An Investigation of Patch Porting Practices of the Linux Kernel Ecosystem

Xingyu Li
UC Riverside
xli399@ucr.edu

Zheng Zhang
UC Riverside
zzhan173@ucr.edu

Zhiyun Qian
UC Riverside
zhiyunq@cs.ucr.edu

Trent Jaeger
UC Riverside
trent.jaeger@ucr.edu

Chengyu Song
UC Riverside
csong@cs.ucr.edu

ABSTRACT

Open-source software is increasingly reused, complicating the process of patching to repair bugs. In the case of Linux, a distinct ecosystem has formed, with Linux mainline serving as the upstream, stable or long-term-support (LTS) systems forked from mainline, and Linux distributions, such as Ubuntu and Android, as downstreams forked from stable or LTS systems for end-user use. Ideally, when a patch is committed in the Linux upstream, it should not introduce new bugs and be ported to all the applicable downstream branches in a timely fashion. However, several concerns have been expressed in prior work about the responsiveness of patch porting in this Linux ecosystem. In this paper, we mine the software repositories to investigate a range of Linux distributions in combination with Linux stable and LTS, and find diverse patch porting strategies and competence levels that help explain the phenomenon. Furthermore, we show concretely using three metrics, i.e., patch delay, patch rate, and bug inheritance ratio, that different porting strategies have different tradeoffs. We find that hinting tags(e.g., Cc stable tags and fixes tags) are significantly important to the prompt patch porting, but it is noteworthy that a substantial portion of patches remain devoid of these indicative tags. Finally, we offer recommendations based on our analysis of the general patch flow, e.g., interactions among various stakeholders in the ecosystem and automatic generation of hinting tags, as well as tailored suggestions for specific porting strategies.

1 INTRODUCTION

Open-source software has greatly lowered the cost of software development. It is a common practice to reuse, customize, and even rebrand open-source projects. This also means that multiple versions or forks of the same upstream project can co-exist. In this study, we focus on the Linux kernel ecosystem. It is widely reused in numerous desktop and server distributions like Ubuntu, Fedora, and Red Hat, as well as in billions of Android, Amazon AWS, and Internet of Things (IoT) devices. These Linux forks are often based on specific Linux stable or long-term support (LTS) branches of the kernel source, which in turn were originally forked from the Linux mainline (also sometimes called Linux upstream). The Linux mainline is where all the development and bug fixes occur — including feature commits and patch commits. In contrast, stable or LTS branches only port patch commits from mainline, as their goal is to maintain stability.

For large and complex open-source projects like Linux that are being actively developed, it is inevitable that there will be a continuous stream of bugs being discovered (and introduced), including security vulnerabilities. Unfortunately, the co-existence of many forked downstream branches makes it challenging to keep track and fix issues in a timely manner. In Linux specifically, all bugs are supposed to be fixed in the mainline first (a.k.a., upstream first) [22]. Once a patch is merged into the mainline, downstream kernel maintainers are responsible for porting patches from upstream. But, as we will show, it is often not straightforward to figure out which of the upstream patches should be ported, especially considering that downstream branches may be forked from older versions that have diverged from the upstream, requiring more complex backporting. This problem is further exacerbated because the downstream Linux distributions are maintained by independent organizations with differing priorities.

Ideally, all of downstream branches, including Linux distributions and Linux stable/LTS branches should port all necessary and applicable patches from Linux mainline, and do not introduce any new bugs at the same time. However, we know anecdotally that the patch porting process is far from ideal. Several studies have been conducted to measure the vulnerability lifetime and patch timeliness [8, 29, 30, 39, 47], though few studies [61] focus on an ecosystem with upstream and downstream kernels. We believe a study of existing patch porting practices or strategies employed by the individual parties in the ecosystem is beneficial in several ways. For example, we hope to understand the pros and cons of different patch porting strategies, evaluate their implementation, and identify opportunities for enhancing the patch process in the Linux ecosystem (some of which may generalize to other open-source software at large).

To conduct this research, we utilized various publicly available data sources to produce measurements, which we then form hypotheses. To validate the hypotheses, we also engage in communication with 23 Linux distributions and LTS maintainers, who are among the top contributors in public Linux git repositories. In addition to measurement results, we also relied on (1) publicly available documents and (2) reverse-engineering labels in commit messages, which were then verified through discussions with the maintainers.

For measurements, we focus on three main metrics: (1) patch delay: how long it takes a patch to port from upstream to downstream; (2) patch rate: the frequency of patch porting, i.e., number of ported patches per unit time; (3) bug inheritance ratio: the fraction of

mainline bugs introduced in a downstream branch (in a given time period). In addition, to understand patch practices and implications better, we also categorize patches based on various information, e.g., security patches, labels provided by patch authors.

At a high level, we are interested in these high-level questions:

RQ1 What are the patch porting strategies employed by downstream kernel maintainers?

RQ2 What is the performance of these strategies in terms of the three metrics we defined? How far away are they from ideal?

RQ3 What improvements can be made to enhance the patch porting practices?

Overall, by studying 21 branches across 8 popular Linux distributions (including Android), we find drastically different patch porting strategies are employed by the various downstream kernel branches, and none of the strategies can achieve best results in all three metrics simultaneously. Specifically, if they choose to favor porting as many patches as possible and as quickly as possible, then they unavoidably introduce many more bugs at the same time. Conversely, if they choose to favor stability (i.e., fewer bugs introduced), then they tend to miss patches or port them with significant delays. Interestingly, even for the downstream branches that do favor stability, their patch porting performance can still differ drastically, indicating that there is significant room for improvement.

We summarize our contributions as follows:

- We investigate patch porting strategies in the Linux ecosystem both qualitatively and quantitatively. Specifically, we define three metrics that comprehensively and objectively capture the performance of patch porting strategies.
- In our study, we do not only report objective metrics but also look for factors that contribute to the performance captured by the metrics. This helps us understand the operational burden and rationale behind their strategies.
- Based on the study, we distill the results into a list of suggestions from the perspective of patch authors, maintainers, and the community at large to improve the patch porting process.

2 BACKGROUND

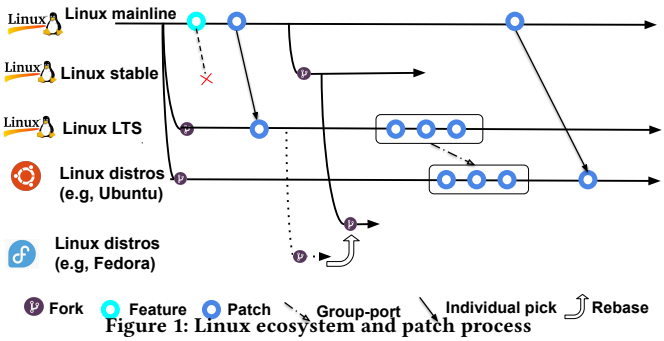

Figure 1: Linux ecosystem and patch process

The Linux Ecosystem. The Linux ecosystem's patch porting process is shown in Figure 1. The trunk branch is the Linux mainline [1], which keeps integrating all the new features and bug fixes from a great number of kernel developers and maintainers. The mainline branch is for development only and is not supposed to be used by end users directly. Instead, whenever Linux mainline reaches a milestone, it is forked into either a stable or a long-term support (LTS) branch [1], e.g., v4.19.y is a LTS branch while v4.20.y is a stable branch. In between the mainline releases, e.g., between v4.18 and v4.19, there are also release candidates such as v4.19-rc1 and v4.19-rc2 signifying the stages of the development process (e.g., most feature commits occur in rc1). Linux stable and LTS branches provide stability by integrating only bug fixes from the mainline without adding any new features (according to some predefined rules [9]), that could introduce new bugs. The main difference between Linux stable branches and LTS branches is their lifetime—stable branches usually are maintained for two to three months, whereas some Linux LTS branches have been maintained for six years [1]. Linux stable/LTS branches usually release a minor version (e.g., 4.15.3) once a week or once every two weeks [1]. Both the mainline and stable/LTS branches are maintained by the Linux community.

Outside of the branches maintained by the Linux community, we have branches maintained by Linux distributions, such as Ubuntu and Fedora. They are what the end-users will be using in practice. These distribution branches are typically a fork from one of the stable or LTS branches and typically include their own customizations (*e.g.,* better or additional hardware support). Distributions can also fork from other distributions. For example, the Linux Mint distribution uses the same kernel as Ubuntu. As such branches are separately maintained, the burden of tracking and applying patches from stable/LTS or even mainline in a timely fashion is on distributions' maintainers.

Upstream and downstream: Depending on the forking and patch porting relationships, we have upstream and downstream branches. The Linux mainline is often referred to as the *upstream*. Linux distributions are usually referred to as *downstream*. Of course, the relationship is also relative. As shown in Figure 1, Linux stable/LTS branches are upstream with respect to distributions and downstream with respect to the mainline.

Patch Porting Practices. In the Linux ecosystem, patches should be applied to the mainline first [22], which are then ported to downstream kernels. As mentioned, Linux stable/LTS branches port patches exclusively from Linux mainline. For Linux distributions, there are four basic patch porting primitives employed by downstream kernel branches (one or more may be applied in a given downstream kernel branch).

Group-port: For branches that are directly forked from stable/LTS branches, they port patches primarily from the corresponding upstream branch it is forked from. Because patches from stable/LTS branches are usually ported in groups (*e.g.,* via `git merge`), we call this patch porting practice *group-port*.

Individual pick: Distribution maintainers can also choose to bypass the immediate LTS/stable branches and directly port patches from the mainline, when those patches are deemed urgent. Because such patches are typically incorporated individually (*e.g.,* via the `git cherry-pick` command), we refer to this practice as *picking*. It is worth noting that distributions may also directly pick feature commits from Linux mainline (e.g., for supporting new hardware devices). We found that the picking is usually driven by each vendor's own bug tracking system, e.g., the launchpad system[4, 11, 18, 24]

in Ubuntu, the OraBug system[41] in Oracle-Uek and the bugzilla system[49] in SLE.

Minor version rebase: Different from selecting what patches to port, *rebase* unconditionally changes the base itself to a given point in a target branch, e.g., latest mainline, stable or LTS branches, automatically inheriting all commits that were made to the target. A minor version rebase indicates that it rebases from one minor version to another (e.g., from 4.15.1 to 4.15.2). It is similar in effect to group-port except that it by default ports/inherits all commits between two minor versions (whereas group-port can and does omit some).

Major version rebase: Contrary to a minor version rebase, a major version rebase represents a rebase that goes from one major version to another (e.g., from 4.15.y stable to 4.16.y). This implies a significant change in the branch as there can be a large number of commits (patches or even feature commits) between the two major versions, all of which are inherited automatically after the rebase.

Fixes tag and Cc stable tag. Mainline patches can be attached with two kinds of tags which make the patch porting easier for maintainers:

- A fixes tag looks like the following: `Fixes: a9e38c3e01ad ("KVM: x86: Catch potential overrun in MCE setup")`. It gives the commit (hash) that introduced the bug that is now patched. Such a tag is generated by the author(s) of the commit typically after manually analyzing the historical Linux versions. It is helpful for maintainers to determine if a kernel version is affected by the bug fixed by the patch. In fact, it can also allow one to compute the bug lifetime, which was interestingly never used in previous measurement studies [26, 39].
- A Cc stable mailing list tag looks like the following: `Cc:stable@vger.kernel.org`. This indicates that the author of the patch has determined that this commit satisfied the patch rules and should be considered for inclusion in stable/LTS branches.

3 DATASET, METRICS, AND METHODOLOGY

Measurement targets and dataset. For Linux distributions, we choose to measure a few representative popular distributions according to a list maintained by LWN [23] which has been tracking Linux distributions since 1999. Specifically, we study the following: four Ubuntu branches, three SUSE Linux Enterprise (SLE) branches, four Oracle Unbreakable Enterprise Kernel (UEK) branches, three Amazon Linux 2 branches, two Debian branches, three Android branches, one Fedora branch, and one Arch branch (both Fedora and Arch maintain a single kernel branch which keeps moving forward). In the case of Android, we picked three branches from Android-common [2], which is part of the Android Open Source Project (AOSP), further forked by downstream Android OEMs such as Samsung. We exclude OEM kernels due to the fact that OEMs, such as Samsung, are known to not publicly release their git repositories, which are required for our study. Note that there are other popular distributions such as Redhat and CentOS that we do not include. This is mostly because of the lack of public git repositories or insufficient data. For example, Redhat Enterprise Linux is not free. Alternatively, CentOS's role has been shifting over the past few years—initially being a downstream of Redhat, but now CentOS is becoming the upstream of Redhat [7]. In addition, we also measure six Linux LTS branches that are actively being maintained [1]: 4.4 LTS, 4.9 LTS, 4.14 LTS, 4.19 LTS, 5.4 LTS, 5.10 LTS and 5.15 LTS. We omit the Linux stable branches because the lifetime of stable branches is only two to three months, which is often less than the patch delays observed in LTS branches. Overall, our measurement targets include ***584K*** commits.

We list the patch porting strategies for all measured distribution branches in Table 1. Specifically, Arch and Fedora are the two distributions that stay on Linux stable/LTS branches only for a short period of time (usually for two to three months) and frequently rebase to new major release versions (e.g., from 4.14.y to 4.15). All other distributions choose to stay on a single base (i.e., a particular major release version) for an extended period of time (in years). While the distributions stay on a base, they choose a variety of patch porting practices such as minor version rebase, group-port, and picking from mainline. Most distributions stick to a consistent strategy. Ubuntu-18.04 is an exception where it first group-ports from Linux 4.15 stable, and then it group-ports from Linux 4.14 LTS and 4.19 LTS. SLE is also exceptional as it first group-ports from a stable release (instead of LTS), then it will directly pick commits from mainline after the stable branch ends (typically in two to three months).

Measurement metrics. We define the following three metrics to evaluate the patch porting performance:

- *Patch delay:* New Linux patches are officially released in Linux mainline first. So, we compute the patch delay for each patch applied to a downstream kernel branch as the time difference between when the patch appears in the mainline and when it appears in a downstream kernel branch. If the branch is a Linux distribution (e.g., Ubuntu), the patch delay is the end-to-end delay where the patch may be initially ported from mainline to an LTS branch first and then group-ported by the distribution.
- *Patch rate:* For each downstream kernel branch, we count the number of ported patches per day to compare the patching effort among distributions. The intuition is that it is challenging to obtain the ground truth in terms of the total number of patches that should be applied to a particular downstream kernel branch. However, approximately speaking, if a downstream kernel branch is well-maintained, it should periodically port enough patches. This metric will allow us to loosely compare the diligence of patch porting for various downstream kernel branches.
- *Bug inheritance ratio:* This metric is the fraction of bugs introduced into the mainline that are inherited by a downstream kernel branch. The intuition is that the mainline branch is the development branch that constantly introduces bugs (e.g., because of features commits or even bugs in patch commits). If a downstream kernel ports these bug-introducing commits from mainline, it will naturally inherit these bugs, which include a subset of mainline bugs (i.e., the ratio is always smaller than or equals 100%).

Methodology to compute patch delays. Recall that all patches in the Linux ecosystem will first appear in mainline, so to calculate

	Major version frequently rebase	Minor version rebase on stable	Minor version rebase on LTS	Group-port from LTS	Group-port from stable	Picking from mainline
Arch	✔	✔	✔			
Fedora	✔			✔	✔	
Oracle Uek-4					✔	✔
Oracle Uek-5 / 6 / 7				✔		✔
Ubuntu-16.04 / 20.04 / 22.04				✔		✔
Ubuntu-18.04				✔	✔	✔
SLE12-SP5 / 15- SP3 / 15-SP4					✔	✔
Amazon-4.14 / 5.4 / 5.10			✔			✔
Debian-10 / 11				✔		✔
Android-10 / 12 / 13				✔		✔

Table 1: Patch porting strategies for all measured distribution branches

the patch porting delay for each patch that appears in a downstream branch, we need to first determine which mainline commit it corresponds to.

Tracking patch origins for Linux stable/LTS branches: we observe that 99.3% of all the git commits in LTS branches contain references to an upstream commit in the forms like "`Upstream commit <hash>`", "`upstream <hash> commit`" in the git commit messages. This is supported by the official script [10] used by Linux LTS maintainers to port patches from mainline.

Tracking patch origins for Linux distributions: As mentioned earlier, distributions have three typical approaches to port patches: group-porting, picking, and rebasing.

For group-porting, we found two commonly used methods: `git merge` and the Ubuntu-specific method.

- `git merge` ports all commits from a specific stable/LTS branch up to a certain commit specified in the command. It will generate one merge commit with the title that usually indicates the latest point that it caught up with. For instance, `Merge tag 'v4.14.107'` indicates group-porting patches reached a specified point in the source branch, which is the version of v4.14.107. In our measurement targets, Oracle-UEK and SLE use this method.
- We find that Ubuntu takes a different approach to apply group-ports of upstream patches in a single commit, without using `git merge` [14]. We observe that such commits will contain links to the corresponding bug tracking pages on `launchpad.net` (*e.g.,* [25]), which contain detailed information about what patches have been ported and from where.

Rebase is performed using `git rebase`, which changes the entire base (*i.e.,* the fork point) of a branch from one upstream commit to another (there is the major version rebase vs. minor version rebase as mentioned previously). This also results in the porting of all commits between the old base and the new base. It is worth noting that `git rebase` will not introduce a new commit in the log. In fact, the log will essentially be a replica of the upstream to which it rebases. This makes it difficult to infer when the rebase occurs and compute the patch delays accordingly. Nevertheless, distribution maintainers always will add a followup commit right after the rebase to assign a unique tag and indicate the rebase has been completed. This commit (*e.g.,* with a title of `kernel-5.10.23-200.fc33`) usually indicates the mainline/stable/LTS version of the distribution from which it is rebased. This allows us to determine when rebase occurs, from which we can then compute the patch porting delay.

When Linux distributions pick patches directly from the mainline, each commit message will contain text referencing the corresponding commit's hash in mainline, such as "`cherry picked from <hash>`" and "`back-ported from <hash>`". Because such references appear in different areas from those that appear in LTS commit messages, this allows us to distinguish picked patches from group-ported ones.

Another issue is that we find some distributions pick features commits in addition to patches from mainline, which we need to exclude when computing their patch delay. For Ubuntu, we find that feature commits picked from mainline are managed by its own bug tracking system on launchpad.com, and labeled as "[Feature]" on the web page. For SLE, they have a separate tracking system for feature requests on jiri.suse.com and attach identifiers that start with "jscSLE" to denote that it is a feature commit. For Oracle-Uek, all commits picked directly from Linux mainline are labeled "OraBug" id in commit messages, indicating they are bug fixes. For other distributions, such as Fedora, according to our observations and direct communications with distribution maintainers, they only pick patches (and no feature commits) from Linux mainline.

Methodology to compute patch rate. For a given downstream kernel branch, we divide the total number of patches that appear in the downstream branch (e.g., no matter if they are group-ported, picked, or inherited due to rebase) by the time duration starting with the first patch appearance date and ending with the last patch appearance date. In our measurements, we typically choose the duration starting with the very first commit since the creation of a downstream branch, and ending with the last commit we observe at the time of writing. If the branch has ended its end of life, we effective computes the average patch rate over its entire lifetime.

Methodology to compute bug inheritance ratio. We define a mainline bug as a commit (either a feature or patch commit) that requires a later patch. As mentioned in the previous section, fixes tags in a patch indicate which previous commit is the buggy commit, so we use fixes tags to identify mainline bugs. To validate the accuracy of the fixes tag, we randomly selected 50 buggy commits based on the fixes tags that have reproducers, obtained from syzbot [21], a public continuous fuzzer for Linux kernels. We then automatically executed these reproducers against Ubuntu branches that contained

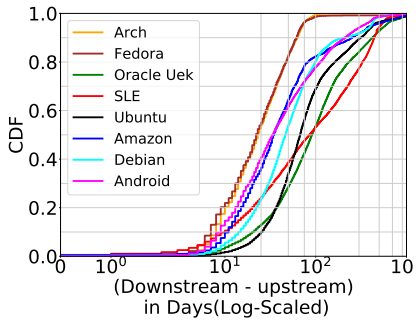

Figure 2: End-to-end delay for all patches in Linux distributions

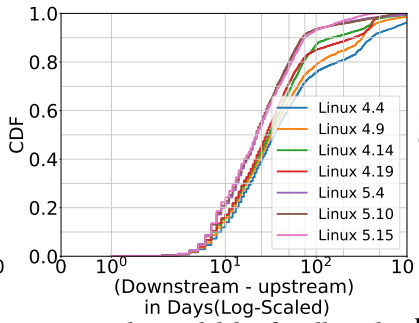

Figure 3: End-to-end delay for all patches in Linux LTS branches

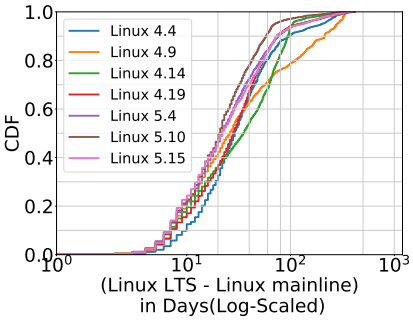

Figure 4: First year delay since the creation of Linux LTS branches

the buggy commits (but unfixed yet), and found that 45 of them were able to successfully trigger the bug. This indicates that fixes tags are generally reliable. We consider a bug inherited by a downstream kernel branch if the buggy mainline commit ported to the downstream branch, through either group-port, picking, or rebase. Note that we consider bugs inherited only after the downstream kernel branch is created. In other words, a downstream branch has zero bugs inherited at the time of its creation. The bug inheritance ratio is then computed by the total number of bugs inherited by a downstream branch divided by the total number of bugs in mainline, in a given time period. Again, in our measurements, we often choose the time window to be the lifetime of a downstream kernel branch to understand its average performance. We note that even though there is an underestimate of mainline bug count as not all patches have fixes tags, the ratio is a relative metric which makes it possible to compare the performance of different downstream kernel branches.

4 STRATEGY OF FREQUENTLY CHANGING BASES

As mentioned in Table 1, Arch and Fedora frequently change their bases from one major version to another (every two to three months). This is intuitively a very different strategy compared to staying on a single base for an extended period of time (i.e., years). In this section, we focus on the pros and cons of the frequent base changing strategy with respect to the metrics we defined.

In terms of patch delay, frequently changing the base is a good strategy because it automatically inherits all commits from the upstream, allowing the downstream kernel branch to keep up with the latest patches. Indeed, we can see in Figure 2 that Arch and Fedora have the lowest patch delay overall compared to other distributions. More specifically, almost all patches in Arch and Fedora are ported within 100 days. This is because all mainline patches are guaranteed to end up in an Arch or Fedora branch every time they rebase, which happens every two to three months. Of course, while they stay on the same base, they can port patches earlier as well, e.g., Arch would perform minor version rebases and Fedora would perform group-ports from stable/LTS.

Unfortunately, when switching bases from one major version to another (i.e., major version rebase), branches also inherit new bugs as well as patches from the mainline. For example, when a rebase occurs from v4.14.y to the beginning of v4.15, there are a large number of commits between v4.14 and v4.15 on mainline that are inherited (e.g., due to the development of v4.15-rc1, v4.15-rc2). All of these commits will be inherited automatically, which means the bug inheritance ratio is always 1. This can be a significant downside leading to stability and security risks. It is worth noting that minor version rebase is less of a concern because all commits in stable and LTS are patches by design. In contrast, for distributions that choose to stay on a single base, their bug inheritance ratio is much lower, as shown in Table 2. We will go into more detail in the next section regarding these measurements.

We suspect that the decision of frequently changing the base is driven by economics. Specifically, Arch and Fedora are community-backed distributions [6, 12] which have limited resources invested to maintain a stable branch on their own. In contrast, the other distributions are considered commercial distributions [6, 13]. For example, Amazon Linux 2 distributions are critical for AWS users where stability and security are critical[48]. As a result, these distributions prefer to stay on a single major version (e.g., v.4.14.y) for an extended period of time.

5 STRATEGY OF STAYING ON A SINGLE BASE

Given that frequently changing bases inherits too many bugs from the mainline, we now set that strategy aside and focus on the strategy of staying on a single base. Ideally, for this strategy, we would like to see that patches are ported as fast as possible, and none of the necessary patches are missed. We aim to analyze how far the downstream branches that follow this strategy, namely Linux LTS, Oracle-UeK, Ubuntu, SLE, Debain, Android and Amazon, are away from the ideal.

In particular, we will provide objective results with regard to individual metrics first and discuss unique behaviors in the patch porting practices for Linux distributions. In addition, we will analyze the factors that influence the results which will help set the stage for future improvements.

5.1 Patch delay

According to the methods described in the previous section, we can measure the patch porting delays from Linux mainline to various downstream kernel branches, including Linux LTS branches and Linux distributions. Figure 3 plots the patch delay between Linux mainline and Linux LTS branches and Figure 4 plots their delay for only the first year since the creation of Linux LTS branches, which

	By all ported patches							By picking from Linux mainline					
base	Linux LTS	Oracle-Uek	Ubuntu	Amazon	SLE	Debian	Android	Oracle-Uek	Ubuntu	Amazon	SLE	Debian	Android
4.1		1.7%/3.3						1.6%/2.8					
4.4	2.7%/8.4		3.5%/10.1						1.0%/1.5				
4.9	3.4%/10.4												
4.12					17.6%/35.1						17.4%/34.7		
4.14	4.5%/13.3	6.7%/15.9		5.1%/14.0				3.1%/4.2		0.3%/0.4			
4.15			5.1%/14.7						1.3%/1.8				
4.19	5.1%/16.1					3.2%/10.1	5.2%/16.05					0.7%/0.2	0.1%/0.1
5.3					14.3%/35.0						13.3%/33.0		
5.4	6.3%/19.9	11.4%/24.7	6.6%/18.9	6.9%/19.0			5.8%/18.5	5.4%/6.3	1%/1.3	0.3%/0.3			1.1%/1.2
5.10	8.8%/24.8			10.5%/24.7		6.7%/17.3	9.1%/23.0			0.5%/0.6		0.2%/0.21	1.4%/2.1
5.14					15.4%/49.0						9.7%/39.8		
5.15	13.1%/30.8	13.3%/30.3	12.3%/31.0					1.4%/1.5	2.7%/2.8				

Table 2: Bug inheritance ratio / patch rate for Linux LTS branches and six distributions that stay on a single base

shows that the three latest Linux LTS branches (i.e., Linux v5.4, v5.10 and v5.15) have noticeably smaller delays overall compared to older Linux LTS branches, indicating improvements over time. We do note that there are still about 5% of the patches are delayed for more than 100 days even in the latest kernel branches. For older LTS branches, the performance is significantly worse, with 15% to 25% of patches are delayed more than 100 days.

Next, we plot the delay for Linux distributions, as shown in the Figure 2. Interestingly, unlike the similar performance between Arch and Fedora, Oracle-Uek, SLE, Ubuntu, Android, Debian and Amazon exhibit substantially different delay profiles. Amazon and Android have the lower delay compared with the four distributions, with about 20% of patches delayed for 100 days or more. The worst delay goes to SLE, with about 20% of patches ported after 300+ days. Note that it is expected that the patch delays for distributions are generally larger than those of LTS branches, because there is always an extra delay for a patch to be ported from LTS to distribution.

Finally, we breakdown the patch delay for Linux distributions. There are two patch porting routes for distributions: (1) Linux mainline to Linux LTS branches, and then to Linux distributions; (2) Linux mainline to Linux distributions directly (through picking). Figure 3 already shows the patch delay between Linux mainline and Linux LTS branches. The patch delay between Linux LTS and Linux distributions is shown in Figure 5. It clearly shows Android is much better than other distributions on the delay for group-porting. About 78% of patches in Android are ported within one day. However, there is a long tail with much longer delays compared to Amazon and SLE. One possible reason is that the main maintainer of Android is also the main maintainer of Linux LTS branches [5, 16]. Second to Android, all of the patches in Amazon and SLE are group-ported from LTS within 25 days. However, about 20% of patches in Oracle-Uek are ported 100+ days after they appear in Linux LTS. Figure 6 plots the patch delay for picked patches with Linux mainline. Except Debian, all of the picked patches are delayed much longer than group-ported patches: 40% - 70% of picked patches are delayed 100+ days.

5.2 Patch rate

Patch rate is another critical facet of the patching porting performance. It is possible that some branch has small patch delays but it ports *only a small number of patches*. In other words, there may be many patches missing (or can be viewed as having an infinite delay).

We show the results in Table 2. Note that we compute the patch rate for "all patches" as well as "patches picked from mainline". This is because the picked patches from mainline can be viewed as extra work performed by distribution maintainers and worth highlighting separately.

There are several notable points from the table:

1) There is a general trend that newer branches (i.e., on newer major versions) have higher patch rate. This is expected as older branches (e.g., v4.4) diverge more and more from mainline as time goes by, and thus fewer patches are applicable. This is true for LTS branches and as well as most distributions (except for Oracle-Uek which is an outlier).

2) SLE has the highest patch rate among the six distributions — much higher than distributions that are based on a similar kernel major version. For example, SLE (v4.12) has 35.1 patches ported per day compared to Oracle-Uek (v4.14)'s 15.9 and Amazon's 14.

3) Regarding the patch rate for picked patches only, we can see that SLE is also the highest. In fact, it picks more patches per day compared to LTS. For example, for SLE (v5.3), its patch rate is 35 per day whereas LTS (v5.4) has a patch rate of 19.9 per day. Note that SLE's patch porting strategy is that it does group-porting initially from a stable branch and switches to exclusively picking from mainline after the stable branch reaches end-of-life. Effectively, SLE can be viewed as maintaining its own LTS branch.

4) Debian has the lowest patch rate among all distributions. For example, Debian (v5.10) has an overall patch rate of 17.3 compared to LTS (v5.10)'s 24.8 and Amazon (v5.10)'s 24.7. We will explain the reason behind this in §5.4.

5.3 Bug inheritance ratio

The ideal patch porting does not only require that distributions port patches, but also that there should be no new bugs introduced during the process. We list the bug inheritance ratio in Table 2 as well.

We summarize a few notable points:

1) There is a general trend that newer branches have higher bug inheritance ratio. This is expected as we can see generally high bug inheritance ratios correlate with higher patch rate. Assuming there are always buggy patches that introduce bugs themselves, the more patches we pick, the more likely we will inherit those bugs.

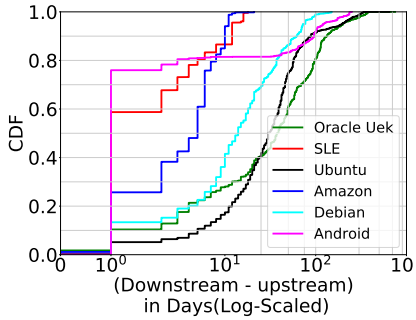

Figure 5: Delay for Linux distribution commits whose sources are LTS

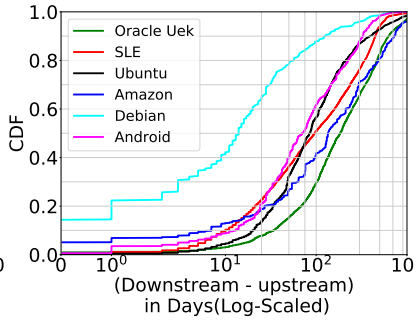

Figure 6: Delay for Linux distribution commits whose sources are mainline

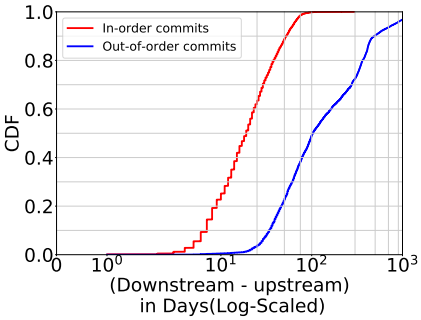

Figure 7: The comparison of in-order and out-of-order commits

2) The bug inheritance ratio is much higher in SLE branches than others, mainly contributed by the extensive picking from Linux mainline. This is likely again because of the substantially higher patch rate.

5.4 Unique patch porting behaviors of Distributions

As demonstrated above, six distributions exhibit varying results in three metrics, even though all of them choose to stay on the single base for a long time. This is due to various other unique (and sometimes subtle) behaviors in the patch porting practices.

SLE effectively maintains its own LTS branches, as previously mentioned, by following short-lived Linux stable branches and picking patches from the mainline directly. SLE appears determined to maintain high quality branches by aggressively selecting patches from the mainline. In fact, it selects significantly more patches compared to traditional LTS branches, showcasing a strong commitment in maintaining a reliable kernel. As a result, its patch rate is high (in fact, the highest), it the bug inheritance ratio is also high (an expected tradeoff). However, even though it ports many patches, its patch delays are larger than most other distributions, especially for patch delays of 100+ days, likely due to the cost of picking more patches.

Oracle-Uek and Ubuntu overall employ a very similar strategy that involves grouping-porting from Linux LTS and picking from the mainline, and has comparable patch rate and bug inheritance ratio. However, we have noticed significant periods of inactivity (i.e., idle periods), during which no group-porting takes place. This is the key reason behind the noticeably worse group-port delay and end-to-end delay compared to other distributions. We will discuss this phenomenon more later in §5.5.

Android and Linux kernels are generally considered to have a significant "gap" because Android kernels are known to have its custom features designed for mobile devices (by heavily modifying the Linux kernel). Intuitively, this should result in larger patch delays as Android kernel maintainers may need to maintain diverging copies of the same functionalities. However, we see that approximately 78% of group-ported patches in Android are delayed for only one day. The result may be counter-intuitive. However, we note that a key maintainer of Android also serves as the main maintainer of the Linux LTS branches [5, 16], which may help explain the timeliness of the majority of patches. We do note that the rest 22% of patches incur significantly larger delays compared to most other distributions, likely due to the gap. Finally, perhaps due to the same maintainer being involved in porting patches to both LTS and Android common, we observe fewer picked patches. In other words, the lack of maintainer diversity might have hurt Android common because there may be important mainline patches missed by LTS and also Android common.

Amazon uses minor version rebase to obtain patches from LTS. As mentioned before, group-porting and minor version rebases achieve the same goal of getting patches from LTS. This minor version rebase strategy is more feasible because Amazon has few picked patches from mainline. This makes it much easier because it just needs to port a small number of picked patches that are accumulated over time to the new base every time. The advantage of this strategy is that it enjoys a fairly small patch delays. The weakness of this strategy is that Amazon will naturally have a lower patch rate, potentially missing some important mainline patches that should have been picked.

Debian chooses to port a much smaller number of patches, i.e., very low patch rate, as mentioned before. Upon a closer inspection, we find that only when patches belonging to enabled kernel modules according to kernel configuration files are ported. As a result, its patch rate is the lowest. However, we argue this is generally a risky patch porting practice. This is because configuration options can change over time and it can lead to a spike of workload to port patches pertaining to newly enabled modules, leading to sub-optimal results. In addition, if others were to customize Debian by enabling additional modules, they will effectively end up with out-dated (and potentially more buggy) versions of those modules.

5.5 Factors Influencing the Results

After the comparative study, we now dive deeper into the results and attempt to identify factors that influence the results. We summarize them into a few points.

Lack of hints and large patches contribute to the long patch delays from mainline to LTS. We know that the patch porting from Linux mainline to Linux LTS is an important step, as most

	In order commits		Out-of order commits	
Linux LTS	Fixes or Cc tag	Big size	Fixes or Cc tag	Big size
4.4 LTS	76%	6.36%	39%	11.18%
4.9 LTS	75%	6.28%	34%	11.13%
4.14 LTS	73%	6.75%	37%	11.43%
4.19 LTS	73%	6.29%	40%	13.06%
5.4 LTS	76%	6.74%	36%	17.68%
5.10 LTS	78%	7.55%	28%	24.72%
5.15 LTS	81%	8.54%	26%	29.05%

Table 3: The comparison of in-order and out-of-order commits

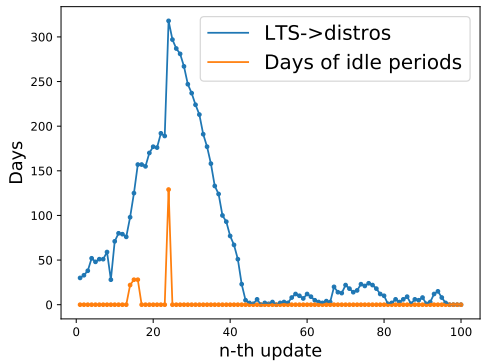

Figure 8: The delay changes for Ubuntu-18.04 over the time

distributions would have to wait for the patches to appear in LTS before they can port them. In general, we know it is a challenging task for LTS maintainers to identify and pick patches from mainline, as there are so many mainline commits that maintainers need to scan (a few hundred daily). Therefore, it is possible that a mainline commit is mistakenly skipped and then picked up at a later point after the community realizes its importance. We call such commits, out-of-order commits. To be more precise, commit A is "out-of-order" if there exists a later commit B in Linux mainline, whereas A is ported later to Linux LTS than B. If A is ported to Linux LTS before all later Linux mainline patches, B, we consider A to be "in-order".

According to our measurement of all LTS branches, we observe that surprisingly 30% of patches are ported out-of-order. Figure 7 compares the delays for patches ported in-order vs. out-of-order. We can see clear differences between in-order and out-of-order commits. All in-order commits are ported relatively quickly (under 80 days), whereas 50% of out-of-order commits have delays longer than 100 days. In addition, about 78% of out-of-order commits incur delays of 50+ days.

We explore the factors that could have led to the out-of-order commits. As mentioned in §2, there are two types of tags provided in the commits themselves that are designed to make the job of maintainers easier. Fixes tags can help maintainers make patch porting decisions. For instance, if a commit given in the fixes tag goes back to somewhere before Linux v4.4 LTS is forked, then we can infer that the v4.4 LTS branch would have inherited the bug, therefore requiring the the patch. Cc stable tags indicate that downstream maintainers should pay more attention to the patch. Some CC stable tags also have additional kernel version indicators, such as `Cc:<stable@vger.kernel.org>#4.14.x` and `Cc:stable@vger.kernel.org#v4.4+`, to suggest which LTS branch (*e.g.,* v4.14 and branches that after v4.4) may need the patch. The version tags can save time for maintainers.

According to various sources from maintainers [15, 17, 38] and our personal outreach to the Linux community, fixes tags and Cc stable tags are indeed consulted in practice when determining whether a mainline patch will be ported. We then measured the correlation between the presence of tags and the patching behaviors, as shown in Table 3. Indeed, there is a strong correlation. For those patches ported in-order, they are much more likely to contain either fixes or Cc stable tags; on the other hand, for out-of-order commits, the fractions with such tags are much lower.

But there are still about 40% of LTS commits that do not have any tags. According to maintainers [15, 17, 38], commits without tags will go through the AUTOSEL bot, which is a neural network model introduced in Linux in 2017 to identify if a commit is a bug fix or feature commit; and then maintainers manually check whether the patch is necessary for specific Linux LTS branches. Since the AUTOSEL and the manual analysis by maintainers are blackboxes to us, we can only hypothesize what factors might influence the decision of whether to port a patch. In particular, we consider the patch size as a candidate factor. From the official document [9], Linux LTS/stable branches do not accept patches larger than 100 lines. We therefore pick 75 lines as the threshold in determining "big patches". We hypothesize that larger patches are generally more complexity and more difficult to port, especially when the LTS/stable has diverged from the mainline. Indeed, Table 3 shows that the percent of big patches in out-of-order commits are twice or three time as that in in-order commits(we also choose 50,60,70,80,90 lines as "big patches", but the result is the same).

Periods of inactivity of group-porting affects distribution's patch delays significantly. Now, we explore the patch porting behavior of distributions. As mentioned earlier, a primary source of patches is through group-porting from Linux LTS. It is natural to take advantage of the efforts by LTS maintainers who already determined which mainline patches are necessary to port for the base. In other words, by following an LTS branch, downstream kernel can shield themselves from the large stream of mainline commits.

As mentioned earlier and shown in Figure 5, Ubuntu and Oracle-Uek perform group-porting and they both have fairly poor delay profiles. Interestingly, after we investigate the delay patterns, we observe some fairly anomalous data points. Taking Ubuntu 18.04 as an example, 20% of group-ported commits have a delay longer than 150 days. Furthermore, the long-delay commits are not uniformly distributed. Instead, there is a strong temporal pattern as shown in Figure 8. The X-axis shows the n-th group-porting events (each event corresponds to a group of commits) since the creation of the Ubuntu-18.04 branch. The Y-axis shows the patch porting delay (for the line of "LTS to distributions") and the number of days between two group-porting events (a.k.a., idle periods). Upon a

distributions	# of LTS commits	# of picked commits	Picked & in upstream LTS		Picked & not in upstream LTS		
			First in Distro	First in LTS	Total num	# of commits with fix tags	# of commits necessary for LTS
Oracle-Uek5 (v4.14 LTS)	25,663	7,070	570	315	6,185	798	192(24.1%)
Oracle-Uek6 (v5.4 LTS)	22,069	7,153	220	302	6,631	934	200(21.4%)
Oracle-Uek7 (v5.15 LTS)	13,790	581	78	98	405	42	14(33.3%)
Ubuntu-16.04 (v4.4 LTS)	18,974	2,830	226	525	2,079	147	39(26.5%)
Ubuntu-18.04 (v4.14&4.19 LTS)	50,771	2,939	569	516	1,854	297	104(35.0%)
Ubuntu-20.04 (v5.4 LTS)	22,069	12,67	94	149	1,024	122	39(32.0%)
Ubuntu-22.04 (v5.15 LTS)	13,790	859	84	183	592	51	11(21.6%)

Table 4: Comparison between picked commits and commits in corresponding LTS branches

closer look, there are two clusters of "idle periods". The first cluster contains three small idle periods (around 25 days each). The second contains a huge 144-day idle period where no group porting activity occurred. These contribute to the massive delays in porting patches to Ubuntu, especially for the patches that have delays between 150 and 300 days.

To understand what happened behind the scene, we reached out to Ubuntu maintainers directly. Interestingly enough, we were told that those "idle periods" actually corresponded to a period of time where the company was busy with other important (but unspecified) tasks. According to our observation, during such periods, there seems to an increased focus on picking patches for CVEs (one possible task) — 1.7 CVE patches per day were picked during the idle period vs. 0.58 per day during the non-idle periods.

Surprisingly, we find that the idle periods are much longer in Oracle-Uek than Ubuntu: the longest idle periods in Oracle Uek-5, 6, and 7 are 233, 126, 112 days, respectively. In contrast, the longest idle periods in Ubuntu-16.04, Ubuntu-18.04, 20.04, and 22.04 are 116, 144, 53, and 58 days, respectively. This explains why the group-port delays (and the end-to-end patch delays) for Oracle-Uek are even worse than Ubuntu.

Note that SLE does not have the same problem because it group-ports patches only from a short-lived stable branch (for two to three months), so it is much less likely to incur such long idle periods.

Many patches in distributions should have been ported to LTS. Since we know that distributions often pick patches directly from mainline and can have higher patch rate than LTS overall, does it mean that distributions actually offer better-quality kernels than LTS? Intuitively, that should be the case because it is the distribution kernels that are user-facing (not the LTS ones). However, it would imply that there are patches missed or delayed by LTS (and they are necessary and should have been ported).

Indeed, as mentioned in §2, distributions have their own bug tracking systems where they have external signals (e.g., from users) that trigger the patch picking. In addition, we find that CVE is also a focus for distributions (as they are responsible for the security of their customers). In particular, maintainers usually attach a CVE ID to indicate that the patch fixes a known security vulnerability. Interestingly, we note that the picked CVE patches appear in distributions 74.2 days earlier than LTS on average; even if the picked CVE patches are later than LTS, it is only 16.7 days later on average.

To get a more comprehensive picture, Table 4 shows the picked commits by distributions and their relationship with LTS commits. We summarize three main points here. First, the number of picked commits in distributions is generally much smaller than the number of Linux LTS commits. Second, we note that, in the cases where the picked commits appear in both the LTS and distribution, many of the picked commits appear first in distributions and then in LTS, which is a strong evidence that distributions do pick useful patches earlier than LTS. Finally, we note that a large fraction of picked commits are not even ported into LTS. We note that only a small fraction of them have tags, which is another evidence that that such pickings are driven by other external signals. More importantly, for patches with fixes tags, we find that roughly 20% to 30% of the patches should have been ported to LTS, because the buggy commits (given in fixes tags) do appear in LTS, which shows that corresponding bugs are introduced in LTS, and LTS is required patches to fix those bugs. Interestingly, the remaining 70% to 80% are cases where the buggy commits themselves are not in LTS. Upon a closer inspection, this is because the buggy commits were inherited by distributions exclusively through their picked patches. This is the same phenomenon we reported earlier about higher bug inheritance ratio as a result of higher patch rate.

Overall, we conclude that many patches picked by distributions should have been ported to LTS as well. In addition, if the patch delays between mainline and LTS improve, that will benefit the entire ecosystem.

6 IMPROVEMENT OF THE PROCESS

In this section, we summarize a few suggestions based on the insights gained from our measurement study. To improve the patch porting process, we believe the responsibility is not only on Linux LTS maintainers but also the whole Linux community. Specifically, we propose the following suggestions.

Patch authors: increase the awareness and willingness to attach tags. According to the above analysis, tags associated with patch commits play a critical role in the patch porting process. The Cc stable tags and fixes tags directly inform LTS maintainers on whether a commit is a patch and a patch is necessary for stable or LTS branches. As we have shown, the lack of tags correlates directly with longer patch porting delays. Interestingly, the patch submission in Linux is heavily decentralized. Anybody can submit patches and there is an official document outlining the desired patch submission process [20], including the suggestion of including various tags. However, according to [19], about 300 authors submit their first change into Linux mainline in every kernel release. They

don't all know the rules. This requires efforts to educate the developers and maintainers. For example, we note that sometimes patch reviewers will remind patch authors to add these tags but they are quite inconsistent.

Of course, generating these tags will requires some more effort on the patch authors, e.g., figuring out which commit introduced the bug. Therefore, it calls for automated reasoning tools that can generate these tags.

Maintainers: automated patch relevance analysis tools. For each Linux mainline commit, there are two indispensable steps for Linux LTS maintainers: (1) identify whether it is significant enough, e.g., an important bug fix, and conform to rules in [9]; (2) check whether the commit should be ported into one or more specific stable/LTS branches.

Currently, there are already several machine learning tools [3, 32, 33] for the first step, the best accuracy is as high as 90%. LTS maintainers also have already been using such tool, such as AUTOSEL [3].

Nevertheless, there are no automatic tools for the second step. The second step is required as not all stable/LTS branches may be affected by a bug. It is also more difficult. Linux LTS maintainer usually have to manually understand the patch and see if the patch is applicable to a specific stable/LTS branch. We believe the automation of this step can continue to help maintainers further.

Distribution maintainers: mitigating regression. As we see, a good fraction of patches ported by distributions (whether through LTS or picked directly from mainline) end up being buggy patches introducing new bugs. In other words, there are significant regression issues in patches for Linux mainline. In particular, the bug inheritance ratio shows that a decent fraction came from picked patches. Interestingly, we note that there is a chance to mitigate such issues. The observation is that it is possible that the buggy patches already have fixes in Linux mainline when distributions picked the buggy patches themselves. Given that picking in generally is already substantially delayed, the likelihood of a patch for the initial picked patches being fixed on mainline is therefore quite high. As a concrete example, in Oracle-Uek, we find that about 60% - 70% of buggy patches introduced by picking are fixed within one day. This is likely because Oracle-Uek looked ahead to see if these picked patches on mainline have corresponding fixes already. If so, they ported those fixes as well. However, other distros do not seem to have adopted this method. Using SLE as an example, only 3.2% - 6.4% of picked buggy patches are fixed within a day. We suggest this simple trick of look ahead on mainline to help mitigate the risk of regression caused by extensive picking (i.e., reducing the bug lifetime).

Community: cross validation metadata tracking system for all LTS and distributions. As shown in Table 4, there are about 3.1% - 19.4% of picked commits that are ported into distributions before LTS. And about 30% of LTS-missed picked patches could possibly also be necessary for Linux LTS branches. Also, the picking behaviors are drastically different even for same-based distributions. Therefore, it is useful for distribution and LTS maintainers to review each other's patches to minimize missing patches. Furthermore, we suggest that the Linux community to collectively build a unified bug tracking system so that they can all benefit and quickly port important patches.

7 RELATED WORK

Linux patches measurement. Zheng *et al.* [61] have recently conducted a measurement on the Android kernel patch ecosystem, which is the most relevant work. They focus on the patch porting relationships in the Android branches, including measuring the patch porting delay, with an emphasis on analyzing binary-only OEM Android kernel images (e.g., Samsung). In our study, we focus on a very different target: downstream Linux distributions with public git repositories, without which it is difficult to infer the exact patch porting strategies. Finally, the measurement is limited to only CVE patches publicized on Android security bulletin, and focus on a single dimension of patch delay, whereas we additional derive insights from other metrics such as patch rate and bug inheritance ratio. Furthermore, their analysis for the patch delay lacks the knowledge of key data such as fixes tags and Cc tags, which we showed to be critical in assisting patch porting.

There are also other related research measuring patches and vulnerability life cycles. For example, Li *et al.* [39] performed a large-scale measurement on CVE dataset across different various open-source projects, including Linux kernel. However, the study does not look at the Linux kernel as an ecosystem. Rather, the measurement focuses on Linux mainline only, e.g., when a bug was introduced in mainline and when it is fixed in mainline. Ozment *et al.* [42] measured whether security improves when software ages (using OpenBSD as a case study). Farhang *et al.* [29] measured the life cycle and timeline of patches in Android. Shahzadet *et al.* [47] studied various dimensions of vulnerabilities such as risk level. Finally, Jones *et al.* [8] conducted an extensive study on the impact manufacturers, carriers, and end-users have on the rollout of Android security updates and OS upgrades.

Automated patch analysis. There exist several studies [33, 50, 55] that use machine learning to determine whether a Linux mainline commit is a patch and should be ported to stable/LTS branches, with varying degrees of success.

Alternatively, one can recognize whether a commit is a security patch and therefore should be considered for back-porting. For example, there have been studies using machine learning consuming commit messages and bug reports to achieve the goal [27, 28, 31, 40, 44, 45, 51, 56, 63]. Another recent work by Wu *et al.* [57] takes a different route to analyze the bug semantics in a patch.

In addition, bug localization techniques[34, 36, 37, 43, 46, 52, 53, 59, 62] can can automatically locate a potential bug in a software, which can help port more necessary patches. But they require bug reports which are unfortunately not available for the majority of the Linux mainline patches. Wen *et al.* [54] explored the relationships between bug introducing commits and bug fixing commits.

Finally, there are also recent work on determining whether a given patch has been applied to a specific target, for both source [58] and binary targets [35, 60]. However, they cannot determine whether a patch really needs to be ported to the target.

8 CONCLUSION

In this work, we have performed a deep investigation into Linux kernel ecosystem to understand the patch porting strategies in three metrics. We uncover many interesting findings on the current practices of patch porting, including the causes of sub-optimal results. Finally, we also provide suggestions on how to improve the patch porting process.

REFERENCES

[1] Active kernel releases. https://www.kernel.org/category/releases.html.
[2] Android Common Kernels. https://source.android.com/docs/core/architecture/kernel/android-common.
[3] Automating stable-kernel creation. https://lwn.net/Articles/701304/.
[4] [Bug]SKL-H boot hang when c8+c9+c10 enabled by intel_idle driver. https://bugs.launchpad.net/intel/+bug/1559918.
[5] Common Android Kernel Tree. https://android.googlesource.com/kernel/common/.
[6] Community vs. Commercial GNU/Linux Distributions. https://www.datamation.com/open-source/community-vs-commercial-gnu-linux-distributions/#:~:text=Community%2Dbased%20distributions%20like%20Debian,as%20Windows%20and%20OS%20X.
[7] Comparing Centos Linux and CentOS Stream. https://www.centos.org/cl-vs-cs/.
[8] deploying android security updates: an extensive study involving manufacturers, carriers, and end users.
[9] Everything you ever wanted to know about linux-stable releases. https://www.kernel.org/doc/html/latest/process/stable-kernel-rules.html.
[10] Git format scripts when propagating patches to Linux LTS. https://git.kernel.org/pub/scm/linux/kernel/git/bwh/linux-stable-queue.git/tree/scripts/git-format-patch-for-backport.
[11] in Ubuntu16.10: Hit on Call traces and system goes down when transactional memory tests are running in 32TB Brazos system. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1606786.
[12] Introduction of Fedora. https://distrowatch.com/table.php?distribution=fedora.
[13] Is Ubuntu an enterprise Linux distribution? https://ubuntu.com/blog/is-ubuntu-an-enterprise-linux-distribution#:~:text=Yes.,publisher%20and%20maintainer%20of%20Ubuntu.
[14] KernelBugFixing. https://wiki.ubuntu.com/Kernel/Dev/KernelBugFixing.
[15] Lessons Learned Maintaining a Stable Tree. http://events17.linuxfoundation.org/sites/events/files/slides/stable.pdf.
[16] Linux Kernel Stable Tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/.
[17] Maintaining stable stability. https://lwn.net/Articles/825536/.
[18] OpenPower: Some multipaths temporarily have only a single path. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1696445.
[19] Some 5.16 kernel development statistics. https://lwn.net/Articles/880699/.
[20] Submitting patches: the essential guide to getting your code into the kernel. https://docs.kernel.org/process/submitting-patches.html.
[21] Syzbot. https://syzkaller.appspot.com/upstream/fixed.
[22] The lifecycle of a patch. https://www.kernel.org/doc/html/v4.14/process/2.Process.html#the-lifecycle-of-a-patch.
[23] The LWN.net Linux Distribution List. https://lwn.net/Distributions/.
[24] UbuntuKVM guest crashed while running I/O stress test with Ubuntu kernel 4.4.0-47-generic. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1659111.
[25] Xenial update: v4.4.262 upstream stable release. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1920221.
[26] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser. How long do vulnerabilities live in the code? a {Large-Scale} empirical measurement study on {FOSS} vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, 2022.
[27] D. Behl, S. Handa, and A. Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 294–299. IEEE, 2014.
[28] K. K. Chaturvedi and V. Singh. Determining bug severity using machine learning techniques. In *2012 CSI sixth international conference on software engineering (CONSEG)*, pages 1–6. IEEE, 2012.
[29] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. *arXiv preprint arXiv:1905.09352*, 2019.
[30] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, 2006.
[31] K. Goseva-Popstojanova and J. Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International conference on software quality, reliability and security (QRS)*, pages 344–355. IEEE, 2018.
[32] T. Hoang, H. J. Kang, D. Lo, and J. Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.
[33] T. Hoang, J. Lawall, R. J. Oentaryo, Y. Tian, and D. Lo. Patchnet: a tool for deep patch classification. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 83–86. IEEE, 2019.
[34] X. Huo, M. Li, Z.-H. Zhou, et al. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, volume 16, pages 1606–1612, 2016.
[35] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.
[36] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.
[37] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229. IEEE, 2017.
[38] S. Levin. Safeguards in the Stable Kernel Process. https://www.youtube.com/watch?v=_GLtZg5IjzE.
[39] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.
[40] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355. IEEE, 2008.
[41] Oracle. ELSA-2021-9084 - Unbreakable Enterprise kernel security update. https://linux.oracle.com/errata/ELSA-2021-9084.html.
[42] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, pages 10–5555, 2006.
[43] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 621–632, 2018.
[44] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi. Deep neural network-based severity prediction of bug reports. *IEEE Access*, 7:46846–46857, 2019.
[45] N. K.-S. Roy and B. Rossi. Towards an improvement of bug severity classification. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 269–276. IEEE, 2014.
[46] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.
[47] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.
[48] SLE. Amazon Linux 2. https://aws.amazon.com/amazon-linux-2/?amazon-linux-whats-new.sort-by=item.additionalFields.postDateTime&amazon-linux-whats-new.sort-order=desc.
[49] SLE. SLE bug tracking example. https://bugzilla.suse.com/show_bug.cgi?id=1199665.
[50] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. ICSE'12.
[51] J. P. Tyo. *Empirical analysis and automated classification of security bug reports*. West Virginia University, 2016.
[52] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63, 2014.
[53] M. Wen, R. Wu, and S.-C. Cheung. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–273. IEEE, 2016.
[54] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 326–337, 2019.
[55] Y. Wen, J. Cao, and S. Cheng. Ptracer: A linux kernel patch trace bot. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1210–1211. IEEE, 2019.
[56] D. Wijayasekara, M. Manic, and M. McQueen. Vulnerability identification and classification via text mining bug databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, pages 3612–3618. IEEE, 2014.
[57] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.
[58] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi. MVP: Detecting vulnerabilities using patch-enhanced vulnerability

signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182. USENIX Association, August 2020.

[59] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

[60] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.

[61] Z. Zhang, H. Zhang, Z. Qian, and B. Lau. An investigation of the android kernel patch ecosystem. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[62] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.

[63] Y. Zhou and A. Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.