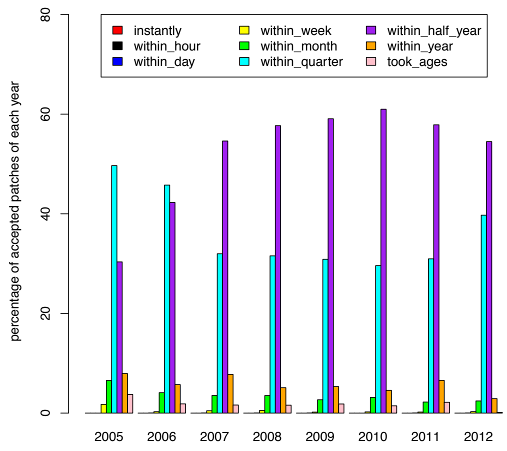


Fig. 3. Total time for a patch to occur in an official release.

need until they are integrated into the next kernel releases? How much of that time is spent on reviewing compared to integration effort? The answers to these questions provide insight into the process, output, productivity and effectiveness of collaborative open source development at massive scale in the Linux project.

Approach: We compute the number of accepted and rejected patches that were sent to a kernel mailing list between 2005 and 2012 and measure the total time it took for the accepted patches to end up in a release, as well as the reviewing, integration and staging time.

Findings: **The yearly number of submitted patches to kernel mailing lists keeps on increasing.** Figure 2 shows that, starting in 2005, the number of submitted patches discussed on the mailing list has kept on increasing, with a temporary slowdown from 2008 to 2010. The numbers are staggering: In 2013 alone 136,932 patches were submitted to Linux mailing lists. Even though the number of submitted patches has increased, the percentage of accepted patches has remained between 27 and 34% (see Figure 2). The rest (more than 66%) have not made it, but it is hard to know why they were refused. This might be due to some patches being reworked and resubmitted, others still being discussed for consideration, and some might simply be ignored and never applied. Therefore, it is important to know what is the expected time it takes to have a patch incorporated into the kernel.

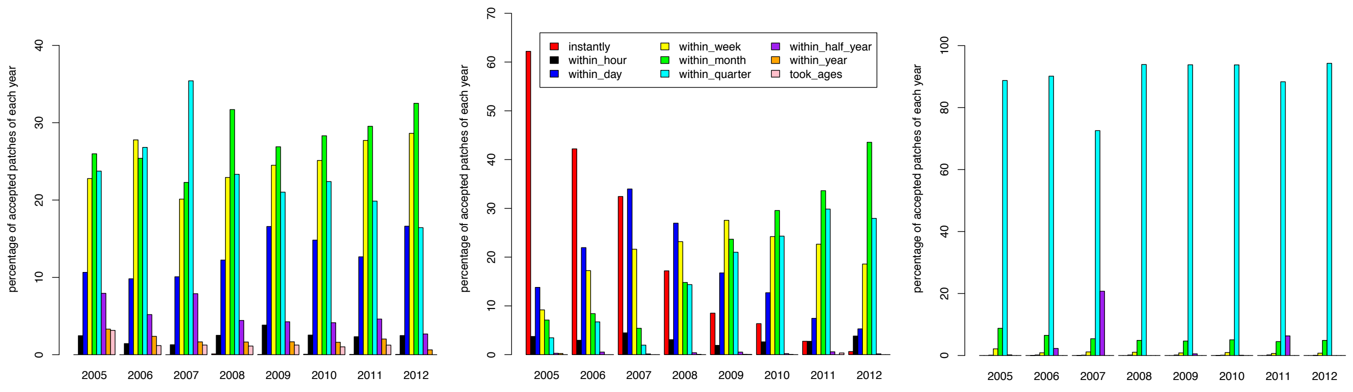
Patches take 3-to-6 months towards inclusion in a kernel release. With respect to the total time it takes a patch to be released as part of the kernel, Figure 3 tells us that in 2005, most of the accepted patches took between 1-to-3 months. Afterwards, the time most patches took grew to 3-to-6 months, remaining relatively stable from 2007–2012. To better understand this, we have decomposed the total time into reviewing, integration and staging time, and analyzed each of them separately.

The distribution of reviewing time has become shorter (Figure 4(a)), with more patches being reviewed within a day and a shift from 1-to-3 months to within 1-week-to-1-months. At the same time, less patches take longer than one month to be reviewed. A non-parametric Kruskal-Wallis test, followed by pairwise Mann-Whitney tests with a confidence level of 0.05 (using Bonferroni correction) confirmed the changes in reviewing time. This suggests that Linux has improved its reviewing process over time. Either reviewers became more efficient, or the patches became easier to review.

On the other hand, the time of integration has slowed down. Figure 4(b) shows that in 2005 almost 60% of patches were committed directly (“instantly”) to the Linux kernel by Linus Torvalds and almost 80% made it within a day, while in 2012 only about 10% were integrated within a day. There is a clear increase of the percentage of patches taking a month or even a quarter to be integrated. Again, a non-parametric Kruskal-Wallis test, followed by pairwise Mann-Whitney tests with a confidence level of 0.05 (using Bonferroni correction) confirmed these changes in integration time.

These observations can only be explained by Linus Torvalds applying less and less patches over time. We tested this assumption by computing the proportion of commits authored by other developers but directly committed by Linus Torvalds relative to all non-merge commits. Starting in 2005, he has committed yearly 39%, 30%, 21%, 11%, 7%, 5%, 4% and 4% of all non-merge commits. In 2005, 4,508 patches were accepted for which Linus made 5,623 commits authored by others, while in 2012, there were 42,088 patches accepted and Linus committed others’ work only 2,370 times. Similarly, the number of other committers who commit changes authored by another person has grown from 64 to 261. This suggests that Linus is integrating patches less (in addition to developing less himself), delegating this task to others in order to cope with the overall growth in patches. Similar observations were made elsewhere [1].

To understand whether the speed-up of reviewing time and slow-down of integration time is due to changes in development style, we analyzed characteristics of the patches that were reviewed and/or integrated. We found that the size of patches has changed, dropping initially from a median size of 61 lines of code in 2005 to 29 lines in 2007, then increasing again up



(a) Reviewing time (b) Integration time (c) Staging time

Fig. 4. Distributions of times for patches

to a median of 77 in 2012 (87 in 2011). These larger patches are affecting more files as well, since the median number of changed files increased from 3 to 4. Finally, the percentage of bug fix patches has dropped continuously from 24% to 16%, i.e., more patches contain feature enhancements instead of bug fixes. This provides an explanation of the growing size of patches as well as the longer time it takes to integrate these more complex patches. It is not clear why reviewing is not affected by this.

Staging time is fixed at 1–3 months. Figure 4(c) shows how most of the patches got in within one quarter. Indeed, the time in between releases has been set to a fixed number of weeks by Linus Torvalds, i.e., it is a management decision. In addition, patches are only merged into Linus Torvalds' repository roughly the first two weeks after the previous releases. Commits sent outside this merge window are ignored until the next release's merge window. Only in 2007, many patches took more than half a year to be accepted. Since staging time is dictated, we do not consider it in this paper.

The reviewing time is becoming shorter and shorter, seemingly at the expense of a longer integration time. However, the total acceptance time does not change too much.

RQ2: What kind of patch is accepted more likely?

Motivation: In RQ1, we found that the accepted patches represent only around 30% of all submitted patches. Furthermore, it takes several months before knowing that a patch has passed review, and even more time before knowing that it is accepted by Linus Torvalds. Hence, in the worst case, developers can lose a lot of time working on and maintaining a patch that will never make it. Even worse, they could find themselves having to choose between different bugs to fix or features to implement. The same problem applies to maintainers and integrators, who are buried under a large load of incoming patches and need to separate the wheat from the chaff. Thus, RQ2 examines the characteristics of successful patches.

Approach: As we saw in RQ2 that Linux has a release every 2 or 3 months [1], i.e., roughly once every quarter, we build decision trees for every 3 months of development. In order to understand the characteristics of accepted patches, we build

TABLE II
SIGNIFICANT ATTRIBUTES FOR PATCH ACCEPTANCE.

Attribute	Influence
`msg_exp`	Smaller value leads to acceptance.
`commit_exp`	Larger value leads to acceptance.
`thr_part`	Larger value lead to acceptance.
`nth_try`	Larger value leads to acceptance.
`commit_sub`	Smaller value leads to acceptance.

decision tree models with the metrics in Table I as independent variables and `accepted` as dependent variable. This results into 32 decision tree models from 2005 to 2012.

For each model, we use 10-fold cross-validation to obtain more accurate performance measures. The data basically is split into 10 folds, and we use each fold once as test set with the other folds as training set. This generates 10 trees for each quarter. We use precision (how many patches classified as "accepted" really were "accepted"?) and recall (of all "accepted" patches, how many did we classify as "accepted"?) to measure the performance of the decision trees. As baseline to compare the performance to, we use a zeroR model, i.e., a model that always predicts "accepted". For example, if there are 13% accepted patches, then a zeroR model would have a prediction and recall of 13%.

Then, we analyze the most influential metrics in the models using top node analysis, i.e., we look up the metrics that occur in the conditions of the top 2 levels of each decision tree, since they contain the most decisive information regarding patch acceptance. To visualize the results of top node analysis across all quarters, we use heat maps. These are plots with time as the X axis and the metrics as the Y axis. For a particular quarter and metric a cell is non-white if that metric was a top node in at least one of the 10 trees of that quarter. The darker the cell (light grey up to black), the more trees contained the metric in their top 2 levels.

Findings: **We are able to build high-performing models with up to 73% precision and recall**. The precision values of our models range from 52% to 73% with an average of 66.28%, compared to 40.41% for the zeroR models (between 17% and 48%). Similarly, for recall we see that the recall values of our models range from 63% to 73% with an average of 68.47%, compared to 40.41% for the zeroR models.

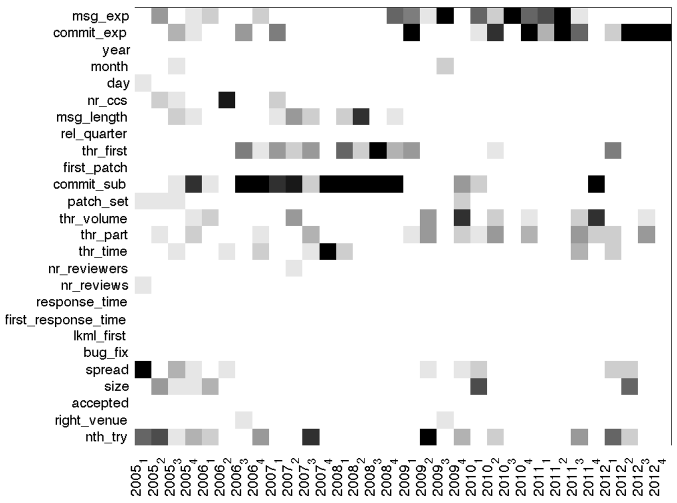


Fig. 5. Top node analysis results for the patch acceptance models (RQ2).

Developer experience, patch maturity and prior subsystem churn are the major factors impacting acceptance of a patch. Figure 5 shows the top node analysis results of the decision trees for patch acceptance, while Table II lists the 5 most impacting metrics overall with their interpretation.

The top two metrics (`msg_exp` and `commit_exp`) relate to developer experience. Developers who have had a commit being merged into Linus Torvalds' repository before have a higher chance of getting their patch accepted, presumably since they have more experience with the creation and development process of kernel patches, and maybe a higher standing in the community. Surprisingly, having posted more emails (and hence reviews) to mailing lists before has a negative impact on acceptance. It is not clear why this is the case. We hypothesize that those experts might be working on more complex features, which are more risky and prone to rejection. However, more work is needed to verify this hypothesis.

The third and fourth most important metrics are related to patch maturity, i.e., the amount of reviewing discussion preceding a patch submission (`thr_part`) and the number of previous iterations of the patch in the same thread (`nth_try`). Unsurprisingly, the more mature a patch, the higher the probability of acceptance.

The fifth most important metric relates to the number of previous commits to the changed subsystem, i.e., the subsystem's prior churn (`commit_sub`). According to the data, the more popular a subsystem is in the Git commits, the lower the probability that a new patch will make it. It is not clear what exactly this implies. It might refer to the fact that subsystems that see lots of change are harder to keep up with, causing new patches to be outdated by the time they are proposed. Alternatively, it could mean that there is a lot of duplicate work going on, with multiple developers trying to compete for the same features. More analysis is needed to fully understand the impact of this observation.

Among all the above attributes, only the top 2 experience attributes can be controlled by developers by participating more actively in the kernel community. This helps them learn more about kernel development, and to earn the trust of other developers and maintainers. The other 3 attributes are hard to manipulate for a developer.

The major metrics vary over time. We can see how no metric is important in all 8 studied years. Instead, there seem to be roughly three phases in Figure 5, i.e., 2005/2006, 2007/2008 and 2009–2012. Phases one and three especially value metrics on patch maturity, patch characteristics, review activity and experience, whereas phase two especially focuses on review activity. We suspect the change in phase two at the end of 2008 to coincide with the introduction in February 2008 of the linux-next integration repository. This repository was introduced to test how a patch would merge with the current version of the Linux kernel, in order to avoid that patches only are merged six months later by Linus and fail at that time. As such, linux-next aimed to increase acceptance rate and reduce integration time through faster feedback.

Patch characteristics, time of submission and review response time do not play a major role in patch acceptance. Looking at the white areas in Figure 5, we can see that patch characteristics like size, spread and whether a patch is sent in one piece or has been split into smaller parts (patch set) only played a role initially. This contradicts existing work [6], [4], as well as the Linux Foundation's guidelines [10]. The time in a release quarter (`rel_quarter`) does not play a role either, and neither does the quantity and responsiveness of reviews.

The most important metrics for patch acceptance are changing all the time. We find that the patches of experienced developers, mature patches and patches in subsystems with less prior churn are easier to be accepted.

RQ3: What kind of patch is accepted faster?

Motivation: Similar to the motivation for RQ2, we find that some patches took much longer than others to arrive to Linus Torvalds' repository. For developers, maintainers and integrators, it is important to understand which patch properties determine reviewing and integration time. Hence, we will analyze only patches that eventually were merged by Linus Torvalds.

Approach: Similar to RQ2, we build decision tree models for each quarter. This time the dependent variable is either the discretized reviewing time or integration time. We again use 10-fold cross-validation for the reviewing time and integration time models, followed by top node analysis. Instead of precision and recall for each of the 9 outcomes, we compute the global accuracy, i.e., the percentage of correctly classified patches (across the 9 outcomes) relative to all patches.

Findings: **The reviewing and integration time models obtain an accuracy of up to 70% and 76%, resp., and again evolve over time.** The accuracy values for the reviewing time model fluctuate between 53% and 70% (mean of 59%), while for the integration time they fluctuate between 62% and 76% (mean of 68.4%). Although the models evolve over time, we observe more noise in the top node heat maps (Figure 6 and Figure 7) compared to the acceptance models. It is hard to spot

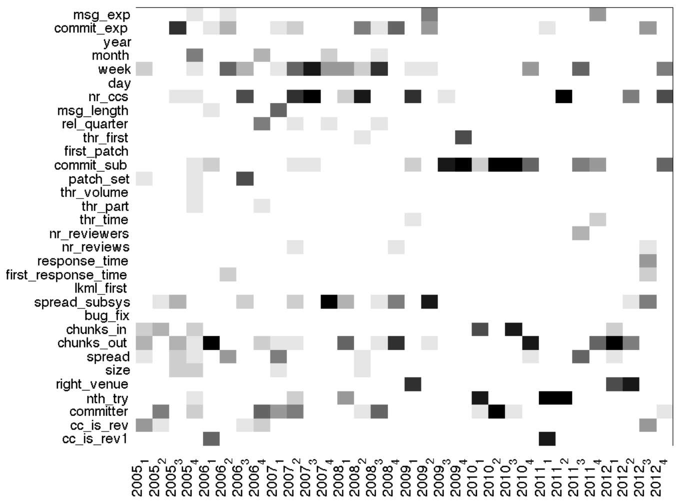


Fig. 6. Top node analysis results for reviewing time.

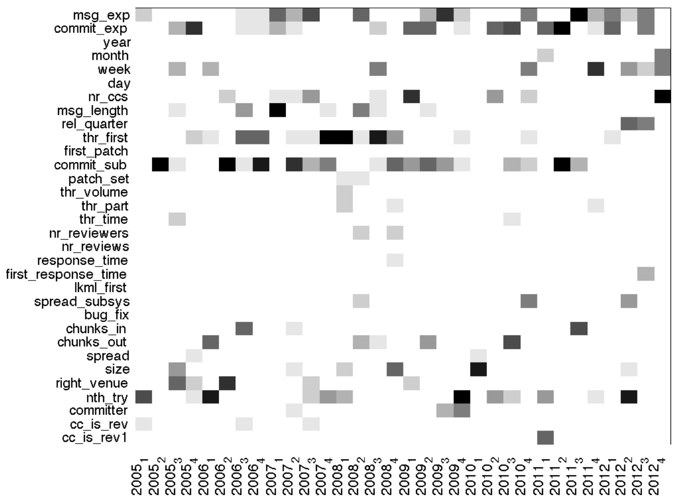


Fig. 7. Top node analysis results for integration time.

different phases, except for the reviewing time models, where the experience-based metrics at the top stop being influential. It is unclear why this is the case (linux-next only applies to integration, not to reviewing).

For reviewing time, a variety of dimensions is influential. Figure 6 and Table III show the following top 6 influential dimensions: Prior subsystem churn, time, patch characteristics, reviewers and experience. Contrary to the model in RQ2, most of these attributes (except the prior subsystem churn) can be directly influenced by a developer. Prior subsystem churn plays a rather complicated role, changing its impact across different quarters. Since bugs need to be fixed irrespective of a subsystem, one cannot really tweak this attribute. On the contrary, the week of the year in which one submits a patch for review plays a significant role. This is tied to the merge window of a release, i.e., the period in which the new submissions are solicited. If timed properly, a submission of a patch will arrive in time for a merge window, otherwise it might be delayed at least an entire cycle until the next release merge.

TABLE III
SIGNIFICANT ATTRIBUTES FOR REVIEWING TIME.

Attribute	Influence
`commit_sub`	Impact varies a lot across time, but in recent years larger values lead to shorter time.
`week`	Larger value leads to longer reviewing time.
`chunks_out`	Larger value leads to longer reviewing time.
`spread_subsys`	Larger value leads to longer reviewing time.
`committer`	As the reviewing time increases, the number of committers first increases, then decreases.
`nr_ccs`	Larger value leads to longer reviewing time.
`commit_exp`	Larger value leads to shorter reviewing time.

TABLE IV
SIGNIFICANT ATTRIBUTES FOR INTEGRATION TIME.

Attribute	Influence
`msg_exp`	Larger value leads to shorter integration time.
`commit_exp`	Larger value leads to shorter integration time.
`thr_first`	"true" value leads to shorter integration time.
`nth_try`	In early years, having more iterations leads to shorter integration time, but recently the opposite holds.
`commit_sub`	Larger value leads to shorter integration time.

Various patch characteristics, such as the `spread` of a patch and the number of chunks in it play an important role. In fact `chunks_out` represents the number of chunks that were not committed in the end, i.e., the chunks that were ruled out during reviewing or integration. Deciding about this indeed takes up quite some reviewing time. This brings us to the reviewing dimension (`committer` and `nr_ccs`). `committer` corresponds to the number of different people committing (and hence probably involved in reviewing) chunks of a patch, whereas `nr_ccs` refers to the number of people suggested as reviewer by putting them into the list of cc-ed email addresses. Hence, the choice of reviewers indeed plays an important role in reviewing time [2]. Finally, experience again plays a major role.

Surprisingly, some metrics that do not play a role are whether a patch fixed a bug (we expected bugs to be expedited), nor whether the patch had been posted to a specific mailing list for a subsystem vs. the global LKML. This finding contradicts the Linux Foundation guidelines [10].

For integration time, the same dimensions are influential as for probability of acceptance. In Table IV, the only difference is that a high `msg_exp` this time leads to shorter integration, whereas it decreased the probability of acceptance (RQ2). Furthermore, experience plays a much larger role for integration than for reviewing. The fact that the models for acceptance and integration value the same attributes suggests that integration plays an essential role in patch acceptance, maybe more than reviewing by itself.

Developers can reduce reviewing time by controlling the submission time, the number of affected subsystems by the patch, the number of reviewers CCed on the mailing list submission, and by being more active in the community. Gaining experience through active participation in the open source community is also linked to shorter integration time.

V. Threats to Validity

We now discuss the threats to validity of our study, following common guidelines for empirical studies [19].

Construct validity threats concern the relation between theory and observation. We use decision trees to find important relations between a rich set of metrics and patch acceptance (time). Since these models are not perfect (precision, recall and accuracy below 100%) and are statistical in nature, they might not correspond to actual relations or actions in the development process. Although qualitative analyses are needed to validate these models, we built many models across time and observed various recurring phenomena, giving us confidence that the models do provide major indications about the relations. Furthermore, our selection of metrics is based on guidelines provided by the Linux Foundation [10], enhanced with other patch-related metrics.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The automatic mapping between email patches and Git commits contains many different steps that might introduce noise. First, the mailing list archives typically contain incomplete information (such as emails that are not clearly linked to a thread, which even specialized tools cannot resolve perfectly [15]. Second, the chunk-based linking approach cannot handle changes to identifier names (contrary to Bird et al. [17]). Also, even though the chunk-level is sufficiently coarse-grained, it is still possible that similar changes are made to the same file in a short timespan, for example on different clones in a file. However, our linking approach obtained a high precision and recall when evaluated on a sample of the data. Third, Git does not retain information about the name of the repository or branch where a commit originated, and it even allows developers to change (clean up) the history of changes in order to make later integration easier [20]. This might affect the heuristics and other techniques used to calculate the integration time of a commit.

Threats to external validity concern the possibility to generalize our results. Since we have only studied one large open source system, we cannot generalize our findings to other open and closed source projects, even in the same domain. Furthermore, since Git is a flexible version control system that can be used in various setups, we also need to take care when extrapolating our findings to other projects using Git. Hence, more case studies on other projects are needed.

VI. Related Work

Our work is related to previous studies on reviewing processes in open source projects, on software integration and on prediction of bug fixing time.

Rigby et al. [6] studied reviewing practices in the Apache open source system, and compared these to those of a commercial system. Between January 1997 and October 2005, the Apache mailing list contained 9,216 review email messages for 2,603 patches. They found that small, independent, complete patches are most successful, whereas we found that size only plays a role for reviewing time, whereas patch sets do not play a major role. 50% of the patches are reviewed in less than 19 hours, whereas for Linux this takes 1–3 months. Finally, Rigby et al. found that 44% of the submitted patches eventually were accepted, compared to 33% in our study.

Weissgerber et al. [4] studied contributions in two small open source systems (196 and 1,628 patches respectively, changing 6% of all files in the systems), and also found that around 40% of the contributions are accepted. Similar to Rigby et al., they find that smaller patches have a higher probability of being accepted. Most of the patches are accepted in the repository within one week (61% even within three days). Since they looked at CVS repositories, a commit automatically corresponds to a chunk (one file change). These chunks were mapped to mailing list patches by looking for the changed lines in the files (irrespective of their order), whereas we concatenated all changed lines and removed all whitespace and capitalization to improve performance. One quarter of the accepted patches took less than one day for reviewing and integration, half were accepted within one week, and one third took longer than two weeks. No relation between patch size and the time towards acceptance could be found. Similarly, we only found a link between patch size and reviewing time, but not with integration time.

Baysal et al. [2] studied reviewing effort in the Mozilla project. Contrary to the previous two studies and our study, Mozilla uses a modern web-based system (Bugzilla) to manage review requests, comments and their outcome. On average, 78% of the reviewed patches made it through reviewing, but only around 60% of those (i.e., 47% in total) eventually make it into the code repository, which is slightly higher than the other studies. Similar to us, ore contributors have a higher probability of getting their patches accepted in the code repository, although experience does not impact the outcome of the reviewing phase (it does impact the reviewing time in our case). Similar to Linux, people can suggest reviewers and this choice plays an important role to avoid having a patch end up to be rejected. Mozilla patches that make it into the code base took 4.5 days before the switch to rapid release, and 2.7 now for core developers (slightly less time for casual developers). For the Linux kernel, this can take up 3–6 months.

Our work differs from the work above in multiple ways. First off, we studied the whole integration process, i.e., not just reviewing. Furthermore, we studied the relation between more than thirty metrics and patch acceptance (time) and how this relation evolves over time. Finally, Linux has a much deeper hierarchical structure of contributors and maintainers, whereas projects like Apache and Mozilla are relatively flat. As a consequence, the time to get through reviewing or merging is much shorter than is the case for Linux, and the impact of merging becomes more important.

Regarding software integration, various researchers have studied the impact of branch structure (e.g., the hierarchical structure of source code repositories in Linux) on software quality. Bird et al. [7] proposed a what-if analysis that allows to identify harmful, redundant branches in order to simplify the branching structure. Such simplification can avoid merge conflicts (incompatible changes in the two versions of the code base), which inflate integration time. In their case study on a large commercial system, such a simplification was able to reduce integration time by up to 9 days. Shihab et al. [9] found how a mismatch between the branching structure and a company's organizational structure can increase post-release failure rate. This is why Linux kernel development uses a distributed version control system.

Brun et al. [8] studied different kinds of merging conflicts, and identified four major reasons why integration can be risky: (1) The integrator did not develop the code herself, (2) the patches to merge often contain too many changes in one big lump, (3) merging typically happens a relatively long time after the actual development, and (4) most of the serious conflicts are due to semantical instead of textual issues. We indeed found evidence for these four reasons in Linus.

Our models to explain the reviewing and integration time are related to the work on bug fix time prediction, where, based on a bug report, the time for fixing this bug report is predicted. Guo et al. [21] built models to predict which Windows 7 bugs would be fixed. The precision and recall of those models were around 65%. They found that having more people following the status of a bug improves the probability of the bug being fixed. Similarly, having submitted bugs before that ended up being fixed improves the chances of getting a bug fixed in the future, i.e., experience plays an important role. We made similar observations for our models of acceptance (time).

Giger et al. [22] found that post-submission data of bug reports such as the number of comments or number of interested people improves the accuracy of bug fix time prediction models. However, in our models, adding the number of reviewers and the number of reviews did not make a significant change. Finally, Zhang et al. [23] build a model to predict the time between assignment of a bug report and the time when bug fixing starts. This corresponds to the time between sending an email talking about a new idea for a feature and sending an email with the patch implementing the idea. However, since related email threads are hard to link to each other, we were not able to build models for this incubation time.

VII. Conclusion

Given the complexity of code integration, and its opaque nature for most developers, we studied the characteristics of patches that could explain patch acceptance and reviewing/integration time. We performed a study on the Linux kernel mailing lists and Git repository, and found that reviewing and integration are two relatively independent processes. Developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance and integration time, while submission time, the number of affected subsystems, the number of contacted reviewers and developer experience correlate the most with reviewing time. The one common thing in the three kinds of models is developer experience, i.e., there seems to be no substitute for active participation in an open source project while learning the project's ins and outs.

Acknowledgments

We thank Amatul Mohosina for her initial work on this topic, and Richard Ellis for kindly providing us access to the Linux kernel mailing list mbox files.

References

[1] J. Corbet, G. Kroah-Hartman, and A. McPherson, "Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it," http://go.linuxfoundation.org/who-writes-linux-2012, April 2012.

[2] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A firefox case study," in *Proc. of the 19th Working Conf. on Reverse Engineering (WCRE)*, 2012, pp. 447–455.

[3] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," University of Victoria, Tech. Rep. DCS-305-IR, January 2006.

[4] P. Weissgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proc. of the intl. working conf. on Mining Software Repositories (MSR)*, 2008, pp. 67–76.

[5] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *Proc. of the 22nd Intl. Conf. on Software Engineering (ICSE)*, 2000, pp. 263–272.

[6] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *ICSE '08: Proc. of the 30th Int. Conf. on Soft. Eng.*, 2008, pp. 541–550.

[7] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proc. of the ACM SIGSOFT 20th intl. symp. on the Foundations of Software Engineering (FSE)*, 2012, pp. 45:1–45:11.

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. of Foundations of Software Engineering (FSE)*, 2011, pp. 168–178.

[9] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proc. of the Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 301–310.

[10] J. Corbet, "How to participate in the linux community," http://ldn.linuxfoundation.org/book/how-participate-linux-community, July 2008.

[11] G. Kroah-Hartman, "Android and the linux kernel community," http://www.kroah.com/log/linux/android-kernel-problems.html, Feb 2010.

[12] J. Andrews, "Linux: Cml2, esr & the lkml," http://kerneltrap.org/node/17, February 2002.

[13] A. Mills, "Why i quit: kernel developer con kolivas," http://apcmag.com/why_i_quit_kernel_developer_con_kolivas.htm, July 2007.

[14] R. Ellis, "http://www.spinics.net/lists/," last accessed in January 2012.

[15] N. Bettenburg, E. Shihab, and A. E. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," in *Proc. of the 25th IEEE Intl. Conf. on Software Maintenance (ICSM)*, 2009, pp. 539–542.

[16] L. Torvalds, "git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git," last accessed in January 2012.

[17] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in oss projects," in *Proc. of the 4th Int. Workshop on Mining Software Repositories (MSR)*, 2007, p. 26.

[18] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Proc. of the 2012 Intl. Conf. on Soft. Eng. (ICSE)*, 2012, pp. 386–396.

[19] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[20] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proc. of the 6th Intl. Working Conf. on Mining Software Repositories (MSR)*, 2009, pp. 1–10.

[21] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proc. of the 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE) - Volume 1*, 2010, pp. 495–504.

[22] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proc. of the 2nd intl. workshop on Recommendation Systems for Software Engineering (RSSE)*, 2010, pp. 52–56.

[23] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," in *Proc. of the 19th Working Conf. on Reverse Engineering (WCRE)*, 2012, pp. 225–234.