Analyzing long-term maintenance releases of the Linux kernel

Abstract—Context: The LTS (long-term support) releases of the Linux kernel are used in most production deployments. These releases are frequently updated while they are under active maintenance (which is several years), to fix known bugs. Even when these releases are considered as "stable", they get several thousands of commits every year. Despite the industrial importance of the maintenance activity leading to all these changes, this activity has not been studied in depth.

Goal: To characterize source code changing activity in these LTS releases, so that the maintenance processes of the Linux kernel can be better understood.

Method: To mine the git repositories where commits for LTS releases are collected, and analyze them from three different perspectives: (1) find out a lower bound for how many of the commits are bug fixes, and how many bugs were present when maintenance started, by using developer tags; (2) determine maintenance activity in the different areas of the kernel source code, by analyzing which files are modified in commits to LTS releases; and (3) find out how long it took for source code changes to reach LTS releases since they were produced.

Results: (1) At least around 50% of the commits in LTS releases are for fixing bugs, and the first release in each LTS branch includes in the order of 10,000 bugs. (2) The maintenance activity in different areas of the kernel is diverse, but some of those areas (such as drivers) are more represented than others. (3) The time lag since changes are produced to when they are ported to LTS releases is usually in the range of tens of days, although it can be of several years.

Conclusion: We offer a perspective of the activity in Linux LTS releases, which helps to understand the maintenance activities of the project, and the balances that maintainers have to uphold. Additionally, our research provides insights into the measurement and analysis of these releases, which can be useful for deployers of Linux in production.

Index Terms—Linux, bug, bug fixing, mining software repositories, software maintenance, software development processes, long-term support.

I. Introduction

The Linux kernel is one of the most well-known FOSS (free, open source software) projects. It undergoes continuous development, with major releases approximately every three months, followed by frequent maintenance releases. These releases are the result of changes to the source code implementing new features, or aimed at maintenance. This is common in many modern FOSS and non-FOSS projects, which are no longer structured with an initial development phase, and then a maintenance phase while the software is in production, which does not add new features. Instead, these projects perform development and maintenance activities in parallel, even if there are periods in which including new features is most frequent, and other periods in which merging maintenance changes is more usual.

Since 2005, the history of all changes to the Linux source code is recorded using git, the source code management tool. Linux developers work each with their own git repository. Eventually, they submit their change proposals to follow a review process until (if accepted) their changes make their way into the Torvalds repository. It contains only accepted changes, which periodically are published as major (upstream) releases (e.g. v6.2).

Switching to new upstream releases in production is not always easy, since changes in the kernel API may break some applications. Therefore, many deployers prefer to avoid the upgrade for long periods of time. Stable releases are produced to acknowledge this fact: they ensure minimal functional changes, but provide bug fixing. Hence, it is less risky to upgrade to a newer stable release. The activity to produce stable releases happens in a separate git repository known as the "stable repository". In it, parallel branches are kept for the stable releases of each major release. For each of these branches, changes that are accepted in the upstream repository are examined and "ported" to it if they are considered important for production, and not likely to break deployments.

Since this practice of sticking to stable releases is so widespread, and in many cases deployers want to follow it for very long periods of time, the Linux project considers some major releases as "longterm" (long-term support, or LTS), and produce frequent stable releases of them during several years. At any given moment there are several active LTS "branches", each producing stable releases for different major releases. LTS branches are of special importance. Anything that happens to LTS releases have a great impact the whole IT industry, given how popular is the Linux kernel as the base for production deployments from different kinds, from cloud computing, to mobiles, to embedded devices.

LTS branches are very active. They produce releases every few days, totaling hundreds of changes to the source code per month. In principle, all this activity is focused on fixing bugs that were already fixed in the upstream repository, which would lead to two interesting implications: (1) there are a lot of bugs in major releases, because a lot of changes are needed to fix them, and (2) the time that a change takes to reach LTS releases is very important because it shows for how long a known fix is still not present in the kernels that the industry is using.

The first implication is troublesome, because the Linux kernel is widely considered to be trustworthy, being the base of many 24x7 production deployments. Therefore, either not all changes are fixing bugs that affect popular deployments, or the industry considers as trustworthy a kernel in which thousands of bugs are fixed every year (with the consequence that they were present in earlier releases of it). The degree to which fixed bugs affect users could be related to the fact that kernel deployments normally include only a part of the modules in the source code. Therefore, depending on the zones of the kernel affected by fixes they could affect more or less to "normal" use cases.

The second implication signals an interesting parameter to follow: the time it takes for proposed changes to the kernel to reach LTS distributions. This parameter would say a lot about to what extent do the review and porting processes contribute to delays in the release of changes deemed to be useful in production.

We have found little research devoted to better understand what is happening in these LTS branches, and how that activity could affect to these implications. In consequence, in this study we investigate what happens in LTS releases of the Linux kernel, with the main aims of (1) understanding how many bug fixes are applied to LTS branches, and for how long bugs are present in them, (2) to identify which zones of the source code are been changed, and (3) how long accepted changes are delayed, since the moment they were produced to the moment they are accepted in the upstream repository, and to when they are applied in LTS releases.

The main contributions of this study are: (1) the estimation of a lower bound of the number of bug fixes applied to each LTS release (12,500 for 5.x LTS releases), and a characterization of the time since a bug is introduced to when it is fixed in a LTS (median of over three years in most cases), (2) a description of which zones of code are changed in maintenance activities, finding that a large fraction of the changes happen in areas that are optional for deployments, depending on their specific architecture, devices, protocols or file systems used, and (3) a detailed characterization of the distribution of the lag since changes are produced until they land in maintenance releases, and how that lag can be split in phases, finding that at least in some cases, that lag is longer than three years, even though the median is around 30 days.

These contributions help to understand why the LTS releases of the Linux kernel are considered "suitable for production", even when they have so many bug fixes (which implies that those bugs were present before being fixed), and provide a way of quantifying and tracking the time spent in the review and porting processes, needed to ensure the quality of these "production-ready" releases.

II. Related Research

Much research has already been done on Linux [1], [2], its repositories [3], [4], its other data sources [5], [6], and its release management [7].

Back in 2012, Tian et al. [8] used an approach based on automatically examining commit records to identify bug-fixing commits. Passos et al. [9] focused on commits that modify configuration files to uncover feature removal plugins, while Xu et al. [10] offered a dataset consisting of threads discussing patches on mailing lists, including references to relevant posts. More recently, Page et al. [11] focused on automatically identifying commits that fix security issues. Duan et al. [12] examined a dataset of Linux exploits, including a number of its versions, and Tan et al. [13] analyzed the communication that follows the submitted changes, which is roughly equivalent to the commit comment when a patch is accepted.

Recently, Lyu et al. [22] have taken advantage of the fact that since October 2013, the Linux kernel developers have started labelling bug-fixing patches with the commit identifiers of the corresponding bug-inducing commit(s) (i.e., use "Fixes: ⟨commit-hash⟩" in bug-fixing commits, where ⟨commit-hash⟩ is the hash of the commit that introduced that bug) as a standard practice and evaluated different implementations of the SZZ bug-introducing commits detection tool.

Palix et al. [14] investigated the evolution of fault patterns in the Linux kernel from 2003 to 2010, building upon the work of Chou et al. [15]. By replicating their experiments on more recent Linux versions, Palix et al. found that while the kernel has grown significantly, the fault density has decreased. Interestingly, the fault rate in the driver directory, once a major concern, has now fallen below that of other critical components. These findings offer valuable insights for future research and development efforts aimed at improving the reliability and security of the Linux kernel.

Ahmed et al. [5] studied the reliability of the Linux kernel by analyzing bug data over a seven-year period. The analysis reveals that while the Linux kernel has made significant strides in improving reliability, certain areas, particularly those related to less common configurations and hardware platforms, still require attention. The study highlights the importance of understanding the root causes of bugs, such as inter-module dependencies, to further enhance the reliability of the Linux kernel.

Tan et al. [16] analyzed 2,060 real-world bugs from Linux, Mozilla, and Apache to understand their root causes, impacts, and components, aiming to inform the development of effective tools for detecting and recovering from software failures.

Jiang et al. [17] analyzed eight years of Linux kernel patch reviews to understand the factors influencing patch acceptance and integration time, finding that only one third of submitted patches are accepted, and that factors like developer experience, number of affected subsystems, and number of requested reviewers impact review and integration time.

Lin et al. [18] looked at how high-severity bugs in upstream packages are fixed within Linux distributions like Debian and Fedora. It found that a significant portion of bugs are not explicitly fixed upstream, and that fixing bugs locally can be faster than waiting for upstream fixes. The study also highlights the importance of traceability tools to facilitate the identification and application of upstream bug fixes, ultimately reducing the cost of upstream bug management.

Da Costa et al. [19] investigated the delay between issue resolution and integration into official releases in three open-source projects (Argo UML, Eclipse, and Firefox). The researchers found that a significant proportion of addressed issues experience delays, often due to factors like system-wide impact assessment. By analyzing issue data, they developed a model to predict integration delays, highlighting the role of integrator workload as a key factor. This research underscores the importance of considering integration delays when assessing the overall time it takes to address issues and deliver value to users.

Bettenburg et al. [20] studied the contribution management processes of Linux and compare it to the one used in Android. The study reveals distinct approaches, with Android employing a hybrid model and Linux a more open model. By analyzing these two cases, the paper identifies the strengths and weaknesses of each approach, offering valuable insights for organizations seeking to effectively manage external contributions to their software projects.

Da Costa et al. [21] researched the impact of rapid release cycles on the integration delay of fixed issues. By comparing traditional and rapid release cycles in the Firefox project, the researchers found that rapid releases, despite their intention to accelerate delivery, can actually lead to longer integration delays. The study identifies factors such as the prioritization of backlog issues in traditional releases and the focus on polishing issues in rapid releases as key contributors to these delays. The findings suggest that while rapid release cycles can offer benefits like increased stability and user feedback, they may not always result in faster delivery of fixed issues.

III. Context

To better understand the methods and results of our study, it is convenient to introduce briefly how changes to the source code are processed before they land in a stable release. This will also be useful to introduce the terminology that we will use during the rest of this paper.

A. The path to stable

The Linux kernel review process is based on discussion in mailing lists, where the code changes ("patches" in the Linux kernel parlance) are attached to messages. Patches are produced by developers in their local git repository, as commits. To be accepted, those patches usually need to be small ("solve only one problem") and well documented in the commit message. As they are approved, patches travel through a corresponding hierarchy of git repositories, until they are eventually accepted and included in the "Torvalds git repository" (Upstream). This is the moment when we consider the change was accepted into the kernel code base.[1]

TABLE I
Average number of patches per month for the LTS releases which are still maintained in November 2024, considering a cut-off date of October 2023.

Branch	Patches per month
v4.19	447
v5.4	514
v5.10	662
v5.15	800
v6.1	970

The Upstream repository follows several phases when preparing a new upstream release, with different rules about which patches are accepted during each phase. The preparation starts with a sequence of Release Candidate releases (e.g. "v6.2-rc1", "v6.2-rc2"), which are thoroughly tested, until the major release (the upstream release is finally published). This is done by tagging the release commit with the release identifier (e.g., "v6.2"). Upstream releases are produced approximately every three months.

At that point, the release is considered to be in maintenance. While a new upstream release starts to be prepared in the Upstream repository, the current upstream release is ported to another repository, Stable. This repository hosts all releases under maintenance, trying to ensure that they are as much free of defects as possible. To maintain the new upstream release, a new stable branch is created for it. Patches are applied to this branch until a new major release is published. But some major releases are designated as a Long Term Release (LTS), and their corresponding stable branches are maintained for much longer periods, usually several years. At a given moment, several LTS releases are maintained. As a consequence, any change to the source code which should go to production must be applied (when appropriate) to all the currently maintained releases (the future release, the current release and each of the LTS releases).

The patches applied to the stable branches are patches ported from the upstream repository, usually for bug-fixing changes. These new commits to stable branches are grouped into stable releases, which are produced "as-needed", usually at least once every week, and are prefixed by the major identifier followed by a consecutive integer, starting with "1", such as "v6.2.23".

In November 2024, the following LTS branches are being maintained: v4.19, v5.4, v5.10, v5.15, v6.1, and v6.6. As

[1]Details about the submission of patches and their review process: https://docs.kernel.org/process/submitting-patches.html

an example, v5.15 was released in October 2021, and will be maintained until December 2026[2]. Table I shows the number of patches per month for those branches (except for v6.6, which started maintenance after the end of the cut-off date of our data).

Patches in LTS releases correspond to a patch in the upstream repository. They may be exactly equal, if they are ported as such, or different, if they had to be modified in the porting process, due to the differences between the current code in the upstream repository, and the "old" code maintained in the branch. In other words, all patches in LTS (and in general, stable) releases went through a review process that finished by being accepted in the upstream repository. It is also important to notice that the same patch (or a modified version of it) may land in several LTS branches, if it is ported to all of them.

While patches move from one repository to another, Linux developers are careful to maintain the authorship data (author and author date), even in the case of porting or migrations. Some other data, including the commit identifier (the commit "hash"), may change, due to git fast-forwards, changes in the commit comment, and other reasons.

B. Terminology

During the rest of this paper, we use the following terms:

- Patch: Change to the source code. A patch has to be wrapped into a git commit. Each patch will appear with different commit ids in each of the stable branches (including mainline).
- Patch Authorship information, including author's email and date: Author's email and date in which the patch was created. All commits of a patch will maintain the original patch's authorship information.
- Upstream repository: Torvalds Linux git repository.
- Upstream commit: Commit in the upstream repository.
- Upstream release: A release in the upstream repository. It can be a mainline release, with an X.Y identifier (such as v5.15) or a candidate release with a vX.Y-rcN identifier (such as v5.15-rc1).
- Stable repository: The repository where all maintenance releases are hosted.
- Stable branch: A git branch where all the maintenance of a given upstream release is made. It branch starts at the commit of an mainline release (such as "v5.15") which becomes the identifier of the branch. Some stable branches are declared as LTS branches.
- Stable commit: A commit exclusively found in the stable repository. A patch that is ported to Stable appears in one or more commits, one per each Stable branch. Such commit includes a reference to its corresponding upstream commit id.
- Stable release: A release in a stable branch, with an identifier of the form vX.Y.Z, with Z being an integer starting at 1. All commits in a stable release X.Y.Z are found in the X.Y stable branch.

IV. Goal and Research Questions

The main goal of our study is to characterize patches (changes to the source code) in LTS releases of the Linux kernel. We consider three aspects of patches in these releases, which can be summarized in our three main research questions:

- RQ1: How is the bug fixing activity in the branch?
- RQ2: In which zones of the source code are the patches located?
- RQ3: How long did it take for the patches to be released since they were produced?

RQ1 is focused on bug fixing patches. In principle, according to the project rules, patches in LTS releases should "fix a real bug that bothers people or just add a device ID"[3]. But those fixes may have other patches as dependencies, so they also enter LTS releases. We will be interested in finding those patches that are really fixing bugs, and how long it took to fix those bugs since they were introduced. From the number of fixed bugs, we will also estimate the number of bugs present in the first release of the branch.

RQ1 can be refined in three specific questions: (RQ1a) How many patches fixed bugs?, (RQ1b) how many bugs were present in the first release of the branch? and (RQ1c) how long did it take since fixed bugs were introduced?

RQ2 is designed to investigate whether maintenance activity is concentrated in some zones of the source code, or spread all over it, and in which proportions. For the Linux kernel, this is an important question, because not all changes to the source code affect users the same way. For example, some changes may be to core parts of the kernel, which likely affect all use cases. But some other changes may be to code supporting some specific device, file system, or network protocol, which only affect to the use cases where it is relevant.

RQ3 deals with how efficient, in terms of time span, is the process by which patches are reviewed and ported to stable releases. In other words, for how long users of LTS releases will not be able to use kernels with fixes that are already ready, but have still not reached those releases. Any process aimed at the production of code of good quality needs time to consider proposed changes. With this RQ we intend to quantify it.

V. Method

The study presented in this paper is based on the analysis of two git repositories: the upstream repository[4], and the stable repository[5]. For convenience, we produced

[2]Summary of the different kinds of releases of the Linux kernel: https://www.kernel.org/category/releases.html

[3]Detailed information about stable releases: https://docs.kernel.org/process/stable-kernel-rules.html

[4]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[5]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

two datasets:

- Upstream metadata. We extract metadata for all commits in the upstream repository, using Perceval [23], producing a file in JSON Lines format ("Upstream Raw Data"). From it, we produce a CSV file consisting of a row for each commit in the Linux upstream repository. Each row has the commit hash, and the author and committer dates. We also add to it the "fixed" commit, according to the "Fixes" field, if present (see Subsection V-A, and kernel zone to which we assign it (see Subsection V-B). This will be the "Upstream Dataset".
- Upstream-stable mapping. This is set of pairs of commit ids (hashes), one pair for each patch applied to both the upstream repository and a stable branch. This information is extracted by scanning the "upstream" record in the commit message of the stable repository, found in every patch ported from upstream to stable releases. For example, in commit 08970f8cf, the upstream record is as follows:

  commit 81e2073c175b... upstream

  The dataset consists of a CSV file, consisting of one row for each commit in each branch of the stable repository, with information about the corresponding upstream commit (including authoring date and upstream release date), and the stable release date. This will be the "Stable Mapping Dataset".

The rest of this section describes the methods we used to answer RQs by analyzing these datasets.

A. RQ1

To determine which commits correspond to bug fixes, we use the "Fixes" field which may appear in the commit message. This field has the following structure:

Fixes: <hash12> ("<patch_summary>")

For example:

Fixes: e21d2170f366 ("video: remove ...")

According to the Linux kernel documentation[6], the "Fixes" field "indicates that the patch fixes an issue in a previous commit. It is used to make it easy to determine where a bug originated, which can help review a bug fix. [...] This is the preferred method for indicating a bug fixed by the patch". It may happen that in some cases developers do not care of including that field, or that they cannot determine where the bug was originated. But since this is information very useful for the review process, and it is used during it, we think it is safe to assume that those commits with a "Fixes" field are considered by developers as bug fixing commits, and that they also agree that their origin is signaled by the commit hash they include.

[6]Submitting patches: the essential guide to getting your code into the kernel:
https://www.kernel.org/doc/html/v4.13/process/submitting-patches.html

We extract, from the Upstream Raw Data, the the commit identifier (the hash) of the "fixed" commit for all cases when the "Fixes" field is present in the upstream metadata, and add it to the Upstream Dataset.

To determine the fraction of commits in LTS releases which are "fixes", we will use the Stable Mapping Dataset. First, we find upstream commits for all commits in LTS releases, and check if they have a Fixes field. Then it is just a matter of dividing the total number of commits in each LTS branch by the number of corresponding upstream commits with a Fixes field.

Once we have located the fixed upstream commit, we also have access to its commit time, which is when the patch was finally accepted in the kernel. Then, we can compute the time to the moment the fixing commit is published in a LTS branch, which is when the fix became available for production.

B. RQ2

To determine the zones with higher maintenance activity, we consider the directory structure of the kernel, which is also used by the Linux developers to organize a hierarchical structure for the review process. We have considered first-level directories as zones. A commit is assigned to the zone of the files it touches. If it touches files in more than one zone, it is assigned to the zone with more files touched. It is important to notice that only a small proportion of commits touch files in more than one zone. The zone is determined using the data in the Upstream Raw Data, and added to the Upstream Dataset.

Upstream commits corresponding to LTS branches are mapped as they were in the previous section, using the Stable Mapping Dataset. Then, finding them in the Upstream Dataset, their zones are determined. Computing commits per zone in a LTS branch is just a matter of counting zones for all the commits in that branch.

C. RQ3

To investigate the lag since patches in LTS releases were produced, we also use the Stable Mapping Dataset. From it, for each commit in a LTS release, we compute three time spans:

- Publication lag (for a LTS branch): Time since the authoring date in the upstream commit until the release date in the branch. This time lag represents for how long the patch has been ready to be used, but has not been visible to users of the LTS branch.
- Review lag: Time since the authoring date of the upstream commit until the upstream release date. This is how long the review process lasted for the patch, until it was accepted in the upstream repository.
- Porting lag (for a LTS branch): Time since the upstream release date until the stable release date in the branch. This is the time lag in being ported to the branch from the upstream repository.

TABLE II
Number of commits, commits with Fixes field, fraction of commits with Fixes field, number of commits fixed by later commits in the same branch, and estimation for lower bound on bugs in the first release of the, per branch, for our branches of interest, until the cut-off date of October 2023.

Branch	Commits	Fixes	Fraction	Fixed	Bugs
4.1	6,599	2,004	0.304	110	1,894
4.9	22,601	9,820	0.434	524	9,296
4.14	26,320	12,680	0.482	663	12,017
4.19	26,820	14,168	0.528	736	13,432
5.4	24,712	14,725	0.596	762	13,963
5.1	23,197	14,626	0.631	683	13,943
5.15	19,207	12,697	0.661	576	12,121
6.1	10,672	7,361	0.690	260	7,101

Therefore, the total lag can be split into review lag and porting lag, helping to better understand how it is composed. For each of these time lags, we calculate the distribution for all commits in each branch, and their evolution as new releases happen in a branch.

VI. Results

We organized results by the research question, below.

A. RQ1

To answer RQ1a, we calculate the number of commits including a "Fixes" field for each LTS branch, and the fraction of them with respect to the total number of commits in the branch (see Table II). We consider that this is a lower bound on the number of bug fixes for each LTS branch, from the point of view of developers. It is "a lower bound" because for some commits that developers consider a bug fix, maybe they didn't determine the fixed commit, and thus they don't include a "Fixes" field. It is "from the point of view of developers" because according to the kernel documentation, this field is "the preferred method for indicating a bug fixed by the patch", and this was checked by reviewers during the patch review process.

The table shows how the fraction of bug fixes over the total number of commits grows for newer LTS branches. This does not necessarily mean that the fraction of bug fixes increases: it could also be due to developers being more strict with including this "Fixes" field when submitting a bug-fix patch for review. In any case, it is clear that for recent branches, more than 50% of commits are for bug fixing.

To answer RQb, we estimate a lower bound for the number of bugs present in the first release of the branch. We consider two subsets of the commits in the branch releases: bug-fixing commits (fixes, commits with a "Fixes" field) and bug-fixing commits fixing a commit in the same branch (fixes_branch, commits with a "Fixes" field pointing to an upstream commit ported to the same branch). All commits in fixes fix a bug which was present in earlier releases of the branch. Therefore, those bugs were present in the first release of the branch, except if they were introduced later. But this latter case are commits in fixes_branch. Thus the number of fixes minus the number of fixes_branch is a lower bound on the number of bugs that were present when the branch entered maintenance (e.g., in release 5.4). This is column "bugs" in Table II. Of course, there could be more bugs, both because some bugs remain uncovered or unfixed in all releases of the branch, and because in some bug-fixing commits fixing maybe developers didn't include the "Fixes" field.

Figure 1 answers RQ1c, "how long it took since the fixed bug was introduced". Each violin diagram shows the distribution of this time for a LTS branch. The stats of these distribution are presented in Table III.

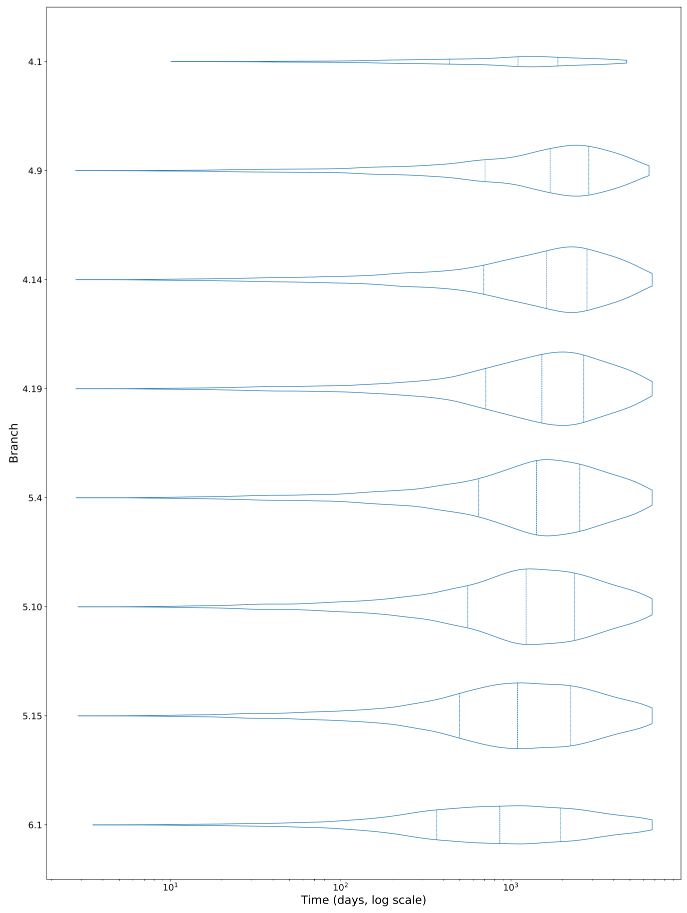

Fig. 1. Time from the moment a fixed commit is committed, to the moment the fixing commit is published in a LTS, for all the fixing commits in each studied LTS. branch

It is interesting to notice how the time from the bug introducing commit to the bug fix is apparently increasing, with a median time that goes from more than 1,500 days (for 4.9, 4.14 or 4.19) to around 1,000 days for 5.15. In any case, the median time is longer than three years for most branches.

B. RQ2

Figure 2 shows the evolution per month of the number of commits for zones in the Linux kernel. We have selected

TABLE III
Statistics of the time from the moment a fixed commit is committed, to the moment the fixing commit is published in a LTS, for each studied LTS branch (time in days).

Branch	Min	Q1	Median	Q3	Max
4.1	10.05	434.25	1,099.86	1,885.06	4,789.17
4.9	2.76	703.98	1,698.03	2,863.70	6,474.53
4.14	2.77	691.79	1,610.37	2,796.38	6,750.89
4.19	2.77	711.18	1,518.83	2,674.09	6,750.89
5.4	2.78	645.95	1,414.29	2,534.75	6,750.89
5.1	2.85	557.13	1,226.71	2,358.49	6,750.90
5.15	2.86	497.67	1,091.93	2,230.38	6,750.90
6.1	3.49	366.34	859.63	1,949.12	6,750.90

v5.10 and v5.15 for these figures because they are branches with a long history (35 and 24 months), but still active at the time of writing the paper. However, other branches show similar behaviors.

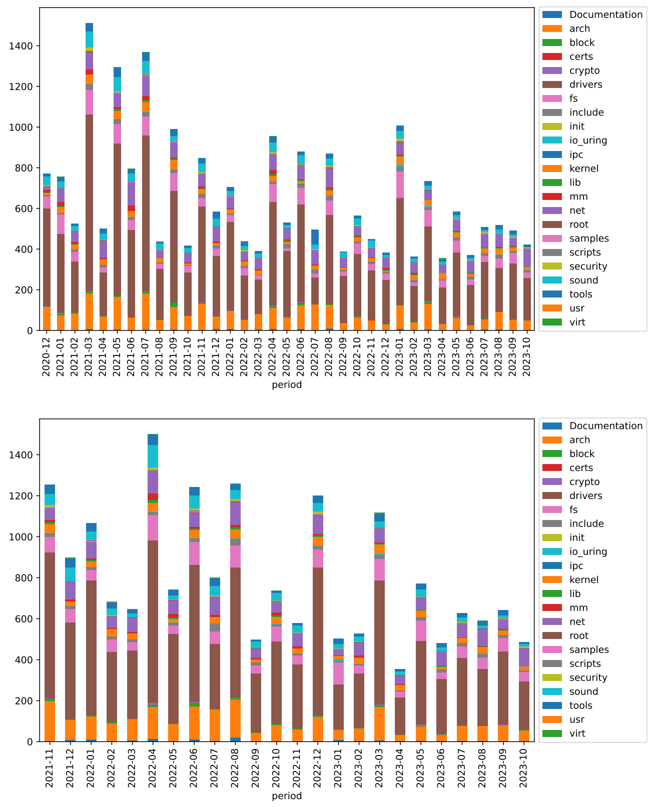

Fig. 2. Evolution over time (per month) of the number of commits for each zone in the source code of the Linux kernel. Top: branch v5.10, bottom: branch v5.15.

These charts help to understand the zones of activity in stable branches. Most of it happens in “drivers” (more than 50% of the activity in most months). This is a directory devoted to drivers for all devices supported by the kernel. A considerable fraction of the activity happens also in “arch” (code for specific architectures), “net” (network protocols), and “fs” (file systems). All of them have in common that they are composed of many modules, only a relatively small part of them used in most kernel use cases.

There are also commits in zones such as “Documentation” (technical documentation of the kernel) or “samples” (sample files and code snippets to show how to write kernel modules and interact with the kernel internals).

C. RQ3

To explore the time lags that patches experience until they land in a LTS branch, we have depicted their distribution in three figures: Figure 3 for publication lag, Figure 4 for review lag, and Figure 5 for porting lag.

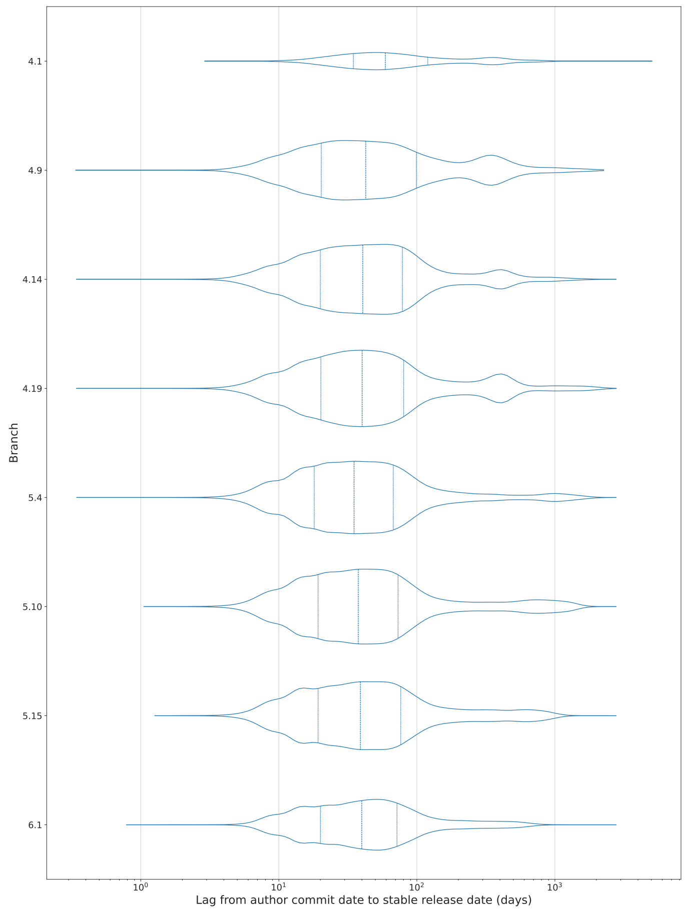

Fig. 3. Distribution of publication lag (authoring to stable release date) in each branch: violin charts and quartiles (log scale).

The first figure shows the direct answer to RQ3: each of the violin charts show the distribution of the publication lag experienced by patches landing in the corresponding LTS releases. It varies from hours to more than 3 years in some cases, but a great proportion of the patches land in less than three months, with half of all patches lagging less than 30 days in most branches.

The second and third figures let us understand in more detail how publication lag is produced. Porting lag shows how patches land relatively fast in the LTS releases once they have landed in upstream releases. The median in this

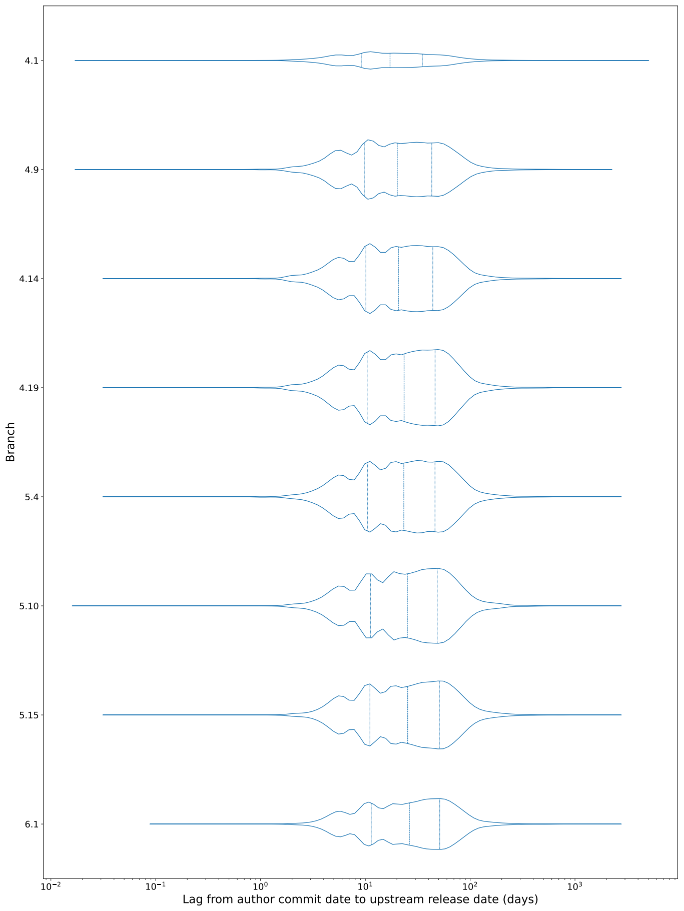

Fig. 4. Distribution of review lag (authoring to upstream release date) in each branch: violin charts and quartiles (log scale).

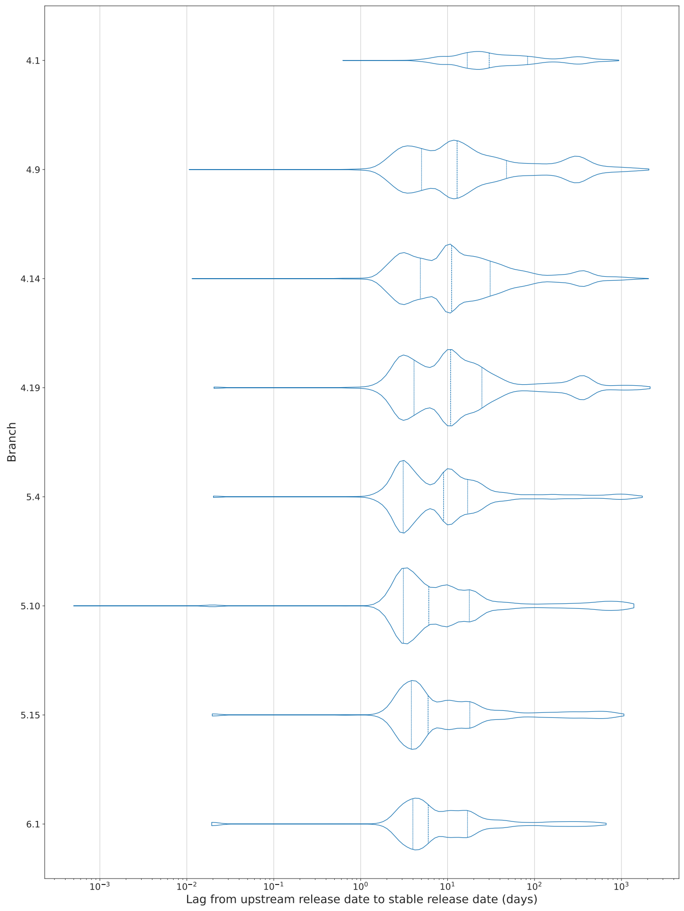

Fig. 5. Distribution of porting lag (upstream release to stable release date) in each branch: violin charts and quartiles (log scale).

case is less than 15 days for all branches, and a very large fraction of patches have a lag of less than 30 days. Review lag show longer spans. In this case, even when a very large fraction of patches land in upstream in less than 100 days, the median is close to one month. For almost all branches, 25% of the patches take more than 50 days to reach upstream releases.

VII. Discussion

After presenting the results of our study, in this section we discuss some of their implications and limitations.

A. Key findings

Answering RQ1, we found that most of the changes to the source code in LTS branches are for bug-fixing. This is consistent with the intent of stable branches in general, which should only include bug fixes and other changes needed for those bug fixed to work. Indeed, the numbers we provide are a lower bound of the real number of bug fixes, since not all commits which fix a bug have a "Fixes" field. We also quantify a lower bound for the number of bugs present in the first releases of each branch: more than 12,000 for all 5.x LTS branches.

We also found that the time from the moment a bug is introduced in Linux to the moment it is fixed in a LTS is long, with 50% of the bugs being fixed in more than two years for all LTS branches studied. Which in turn means that 50% of bugs present in Linux LTS releases were present in the kernel, unfixed, for at least two years.

RQ1: bugs in LTS releases
Most of the patches in LTS releases are bug fixes. First releases of 5.x LTS branches had more than 12,000 bugs, which were later fixed during the life of the branch. Half the bugs fixed in LTS releases were introduced more than two years ago.

The results for RQ2 show how even when there is a large diversity in zones touched by patches, there is some concentration of them in "drivers", "arch", "net" and "fs". All of these zones have in common that their code may or may not be used, depending on the specific configurations of the kernel and on use cases. For example, if drivers, architectures, protocols and file systems touched by a set of patches are not relevant for a use case, there is no impact of those patches for that use case.

RQ2: zones of source code
A very large fraction of patches affect zones with many components, which may be used or not, depending on the use case. This means that many of those patches affect only to specific cases. Therefore, the actual number of patches affecting kernel users is much lower than the total number of changes affecting their LTS branch of choice.

For RQ3, we have studied how long patches took to land in LTS branches. The main results in this respect are a detailed characterization of that lag, and of the two lags contributing to it: review lag and porting lag (Figures 3, 4, and 5).

RQ3: time lag since production
Patches take time to be included in LTS releases since they are produced (sometimes more than three years). But half of all patches in LTS releases have a publication lag shorter than 30 days. It can be split in review lag and porting lag: the latter is relatively short, with a median of less than 15 days, while the former is longer, in the order of 30 days.

B. Implications

The findings for RQ1 show that there are a lot of bugs in early LTS releases, which are later fixed, but most of the bugs fixed were present in the code for more than two years. All of this despite the strict review process that the Linux project follows for all patches, and the intensive testing that is performed on the code before releasing a new stable release. However, it is important to highlight that these are bugs from the point of view of developers, who may have a very exigent standard to decide what is a bug.

The results obtained for answering RQ2 show how maintenance activity happens in many different zones of the kernel, although the relative activity in each zone varies.

The findings for RQ1 and RQ2, combined, shed some light on an interesting aspect of the maintenance of the kernel. As we already commented, the number of patches per branch is relatively high, in the order of hundreds of patches per month. This could lead to the conclusion that the kernel is very unstable: if so much activity is needed to fix bugs, it is because there were many bugs in the kernel. This is contradictory with the general idea that the kernel is good enough, and bug-free enough, for production. Our findings help to nuance this situation.

On one hand, with RQ1 we found a good number of patches fixing bugs. This is positive, in the sense that represents a high maintenance activity. But at the same time it could be considered negative, because all of those bugs were present in the code before they were fixed. What is more interesting, we can assume the number of bugs is stable over time, given that the number of patches per LTS release is stable for different LTS branches. Therefore, about 6,000 bugs are introduced in the kernel every year.

Results for RQ2 provide more context. A lot of maintenance activity happens in zones of the kernel which could be not relevant for many use cases. This means that fixed bugs may not affect many of them, which could explain why the industry has so much trust on Linux even when it has a relatively high number of bugs.

But we don't analyze the number bugs present in the usual configurations deployed in production, which is the number that really impact users. Therefore, more research is needed to learn to what extent bug fixes in the different kernel zones, or in general in LTS releases, affect or not popular configurations of the kernel. These are the important bugs for the perception of users about the stability of the kernel.

We consider our findings with respect to bugs in LTS branches mainly as an starting point, quantifying some interesting characteristics of how they are fixed and introduced, but still with a difficult interpretation. From a practical point of view, they can be useful to track, for the next Linux LTS branches, their performance in this respect.

Our analysis of publication lag, review lag, and porting lag (RQ3) provides metrics to analyze an important balance for the evolution of the Linux kernel. This is the balance between taking as much time as needed for a careful review of code changes, and to port to LTS branches, and being as quick as possible to release kernels with those changes to LTS users (minimizing time-to-production). This is a balance that all software projects performing maintenance should consider. To measure it, we measured the lag for patches that are admitted in releases. Those patches fulfill the conditions to be accepted in production since the moment they were produced, given the review process accepted them as such, and they were also selected for porting. Therefore, any lag in being included in a release is due to delays imposed by the process used to ensure quality.

In our case, we have measured the process, which is the first step to track if it improves or degrades. More research is needed to know if, in the case of the Linux kernel, the review or the porting processes can be improved in terms of how long do they take, while keeping the same quality.

For now, we have found some patches taking a very long time to be included in LTS releases (even more than one year). But a very large fraction of all patches in those releases are ported in less than a month, with almost half of them in less than 10 days, which are quite short lags. A large fraction of patches is reviewed in a relatively short time: half of them take about one month from being produced to being accepted in an upstream release.

C. Threats to validity

Several issues could impact the validity of our results:

Internal validity. We analyzed all LTS branches with active maintenance in November 2024, but they may not be representative of all Linux LTS branches. Besides, we have only considered those with a certain history, to draw more general results, but this could have led to ignoring recent trends, still not evident in older branches. Finally, even when we reviewed our analysis processes and scripts, we could also have errors in them.

Construct errors. When measuring lags, the most relevant is the lack of accuracy in author dates for commits. This date depends on the date and time of the computer where the commit was done, which could be wrong (even when nowadays most computers keep an accurate synchronization), and on it being respected when the patch travels through repositories during the review process. On the other hand, if the authorship date is reset at some point, the real author date would likely be earlier, which means that the publication and review lags we calculated are in fact lower bounds, thus not affecting results significantly.

When determining zones of the source code we could be using a wrong zoning for patches: another one could be more representative than high level directories. In addition, when more that one zone is touched by a patch, we could be wrong with the assignment decision (we selected the zone with more files touched). However, only a few patches touch files in more than one zone, which minimizes this risk.

All of our study is based on patches being a relevant unit for maintenance tasks. However, in principle patches can be very different, of very different sizes, and requiring very different efforts. This could impact the significance of the results, since we are considering all patches as equal. Fortunately, we are considering only patches to LTS branches, which are very specific and all of them relatively small, and all patches went through the Linux kernel review process, with mandates that "each patch solves only one problem", and encourages relatively small and independent patches.

Our study also depends on kernel developers following specific practices, such as tagging commits in maintenance releases with the upstream commit hash, or using the Fixes field. Fortunately, the review process and the care of developers in following their rules, minimize this risk.

External validity. We do not claim that our results are generalizable to other projects, and therefore there are not external threats to validity. Furthermore, we only analyzed some LTS branches, but we expect our results are extensible to other recent LTS branches. This could be wrong, because branches in the past or in the future could be very different due to changes in the project itself, or in its environment.

VIII. Conclusions and further research

We presented a study characterizing maintenance activities in the Linux kernel. We focus on LTS releases because of their importance for the industry, since they are usually the Linux releases deployed in production. Therefore, this study helps to better understand how a key component of the global digital infrastructure is being maintained, with many implications on its trustability.

Our characterization shows some details of bug fixing process in LTS releases, including lower bounds on the number of bugs fixed, and an analysis of how long it takes from bug introduction to bug fix, for bugs present in these releases. It also shows that patches touch many zones in the code, but they do not affect equally all use cases. Finally, we also characterize the lag since patches are ready to be used until they are released, so that they can actually be used by deploying a LTS release, and propose metrics to track how review and porting processes are improving or not in terms of induced delays in time-to-production.

In the previous section we showed some directions for future research, such as better understanding the effect of patches in LTS releases for specific use cases, or the more nuanced consideration of bug fixes when estimating the number of bugs present in the Linux kernel at any given moment. This study opens these and other lines that help to better understand how it is maintained one of the fundamental parts of the current software infrastructure.

Author roles (CRediT statement)

Removed to ensure a double blind review process.

Reproducibility and data availability

The data presented in this paper (including annotations and final results), along with the software for producing the results, are available in a reproduction package[7].

Reproducibility self-assessment, according to [26]

Raw: R, Extraction: U+*, Parameters: U, Processed: U+*, Analysis: U+*, Results: U+*

References

[1] Q. Tu et al., "Evolution in open source software: A case study," in Proceedings 2000 International Conference on Software Maintenance. IEEE, 2000, pp. 131–142.

[2] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," Journal of Systems and Software, vol. 83, no. 3, pp. 485–501, 2010.

[3] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: an empirical study of how linux uses git," Empirical Software Engineering, vol. 21, pp. 260–299, 2016.

[4] Y. Jiang, B. Adams, F. Khomh, and D. M. German, "Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel," in Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2014, pp. 1–10.

[5] M. F. Ahmed and S. S. Gokhale, "Linux bugs: Life cycle, resolution and architectural analysis," Information and Software Technology, vol. 51, no. 11, pp. 1618–1627, 2009.

[6] D. Schneider, S. Spurlock, and M. Squire, "Differentiating communication styles of leaders on the linux kernel mailing list," in Proceedings of the 12th International Symposium on Open Collaboration, 2016, pp. 1–10.

[7]Reproduction package: https://doi.org/10.5281/zenodo.14064242

[7] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger et al., "Progressive scrutiny: Incremental detection of ubi bugs in the Linux kernel," in 2022 Network and Distributed System Security Symposium, 2022.

[8] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in 34th International Conference on Software Engineering (ICSE 2012). IEEE, 2012, pp. 386–396.

[9] L. Passos and K. Czarnecki, "A dataset of feature additions and feature removals from the Linux kernel," in Proceedings of the 11th working conference on mining software repositories, 2014, pp. 376–379.

[10] Y. Xu and M. Zhou, "A multi-level dataset of Linux kernel patchwork," in Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 54–57.

[11] S. Page, "Analysing Linux kernel commits," Blog post, Feb. 2023, https://sam4k.com/analysing-linux-kernel-commits/.

[12] G. Duan, Y. Fu, M. Cai, H. Chen, and J. Sun, "DongTing: A large-scale dataset for anomaly detection of the Linux kernel," Journal of Systems and Software, vol. 203, p. 111745, 2023.

[13] X. Tan and M. Zhou, "How to communicate when submitting patches: An empirical study of the linux kernel," Proceedings of the ACM on Human-Computer Interaction, vol. 3, no. CSCW, pp. 1–26, 2019.

[14] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp. 305–318.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in Proceedings of the eighteenth ACM symposium on Operating systems principles, 2001, pp. 73–88.

[16] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," Empirical software engineering, vol. 19, pp. 1665–1705, 2014.

[17] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? And how fast? Case study on the Linux kernel," in 2013 10th Working conference on mining software repositories (MSR). IEEE, 2013, pp. 101–110.

[18] J. Lin, H. Zhang, B. Adams, and A. E. Hassan, "Upstream bug management in Linux distributions: An empirical study of Debian and Fedora practices," Empirical Software Engineering, vol. 27, no. 6, p. 134, 2022.

[19] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan, "An empirical study of delays in the integration of addressed issues," in 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014, pp. 281–290.

[20] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German, "Management of community contributions: A case study on the Android and Linux software ecosystems," Empirical Software Engineering, vol. 20, pp. 252–289, 2015.

[21] D. A. d. Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan, "The impact of rapid release cycles on the integration delay of fixed issues," Empirical Software Engineering, vol. 23, pp. 835–904, 2018.

[22] Y. Lyu, H. J. Kang, R. Widyasari, J. Lawall, and D. Lo, "Evaluating szz implementations: An empirical study on the linux kernel," IEEE Transactions on Software Engineering, vol. 50, no. 9, pp. 2219–2239, 2024.

[23] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM, 2018, pp. 1–4.

[24] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," Empirical Software Engineering, vol. 25, pp. 1294–1340, 2020.

[25] M. Maes Bermejo, J. M. Gonzalez-Barahona, M. Gallego, and G. Robles, "A dataset of Linux kernel commits," Feb. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10654193

[26] J. M. Gonzalez-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," Empir. Softw. Eng., vol. 17, no. 1-2, pp. 75–89, 2012. [Online]. Available: https://doi.org/10.1007/s10664-011-9181-9