Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department of Computer Science

SEBASTIAN DUDA

MASTER THESIS

ANALYSIS OF IGNORED PATCHES IN THE LINUX KERNEL DEVELOPMENT

Submitted on 20 December 2019

Supervisors:
Prof. Dr. Dirk Riehle, M.B.A.
Maximilian Capraro, M.Sc.
Professorship for Open Source Software
Faculty of Engineering, Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 20 December 2019

License

Erlangen, 20 December 2019

Abstract

The importance of Linux in industry increases continuously with the ascending variety of products running Linux based software. Use cases are not only limited to consumer applications but also high-performance computer and business-critical control technology. The usage of software in critical application requires a certification (e.g. IEC 61508), which demands the documentation of and the compliance with the software development process. As a result, the analysis of the Linux kernel development process becomes equally more valuable.

The Linux kernel development process is well understood, documented, and researched. However, some actions of contributors or maintainers do not comply with the process. One of these actions is ignoring patches. Ignoring a patch means that the patch is neither answered nor accepted by the developers. There is no analysis of this phenomenon yet.

In this thesis, we conducted statistical analyses of the ignored patch phenomenon to answer our research questions. In the analyzed time frame (release `v3.0` to `v4.20`), 18k of 792k patches (2.3%) were ignored. The ratio of ignored patches decreased over time. We detected two clusters (minor contributions, and automatically created patches) of patches that make up most of the ignored patches. There were no other statistically significant abnormalities of the ignored patches. Based on the analyses, we were able to show that there are indicators of discrimination against certain groups. We further recognized the trend that larger subsystems and lists are ignoring relatively fewer patches.

To conduct the analyses, we created a dataset of the patches sent to the Linux kernel. We published the dataset for further analyses. The dataset is extracted from the available mailing lists and Torvalds' git repository. We conducted spot tests to validate the correctness and integrity of our dataset. There is a small volume spot test for trivial measurements like the size of a patch. Besides, there is a high volume spot test to test our developed is-ignored metric.

Contents

1 Research

1.1 Introduction

Linux is an open source operating system [149, 64, 152]. Open source software (OSS) is “software released under a license that permits the inspection, use, modification, and redistribution of the software’s source code” [48]. OSS becomes increasingly important [163]. It is estimated that in 2016, 95% of all commercial software products contain OSS components [63]. The success of open source is hardly surprising since the use of OSS and components comes along with several benefits [132, 62].

Benefits of OSS are:

Public Code Since the code is openly available, everyone can assess, extend, and enhance the software.

Fewer Costs Due to the reuse of source code and the shared development effort, the cost of developing software can decrease significantly [137].

More Secure The openness of all artifacts allows experts to take a more sophisticated look during the assessment. Furthermore, each company can implement security fixes on their own and share them. Thereby, the time until a patch is distributed by the vendor can be avoided [10].

More Reliable Raymond published Linus’ Law “Given enough eyeballs, all bugs are shallow” [129]. Al Marzouq says that the “virtually unlimited number of developers [can] suggest both bug fixes and enhancements” [10]. Furthermore, the sophisticated review process in open source projects is a well working quality gate that filters low-quality contributions [7].

No Vendor Lock-in Due to the openness of the source code, there is no vendor lock-in. If a company does not want to or cannot cooperate with a vendor anymore, the company can either use the current source code and continue developing it on its own or easily migrate to another available product [137].

In 1991 Linus Torvalds released the Linux kernel version 0.02 [22]. Back in the days, Linus developed Linux as a hobby. He announced his work as “just a hobby, won’t be big and professional like gnu” [153].

In 1994, Linus published the first stable version of Linux (1.0) [98]. Even before the first stable release, over 30 developers [101] worked together on Linux. At the time, Linux consists out of 261 files[1] The collaboration model was quite informal compared to today's model.

In December 1997, the first publicly available mail was sent on the Linux Kernel Mailing List (LKML):

> Author: MOLNAR Ingo `mingo@chiara.csoma.elte.hu`
>
> Date: Sat Dec 6 06:16:59 1997 +0000
>
> Subject Re: Bug - Re: memleak 'DeLuxe' detector, 2.0.32, patch

This is one of the two backbones of the collaboration model [103]. The LKML is the main mailing list for all kernel developers to exchange patches and information. Besides the LKML, there are several other mailing lists, each for a group of subsystems of Linux.

In 1997, Elizabeth Coolbaugh and Jonathan Corbet [104] started a webmagazine, the `Linux Weekly News` (LWN). In the first years, a summary of the LKML and Linux related news was published once a week. Today, besides the weekly news, there are additional articles about Linux related news, like presentations on conferences (e.g. `https://lwn.net/Articles/804511/`).

In April 2005, Linux started to use Git for source code management which is the second backbone of the collaboration model [103]. In Linus' Linux git repository [151], one can find the current source code of Linux and all previous releases' source code. Since `v2.6.13-rc3`, the git repository enables the tracking of all changes made.

Today, Linux grew big and became the operating system running the world. Linux, and open source in general, became so important that the German government changed the copyright law and added the so-called 'Linux-Klausel' [2][100]. According to Jonathan Corbet, Linux "[…] is a robust, efficient, and scalable solution for almost any situation" [40]. Linux based software is used in everyday devices like smartphones [13], refrigerators [21], in planes [161], in space [75, 97], in the finance sector [157, 158], in all top ten supercomputers (June 2019) [148], and even in the data centers of the former opponent [70, 107] Microsoft [18]. For the last release (`v5.3`) 26812 developer collaborated[3] working on 65261 files[4]. (Almost) all major tech companies work on the Linux kernel:

[1]`find . -type f | wc -l`

[2]The "Linux-Klausel" enables the creator to share a good free of charge; beforehand, according to the copyright law from 2000 a fee was required.

[3]`git shortlog -sne | wc -l`

[4]`git ls-tree -r --name-only HEAD | wc -l`

- Amazon [11, 167]
- AMD [12, 91]
- IBM [15, 172]
- Intel [53, 143]
- Microsoft [76, 168]
- Oracle [32, 83]
- RedHat [8, 162]
- Samsung [14, 27]
- Suse [17, 16]

The growth of the project demands to define a process of how to collaborate. Today, the process documentation can be found in the kernel's repository [99]. In addition, there is more informal documentation from the kernel community [68, 136, 6, 85].

The enormous use of OSS, induce research in this area. Of course, Linux is in the focus of researchers, as the development of Linux was open and traceable; one of the benefits of OSS [160]. As a result, there are many papers regarding open source projects, and specifically regarding Linux [109, 20, 31, 58]. Improvements of the open source processes will affect the corresponding projects, the companies using the software, and the companies developing the software. Additionally, findings in the open source development process research are applicable and helpful for companies developing proprietary software by using inner source.

> Inner source (IS) is the use of open-source software development practices within an organization. The organization still develops proprietary software but internally opens up its development. [24]

Because of the similarity of the open and the inner source process, the findings are applicable. Without the loss of generality, this case study design can be applied to inner source projects if given requirements are fulfilled (see section 1.3.1.1). This can lead to specific insights for the company, which help to improve their inner source program.

During the exploration of the ignored patch phenomenon, we noticed there is no research about ignored patches yet (except two talks by Sang [141, 140]). The analysis of ignored patches can lead to deeper, more sophisticated insights into the software development processes. Furthermore, better knowledge of ignored patches enables us to improve the performance of the OSS development process. This can be achieved by either decreasing the number of patches being ignored due to the created awareness, or by identifying patches to be ignored on purpose.

In addition to performance aspects, analyzing ignored patches can contribute to solve or to identify regulatory issues like discrimination of certain groups. On the one hand, these groups can be socio-ethnic groups (this could contradict Perens' open source definition). On the other hand, these groups can be discriminated due to their employers (e.g. China–United States trade war).

In this thesis, we conduct a two-step approach. First, we generate a generic dataset of patches from the mailing lists sent in a given time frame. Second, we conduct the analysis tackling the research questions. Further the thesis is composed using the terminology as used in the Linux kernel project. A description of terms can be found in section 1.2.2.

In the rest of the section, we discuss the research questions, and the contributions of this thesis. In section 1.2, we discuss the related work and theoretical concepts, followed by section 1.3 describing the methodology used in the thesis. Section 1.4 describes the generic dataset. The analysis of the data is placed in section 1.5. Afterwards we discuss, the validation in section 1.6, and the limitation of the research in section 1.7. The last section (1.8) concludes the thesis with a recapitulation.

1.1.1 Research Questions

We want to get an understanding of how we can use the ignored-patches-metric to gain insights into the software development process. In this thesis, we will tackle this overarching question by conducting an exemplary case study with the Linux kernel. In the case-study, we have the following research questions (RQ1-4):

RQ1 : How many patches are ignored in the Linux kernel development? And how is the rate of ignored patches developing over time?

RQ2 : What are the unique characteristics of ignored patches in the Linux kernel development?

RQ3 : What discrimination is taking place in the Linux kernel development by ignoring patches?

RQ4a : What is the difference between mailing lists that can be derived from the ignored patches data?

RQ4b : What is the difference between subsystems that can be derived from the ignored patches data?

1.1.2 Contributions

This thesis claims following contributions:

- We present an extensive and sophisticated dataset of the patches submitted between the 19th of May 2011 (release of Linux kernel version 2.6.39) and the 23rd of November 2018 (release of kernel version 4.20).
- We discuss example source code how this dataset can be used to generate a datasets of the authors/mailing lists/subsystems for further analysis.

- All measurements are validated in two spot tests to show the exactness and reliability of the data.
- A definition when a patch is ignored.
- A broad analysis of the ignored patches answering the research questions:
  - $18k$ of $792k$ (2.3%) of the patches are ignored with a decreasing trend.
  - Automatically created and small contributions are more likely ignored.
  - There is indication that some groups of contributors are discriminated.
  - Larger subsystems/mailing lists ignore relatively fewer patches.

1.2 Theoretical Concepts and Related Work

In this chapter, we discuss the theoretical concepts our research is based on. First, we explain the theoretical framework. Second, we define some terms we use in our research. Terms defined in section 1.2.1 are not defined in section 1.2.2 again. Third, we will discuss the related work to get an insight into the state of the art research and to note what our research additionally contributes.

1.2.1 Definitions

1.2.1.1 Open Source

In literature, one can find multiple definitions of the term open source. `opensource.com` [114] defines open source as:

> The term "open source" refers to something people can modify and share because its design is publicly accessible.

Besides the definition of open source, the term `the open-source way` [114] is defined as follows:

> The open source way is a set of principles derived from open source software development models and applied more broadly to additional industries and domains. Opensource.com exists to share how the open source way can change our world in the same way the open-source model has changed software.

`opensource.com` defines five principles to follow: Transparency, Collaboration, Release early and often, Meritocracy, and Community. The principles are based on the essay `The Cathedral and the Bazaar` by Raymond [129].

Perens [118] defines open source in his publication "The Open Source Definition". Based on `The Debian Free Software Guidelines` [147], he defined ten criterions to follow [115] (rephrased):

1. Free Redistribution of the Product
2. Free Redistribution of the Source Code
3. Free Redistribution of the Derived Works
4. Integrity of The Author's Source Code
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavor
7. Distribution of License
8. License Must Not Be Specific to a Product
9. License Must Not Restrict Other Software
10. License Must Be Technology-Neutral

Crowston defines open source software (OSS) as "software released under a license that permits the inspection, use, modification, and redistribution of the software's source code" [48]. All definitions have in common that the source code has to be publicly available, and it is allowed to derive and redistribute the code.

1.2.1.2 Linux (Kernel)

Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. [9]

1.2.1.3 Linux Development Process

Linux is a complex product developed by people distributed around the globe. As a result, a process which defines how the contributors interact and collaborate is required. In this section, we will elaborate on the contribution process and the process of how new modules (e.g. drivers) are added.

Contributing In this section, we want to show the process of contributing source code to the Linux kernel. When contributing to the Linux kernel, one developer has to follow the steps which can be found in the kernel documentation [94, 30] and the LKML FAQ [67].

Contributing to the Linux Kernel is an iterative process [4].

1. A developer creates/improves a patch
2. The developer sends it to the most specific subsystem-maintainer(s) and the LKML
3. The maintainers (and additional reviewers) review the patch and give feedback; if the patch does not fit the quality requirements goto step 1
4. The maintainer signs-off the commit and adds it to the related subsystem; if the maintainer is Linus the process is finished, otherwise continue
5. The maintainer asks the superior maintainer to pull the collection of changes; goto step 3

In the Linux kernel, there is a hierarchy of subsystems. Corbet [41] describes this hierarchy in his article “Patch flow into the mainline for 4.14”. In addition, when calling the `get_maintainers.pl`-script one gets all subsystems a file is related to. For example, when calling (see Appendix B):

```
$ ./scripts/get_maintainer.pl --sections drivers/net/wireless/intersil/orinoco
```

One can see how the subsystems are hierarchically organized:

1. **The Rest** - The root subsystem
2. **Networking Drivers** - The overarching network subsystem
3. **Networking Drivers (Wireless)** - The wireless network subsystem
4. **Orinoco Driver** - The driver-specific subsystem

To trace the contributions of reviewers, testers, and others tags were introduced. The tags are added to the patch’s message. Today, there is a sophisticated choice of tags to use for different kinds of contributions:

Signed-off-by The “tag indicates that the signer was involved in the development of the patch, or that he/she was in the patch’s delivery path.” [96] The tag is used in two scenarios. First, all developers who worked on a patch add this tag. Second, all maintainers who integrated the patch add this tag.

Acked-by The tag “is often used by the maintainer of the affected code when that maintainer neither contributed to nor forwarded the patch.” [96] This is used, for example, when a patch modifies multiple subsystems[5] one maintainer acks it, but the other maintainer ought to include it in the repository, because this is the more affected subsystem.

Co-developed-by The tag “states that the patch was co-created by multiple developers.” [96] For example, if a certain bug fix is implemented in parallel by two developers. One implementation will be dropped, but the developer has the attribution due to the tag in the other one’s commit message.

Reported-by “The Reported-by tag gives credit to people who find bugs and report them, and it hopefully inspires them to help us again in the future.” [95]

Tested-by The “tag indicates that the patch has been successfully tested (in some environment) by the person named. This tag informs maintainers that some testing has been performed, provides a means to locate testers for future patches, and ensures credit for the testers.” [95]

Reviewed-by The tag “indicates that the patch has been reviewed and found acceptable according to the Reviewer’s Statement” [95]. This means the following four requirements have to be met. The patch has to be reviewed,

[5] A patch that modifies this file (`drivers/net/wireless/ath/ath9k/ath9k_pci_owl_loader.c`) affects both subsystems `QUALCOMM ATHEROS ATH9K WIRELESS DRIVER` and `ARM/ACTIONS SEMI ARCHITECTURE`.

the reviewer's concerns have been answered sufficiently, the patch is free of (known) bugs or is worth integrating it (the benefits outweigh the damage), and the reviewer believes the patch is sound.

Suggested-by The tag credits the person with the idea which is implemented in this patch.

Fixes The tag indicates the commit that introduced the fixed bug. This tag helps to fix the bug in the stable kernels.

When we change the point-of-view to an outside view (like ours as researchers doing a ex-post analysis), the process can be described like shown in figure 1.1

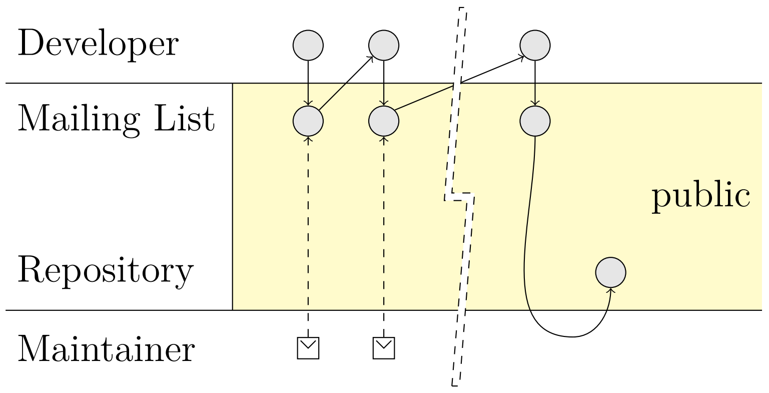

Figure 1.1: Linux Contribution Process, Outside View

The observer can only trace what happens on the mailing list and in the repository (yellow area). The work of the developer, creating and improving the patch, cannot be observed. Further, the integration of the patch into the maintainer's repository cannot be observed as there is no link between the patch-mail and the commit. However, the PaStA-tool enables us to reconstruct both links; the link between versions of patches and the link between the patch-mail and the commit.

New modules When adding a new driver to the kernel, the source code usually does not meet the quality requirements for the kernel. To enable early publication of drivers by the vendors, the so called staging tree was introduced in 2008 [54]. The requirement to get a driver in the staging tree is quite simple; it must compile properly [5].

Corbet analyzed the performance of the processes related to the staging tree. These processes (see figure 1.2) handle how drivers are:

added to staging A new driver is first added to the staging tree, if the driver is mature enough it is integrated to mainline.

merged in staging Some drivers take care of similar hardware often they are merged in the staging tree to minimize the required maintenance.

integrated into mainline When the driver's quality is high enough, the driver is merged to mainline.

removed form mainline When a driver not used anymore the driver is staged out (moved to the staging tree) and is later removed.

removed from staging When a driver is not needed anymore it is removed from the staging tree.

Some drivers failed to mature or are still in the process. The measurements can be summarized in a state diagram (see figure 1.2).

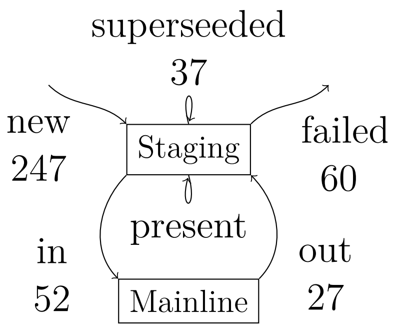

Figure 1.2: Linux Staging Process Performance

Release Process To follow Raymond's statement "release early, release often" Linus established a certain process that enables frequent releases with new features but ensures the stability of the kernel. Usually[6], Linus releases a new version of the kernel every two months.

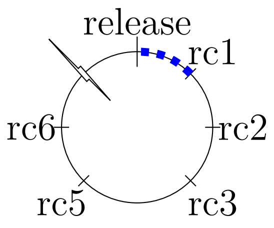

Figure 1.3: Linux Release Cycle

After the release of a new kernel, the merge window (MW) begins (MW is blue-dotted, see figure 1.3). This is the high workload phase of the maintainers. All major changes are merged by Linus into the mainline repository. Two weeks later, the merge window closes, and Linus publishes the first release candidate (`-rc`) [4].

[6]For example, the release of kernel version 3.1 took 94 days, instead of the usual 63/70 days, due to a cyber attack against `kernel.org`. The release of version 4.15 took 78 days due to the Meltdown and Spectre patches.

	accepted	not accepted
answered	accepted	rejected or in discussion
not answered	instant accepted	ignored or new

Table 1.1: Patch Matrix

> When the merge window opens, top-level maintainers will ask Linus to "pull" the patches they have selected for merging from their repositories. If Linus agrees, the stream of patches will flow up into his repository, becoming part of the mainline kernel. The amount of attention that Linus pays to specific patches received in a pull operation varies. It is clear that, sometimes, he looks quite closely. But, as a general rule, Linus trusts the subsystem maintainers to not send bad patches upstream.

Now the stabilization phase starts and Linus publishes a new release candidate every week (see figure 1.3). In this phase, the maintainers and the developers try to fix all bugs which were integrated or introduced by side-effects with other changes. Usually, the stabilization phase lasts six or seven weeks, but as mentioned earlier sometimes it takes longer.

> Over the next six to ten weeks, only patches which fix problems should be submitted to mainline. On occasion a more significant change will be allowed, but such occasions are rare; developers who try to merge new features outside of the merge window tend to get an unfriendly reception. As a general rule, if you miss the merge window for a given feature, the best thing to do is to wait for the next development cycle.

1.2.1.4 Ignored Patch Definition

As described earlier, a patch sent to the mailing list usually receives an answer by the maintainers and the reviewers. But not all patches are answered. Patches can be accepted or not and answered or not. As one can see in figure 1.1 there are six types of patches:

1. Accepted Patches - patches which are in the mainline kernel
2. Instant Accepted Patches - patches which are upstream without a discussion; this can happen when a maintainer contributes
3. Rejected Patches or In Discussion - patches that were discussed but not integrated (yet)
4. Ignored Patches or New Patches- Patches that were ignored; patches not answered yet

For the research in this thesis, we refined the simple framework above and created the following definition for ignored patches.

On average, a patch is answered within four days and five hours. 50% of the patches are answered within 12 hours. 99.5% of the patches are answered within 79 days. Thus we assume a patch that is not answered within 6 months will not receive an answer anymore [7].

A patch is ignored if …

1. … the patch has no answer from persons other than the author for over 6 months
2. … the patch is not accepted upstream
3. … all related patches meet requirements 1. and 2.

It is trivial that an upstream patch is not ignored. Some maintainer (and Linus) worked on that patch.

Additionally, it is trivial, that a patch must not be answered to be ignored. There are several cases where the author of the patch answers on the patch. This can either be a comment or a repost of the patch; nevertheless, the patch is ignored if the author is the only person in the thread.

Related patches are reposts of patches and patch series. Patch series usually receive one answer to all patches, which is sent to the first mail of the series. As a result, the whole patch series is not ignored if there is an answer for one patch of the series. If a repost receives an answer, it is obvious that the first patch will not get an answer anymore. Thus, a patch is only ignored if all posts of the patch are ignored.

1.2.2 Glossary

Code Change A code change is the combination of an operator and an abstract syntax tree (AST) [112]. The operators of the code changes sent to the Linux kernel development are `'+'` adding the AST and `'-'` removing the AST. In the Linux kernel development each AST is only one line of code.

Commit A commit is a patch applied to a repository. In this thesis, a commit is by definition an upstream patch, as we are analyzing only Linus' repository.

Diff A diff is a collection of code changes. The suggested changes are stated combined with the context where the change is placed [34].

get_maintainer.pl `get_maintainer.pl` is a perl-script by Joe Perches which "Print selected MAINTAINERS information for the files modified in a patch or for a file".

LKML The LKML (short for Linux kernel mailing list) is the main mailing list for the kernel development [68]. A mail sent to the LKML (`linux-kernel@vger.kernel.org`) will be forwarded to about 5000 subscribed mail accounts [106]. Besides the LKML there are several other mailing lists specific

[7]The longest time between a mail and its first response is 1836 days [139].

for one topic like network development (`netdev@vger.kernel.org`). Each patch for the Linux kernel should be sent to the specific mailing list and to the LKML for archival reasons. As usual in the kernel development, there is no consent, and every sender of a patch can decide it on his/her own.

Mailing List A mailing list is a mail distribution system. One can subscribe a mail-account to the list (if public). One can send a mail to the list's address. This mail is then distributed to all subscribers of the mailing list. Mailing lists are used by Linux [68], Wikipedia [164] and other organizations.

MAINTAINERS-file The `MAINTAINERS`-file contains a list of the subsystems. The sections of the subsystems are sorted alphabetically by name. Each section contains information about the responsibilities. A subsystem' section contains information about the responsible maintainers and reviewers, a lists of the files and folders assigned to the subsystem. Some sections contain additional details about special mailing lists, IRCs, or web services used.

PaStA Patch Stack Analysis – Tool. This tool maps patches from the mailing list to the commits in the repository.

Patch "A patch is a code contribution" [23]. The contribution can have different forms (e.g. a mail, or a Gitlab merge request). In the Linux kernel development a patch is a mail, containing a suggested change. It contains the description of the change and the diff. The change proposed in a patch is supposed to be one logical change. If the same logical change has to be applied multiple times, it has to be split into a patch series.

Patch Series A patch series is a collection of patches that are related. Either one change has to be applied multiple times [1] or multiple consecutive changes [3] are proposed.

Subsystems The source code of the Linux kernel grew so big that the code was split into parts, called subsystems. Each subsystem has a responsible maintainer. The subsystems are organized hierarchically. When Linus Torvalds collects the changes for a new release, he collects these changes from specific subsystem maintainers. These maintainers collect the changes from inferior subsystem maintainers. Corbet published an article about the patch flow[8] between the subsystems for release 4.14 [41]. Additionally, he published this analysis for release 2.6.29 [39] and 4.4 [38].

Upstream Upstream is a state of code or of a patch. Upstream means if given code or given patch can be found in the official productive repository. In the Linux project this is Linus' repository on `kernel.org`.

[8]This is a different patch flow than described in Capraro's Patch Flow method described earlier.

1.2.3 Related Work

In this thesis, we analyze the Linux kernel development process. As this research combines various areas of studies, the related work section will be split into several sections. The first section will address the analysis of open source. The second section will address the analysis of the Linux kernel. Afterwards, we will discuss process analysis literature. Further, the concluding section discusses papers that affect multiple topics (See overarching papers).

1.2.3.1 Open Source Projects

Quality OSS claims to be of higher quality compared to proprietary software. There is Linus' law: "given enough eyeballs, all bugs are shallow" [129]. Linus' law expresses that by conducting enough review, all bugs can be found and eliminated. In open source projects, there is a sophisticated review process [159]. Aberdour [7] analyzes how open source projects achieve this high quality. He extracts lessons learned from open source projects to apply them to closed source software development. One of these insights is the review process. In contrast, we assess the fact that ignoring patches is not compliant with the review process, as this might influence the quality of the software.

Community Besides the various research of open source projects, in general, several papers are analyzing the community of open source projects. This thesis addresses this topic as well as analyzes a specific type of communication within the Linux kernel development process (ignored patches).

"Every beginning is difficult" is a saying which fits perfectly to open source projects. Most contributors are lost after the first contribution. Plenty of research address this issue. Morgan and Jensen [110] analyze a university course that makes students contributors in open source projects. Furthermore, they suggest improvements to these courses. Steinmacher et al. [144] analyze the behavior of newcomers in open source projects. They found evidence that an early answer, and the tone of the response received influences the newcomers' decision (not) to abandon the project. Jensen, King, and Kuechler [80] analyze the abovementioned topic as well. They look at the first interaction of newcomers with the project's mailing list, having in mind that open source projects need to acquire more developers to grow. The first interaction is key to bind developers to the project. Their research shows that new developers are more likely to continue contributing when the response to the first interaction is timely. In our research, we take a more specific look at the case that the contribution is ignored. Furthermore, we do not limit our research to only newcomers but analyze all developers.

Besides, analyzing the newcomers, there is plenty of research of the existing community. "Communication is key" is another saying matching our situation.

Regarding this saying, Robertsa et al. [133] analyze the communication network in the Apache HTTP-server open source project. Like Robertsa, we are analyzing the communication in open source projects; however, we are only looking at the unanswered messages. Crowston and Howison [46] analyze the structure of communication in open source projects. They focus on the hierarchical structure and the centralization of 122 open source projects. In contrast to these studies, we do not conduct a network analysis. Instead, we look at one kind of communication and deeply analyze this special case in one project, the Linux kernel.

In the Linux kernel development and several other projects, the communication takes place using mailing lists. Bohn et al. [19] conduct a social network analysis on the R-mailing list and include the content of the mails sent. They were able to detect the interests of the people communicating. Guzzi et al. [72] research the mailing list of Lucene[9]. Like Bohn et al., Guzzi et al. use the content (topic) of the mails to enhance the analysis. Similar to their research, we analyze mailing lists. However, we are running our analysis on the Linux kernel data and add data from the SCM-system to our study. In our analysis, we use the content of the mails (the subject) as well to enhance our analysis.

Robles et al. [134] analyze the participation of women in free libre open source software (FLOSS) projects and compare the proposed values to measurements from the 2000s. In this thesis, we are addressing the question of the involvement of women as well. But conduct further analyses.

1.2.3.2 Linux

In this thesis, we do not analyze open source projects in general, but we focus on the Linux kernel.

The LWN publishes statistics about the kernel releases [43, 37, 42]. There is a list of articles regarding the statistics on `KernelNewbies.org`[10]. The statistics published contain information about the top contributors regarding submitted change sets, lines changed, tested change sets, and reviewed change sets. Furthermore, one can find information about the associated companies of the contributors. Besides the LWN statistics, there is the Linux Kernel Development Report [45] published by the Linux Foundation [45]. This report is not published as regularly as the LWN's. This thesis contains further statistics regarding ignored patches. However, we will not state names as we do not want to judge the behavior of the community nor score contributors, maintainers, or subsystems.

[9] `https://lucene.apache.org/`
[10] `https://kernelnewbies.org/DevelopmentStatistics`

1.2.3.3 Process Analysis

As the process of software development is often either not documented or not lived as documented, there are methods to do an ex-post analysis of the developed software. Gousios and Kalliamvakou [84, 69] publish a model that combines "traditional contribution metrics with data mined from software repositories". Their model uses data mainly from the code and documentation repository (here git repository), mailing lists, bug databases, wikis, and IRCs (internet relay chat). Our approach is mainly based on data from mailing lists and additionally uses the code and documentation repository.

Capraro [23] proposes the Patch-Flow method, which enables the management and researchers to find out who sent a patch to which project. The method was developed to analyze the movement of patches within a company (inner source). In contrast to the Patch-Flow analysis, our method's context of analysis is the project itself, not a company. As our approach does not need a strict assignment of the contributors to a company's hierarchy, our method is suited better to be applied to open source projects.

Crowston [47] analyzes the behavior of open source communities working with bulletin boards and mailing lists on how they fix bugs. He summarized the bug fixing process in six steps:

1. Submit
2. Assign
3. Analyze
4. Fix
5. Test and Post
6. Close

In Crowston's step `post patch` (step 5.2), the posted patch can be ignored, and the bug fixing process fails. We look at this specific case, but we do not further investigate the effect on the bug fixing process.

Due to the size of the Linux kernel, the kernel is split into multiple subsystems. This enables the distributed work of maintainers governing the Linux kernel. It is known that some maintainers are overloaded with work [113, 71, 141, 140]. Zhou et al. [171] analyze the scalability of the Linux kernel maintainers' work. In their analysis, they find out that the work accomplished by the maintainers scales with factor 0.5. Meaning a quadrupling of the maintainers is required to double the managable work. As part of our research, we want to find out if we can detect subsystems that are overloaded by analyzing the ignored patches. Additionally, we want to find out how subsystems scale with an increasing amount of patches received.

One of the flaws of conducting software process analysis on the Linux kernel is the missing link between the discussion on the mailing lists and the commit in the repository itself. Ramsauer et al. [125, 126] propose a tool that maps patches from mailing lists to the commits in the repository (for more information regarding this tool, see section 1.4.2). We use his work to determine upstream patches. We further extend the tool by the ability to detect ignored patches and export the collected data.

1.2.3.4 Overarching Papers

Additionally to Kuechler's research of newcomers in open source projects [90], he further investigates the impact of the gender on the joining process. He finds indicators for the discrimination of women in open source projects. Kuechler uses data from the US-Census [26] and openly available semi-scientific data (e.g. [35]) to determine the gender of developers. As we want to analyze not only some regional groups (US, Germany) we use Namsor [111]. This service provides the mapping from names to gender not only for one region but worldwide.

In 2013 and 2016, Wolfram Sang gave a talk [141, 140] about a scaling problem of the Linux kernel. He measured the time until a patch is answered and counted the unanswered patches. We will conduct deeper analyses of the unanswered patches.

In Will My Patch Make It? And How Fast?" German et al. [81] propose some measurements for patches and the sources where the data can be measured (e.g. mail, git). In the second step, they evaluate the data to extract guidelines on how to get patches accepted faster. They analyze the characteristics of patches to generate the guidelines. In this thesis, we analyze the ignoring of patches instead of the acceptance of patches. Further, we extend German's research by the author's, the subsystem's, and the mailing list's point of view.

1.3 Research Method

We want to analyze the phenomenon of ignored patches in the Linux kernel development process. In this thesis, we conduct a field study, more precisely a case study. According to Stol [145], conducting a field study enables us to achieve maximum realism of context. In computer science, the common methods for field studies are descriptive or exploratory case studies [145]. To answer our RQs, we conduct an exploratory, embedded, single-case case study with two stages.

The first step is the data collection. We extract the information from the Linux kernel mailing lists and Linus' git repository. The data collected per patch was selected based on expert knowledge and related work. Information about patches, authors of patches, mailing lists and subsystems is extracted. The process of data extraction is discussed in section 1.4.

The second step is to analyze the data extracted to answer the RQs. We use among others, methods from Yin (e.g. time series) [170] and Tukey (e.g. 5-number summary) [155] to analyze the collected data. We discussed each RQ in a seperated subsection.

1.3.1 Case Study

Due to the limited scope of the thesis, there is only one case analyzed, the Linux kernel. Nevertheless, the case study is designed in a way so that it can be applied to other projects where the necessary data is available [142].

1.3.1.1 Case Selection

The study can be applied to any project that fulfills the following requirements:

1. **Available Source Code (Management System)**: The source code has to be available to the researches. Besides the source code, the change-history is required.
2. **Available Change Request History**: To identify ignored code change requests (here patches), the change-requests have to be available.

The bigger the project, selected as case, the better. Big projects enable statistically sound insights. As the Linux kernel is the biggest open source project matching the requirements, we selected Linux as case.

1.3.1.2 Case Design

There are different kinds of case studies. Case studies can differ on the purpose of research, whether different units of analysis are investigated, and whether they study the phenomenon in different contexts or not. This section discusses that an exploratory (purpose), embedded (one unit of analysis), single-case study (one context) is performed.

Purpose of the Case Study According to Runeson [135] case studies can be used for different purposes:

- **Exploratory** – finding out what is happening, seeking new insights, and generating ideas and hypotheses for new research
- **Descriptive** – portraying the current status of a situation or phenomenon
- **Explanatory** – seeking an explanation for a situation or a problem, mostly but not necessarily, in the form of a causal relationship
- **Improving** – trying to improve a certain aspect of the studied phenomenon - this is very close to action research [170]

As this thesis seeks new insights on ignored patches and answers questions by stating hypotheses, this study is an exploratory case study.

Unit of Analysis According to Yin [170] there are two different units of analysis case studies:

- **Holistic** – studying the case as a whole
- **Embedded** – dividing the case into multiple units of analysis

In this thesis, we analyze the context of the Linux Kernel and the case of patch-based development. In this context, we analyze four different units of analysis:

1. Patches
2. Authors of Patches
3. Lists
4. Subsystems

Thus, this case study is an embedded case study.

Context(s) of study Yin [170] distinguishes between:

- **Single case studies** - analyzing one case
- **Multiple case studies** - analyzing and comparing multiple cases

The case study design can be applied to all cases fulfilling priorly defined requirements (see section 1.3.1.1). In this thesis, due to the limited scope, we are only analyzing one case, the Linux kernel. We try to keep this case study design easily adaptable for similar cases which will likely arise in the future.

1.3.1.3 Data Collection

Marshall and Rossman [108] list several possible sources of evidence. Yin [170] selects the most common of those:

- **Documentation** (Linux kernel documentation)
- **Archival records** (git repository, LKML)
- Interviews
- Direct-observations
- Participants-observation
- Physical artifacts

As proposed by Yin the combination of different sources of evidence improves the quality of the study. The marked (bold-font) sources are used in this thesis.

Lethbridge [93] groups the sources in three categories depending on their need for direct access to people of the studied situation.

First Degree Direct involvement (e.g. Interviews)
Second Degree Indirect involvement (e.g. Instrumenting Systems)
Third Degree No involvement, work with artifacts only (e.g. Analysis of Tool Logs)

The sources used in the thesis can be grouped into the third degree (documentation, archival records). This fits well with the goal of the thesis, as this leads to higher reliability of the research [93] due to unbiased sources of evidence. The research conducted is based on a mixture of several sources proposed by Lethbridge:

Analysis of Electronic Databases of Work Performed Both, the LKML and the git repository can be considered as electronic databases of work performed as both are used to manage work artifacts.
Analysis of Tool Logs Again, both LKML and the git repository can be considered as tool logs because, there is a large quantity of meta-data which can be considered as Log.
Documentation Analysis Besides the kernel documentation, Lethbridge [93] suggests "group e-mail lists" as a source for documentation.

The data collection plan requested by Lethbridge can be found in section 1.4. The common issues with the data collection methods are discussed in the subsection below.

1.3.1.4 Four Principles of Data Collection

To ensure the quality of the data used for the research, we have to follow the four principles of data collection [170] which are:

1. Use Multiple Sources of Evidence
2. Create a Case Study Database
3. Maintain a Chain of Evidence
4. Exercise Care when Using Data from Electronic Sources

Use Multiple Sources of Evidence To prevent biases, Yin suggests that the research should be based on distributed data gathering and analyzing. Yin mentions Patton's [116] four types of triangulation (Data-, Investigator-, Theory-, and Methodological-Triangulation).

Data Triangulation We use multiple sources of evidence (LKML, Mainline Repository). However, the data gathered is not redundant. As our data sources (except Namsor) are the reference data-sources, our data sources are likely to be correct.

Investigator Triangulation We do the investigation mostly computer-aided. A peer group of experienced researchers checks the implementation. One researcher (Sebastian Duda) conducts the analyses as this is required for this type of thesis. However, the analysis is additionally checked by a peer group. In summary, we took care of the investigator triangulation, but we are limited by the thesis requirements.

Theory Triangulation As this is an exploratory study, there is no underlying theory. The theoretical framework defined in section 1.2.1 is based on multiple perspectives. In the paragraph, the differences and similarities are discussed.

Methodological Triangulation We follow well-established methodologies. The methodologies used and how they interact were discussed earlier in the beginning of section 1.3.

Create a Case Study Database Researchers are expected to store the data used for the research in a case study database [170]. For our first step, creating the dataset, all data used is publicly available. In the section 1.4.1, the links to the sources are stated. In this step, we combine the information from these sources and create the dataset. The source code, which does the processing, is publicly available, and a reference is stated as well. Additionally, we published and referenced the dataset created. The second step, the analysis, is based on this dataset. Again, we published and referenced the code and the results of the analysis. Due to licensing issues, we are not allowed to publish the data from Namsor.

Maintain a Chain of Evidence As described in create a case study database anyone can reproduce the results of our research. All links between the raw data and the results are well documented and reproducible.

Exercise Care when Using Data from Electronic Sources Yin [170] lists common pitfalls when using electronic sources:

- The first is to be overwhelmed by the amount of data. We limited the amount of data used based on the available data sources (mailing lists) and a time frame.
- The second pitfall is not to cross-check the data. As mentioned earlier, we use data directly from the official sources.
- The third pitfall is to be careful with social media sites. We simply do not use this source of evidence.

Data Handling and Privacy As all data processed is publicly available, the data can be used for the research and can be stored without special precautions. All data from the Linux kernel development (LKML, other mailing lists, and the kernel's git repository) are processed compliantly to the DCO v1.1 [51].

1.3.2 Quality of Research Design

In case study research, there are four design-tests to judge the quality of the research design [170]:

Construct validity identifying correct operational measures for the concepts being studied

Internal validity Seeking to establish a causal relationship, whereby certain conditions are believed to lead to other conditions, as distinguished from spurious relationships (for explanatory or casual studies only and not for descriptive or exploratory studies)

External validity defining the domain to which a study's findings can be generalized

Reliability demonstrating that the operation of a study - such as the data collection procedures - can be repeated with the same results

1.3.2.1 Construct Validity

Construct validity tackles the issue that the researcher tends to collect data and design studies suiting their expectations. To prevent this, we split the research into two steps.

First, we conducted data collection. To select data to collect, we used expert knowledge and previous research. Some of the previous research was analyzing patches as well but with a RQ orthogonal to ours. As we analyze all patches sent to the mailing lists, the selection of the analyzed patches is not biased. There was no filtering of patches other than the selection of analyzed mailing lists and the time frame. The filtering of the mailing lists is due to the fact, that not all mailing lists were recorded over the complete time frame analyzed. The time frame was chosen so that the quality of the data suits our needs. The selection of the lower limit is based on the fact that we are more interested in the development in the recent past. Selecting the time after the release `v2.6.39` enabled us to simplify the computer program used for analysis. Based on the definition of ignored patches, a patch has to be unanswered for 6 months. When we started our research the potentially newest ignored patch was a patch sent at the 12th of January 2019. To avoid selecting an incomplete development cycle, we selected the release of `v4.20` as upper limit.

Second, we analyzed the data. As the RQs are well defined, we have objective measures to answer the questions.

This approach results in an increased effort for data collection, as more data is collected than needed. However, the data collected will be published for other researchers, thus the additional effort has merit. Furthermore, this approach minimized possible data collection biases.

1.3.2.2 Internal Validity

The threat to internal validity is that the research design is based on invalid assumptions. As we conduct exploratory research, we have no underlying assumptions and can omit this test with the approval of Yin [170].

1.3.2.3 External Validity

External validity addresses the generalizability of the research. To archive external validity, the research has to be analytical and statistical generalizable.

Analytic Generalization Analytic generalization means that the research can be applied to other settings based on similar theoretical concepts. Based on the theoretical concepts we used for the research (see section 1.2), the RQs and the methodology can be applied to almost all open source projects. However, the results might differ. This is due to differences in the development process, tooling, or even a stricter or weaker code of conduct (CoC)[11] [154]. Compliant to Yin, the concepts and boundary conditions diverge in those cases.

Statistical Generalization Statistical generalization is the fact that any of the results are based on enough data to be statistically well sustained. Stol et al. [145] stated that the results of field studies are not statistically generalizable. As we applied our study design to only one case, we agree with Stol's statement. Further research can help to make the insights more generalizable.

1.3.2.4 Reliability

To ensure reliability Yin suggests creating a document describing the process of the case study. Furthermore, all data collected are openly available (see section 1.4). The software used to analyze the data is open source[12]. The analysis is described in detail in the respective parts of the thesis. This enables reviewers to track the development of the case study and reproduce the study results.

[11]"A code of conduct is a set of rules outlining the norms, rules, and responsibilities of, and or proper practices for, an individual." [33]

[12]`https://github.com/lfd/pasta`

1.3.3 Selection of Measurements

In the dataset, patches are listed with measurements. The selection of the measurements is based on previous research by German et al. [81] and expert knowledge. In this subsection, we will describe the process of how we selected the measurements.

The measurements were collected in an iterative two-step process. The first step, was the brainstorming of new measurements. The second step, was the discussion of the ideas in a peer group (Duda, Bulwahn).

In the discussion, the measurements were prioritized, overruled, and it was considered how to measure the patches' properties. Such as, the readout of the `from`-header of the patch's mail was higher prioritized than the number of signed-offs. An example for an overruled measurement is, the size of a patch series as there is no reliable way to detect the size. A discussion on, for example, how to determine the kernel value was required since there was disagreement about the assignment of patches that were sent in the merge window to the kernel versions.

1.3.4 Exploratory Data Analysis

Tukey [155] does not provide a guide to follow. He provides assistance of how to analyze data. As there is no overarching method we can follow, we will describe the methods used at the respective parts of the text. We used for example Tukey's `5-number-summary` and the suggested `sqrt-scale` in some plots.

1.4 Data Collection

To create the dataset (see `https://duda.pub/pub/linux_characteristics_ma.csv`), we combined information from three sources: the Linux kernel mailing lists [68], the Linux git repository [150], and Namsor [111]. The mailing lists contain most of the information, the patches itself, and meta-data like who is the author. The git repository is required to check if a patch can be found upstream. Namsor can detect the country of origin and gender based on the name. To link the various data sources, we used a tool named PaStA [122].

In the following subsections, we will discuss the data sources and elaborate on each measurement. Furthermore, we will assess the quality of the data collected.

1.4.1 Data Sources

1.4.1.1 Mailing Lists

Mailing lists can be stored, for example in `mbox` or `pubin` format. The two formats are the formats that are supported by PaStA. The `mbox` format stores all mails of the mailing-list in one file as a sequence [13]. The `pubin` format stores all mails in a git repository.

To access mails sent, there are several archives for the Linux kernel mailing-lists:

- `https://git.kernel.org/pub/scm/public-inbox/`
- `https://www.mail-archive.com/linux-kernel@vger.kernel.org/`
- `https://marc.info/`
- `https://lkml.org/`
- `https://lore.kernel.org/lkml/`

PaStA uses the mailing-list archives from `git.kernel.org`[14], which are in the `pubin` format. Because, the repositories are the archive maintained by `kernel.org` itself, we assume them to be correct (issues are discussed below) and complete.

Limitations of the Data Source We analyze the mails sent in the following time frame:

- Start: release of kernel version `2.6.39` (19.5.2011)
- End: release of kernel version `4.20` (23.11.2018)

This time frame was chosen due to the following reasons. We chose...

- an upper limit before January 2019, because it was implied by the definition of an ignored patch.
- the release of `v4.20` as upper limit, because we tried to avoid to have data of an incomplete development cycle in the dataset.
- a lower limit, because the quality of the old data decreases.
- the release of `v2.6.39` as lower limit, because this limits the amount of data to analyze to a reasonable amount.

In total, we know of 207 mailing lists that are currently in use. We only analyze the mailing-lists, which are completely archived. Below there is the list of all analyzed mailing lists. We exclude incomplete archived mailing lists as the thread detection cannot cope with missing mails. As the thread detection is key for the research we exclude these mailing lists.

[13] `https://en.wikipedia.org/wiki/Mbox`
[14] `https://git.kernel.org/pub/scm/public-inbox/`

- `linux-amlogic@lists.infradead.org`
- `linux-arm-kernel@lists.infradead.org`
- `linux-i3c@lists.infradead.org`
- `linux-mtd@lists.infradead.org`
- `linux-riscv@lists.infradead.org`
- `linuxppc-dev@lists.ozlabs.org`
- `cocci@systeme.lip6.fr`
- `linux-block@vger.kernel.org`
- `linux-bluetooth@vger.kernel.org`
- `linux-btrfs@vger.kernel.org`
- `linux-cifs@vger.kernel.org`
- `linux-clk@vger.kernel.org`
- `linux-crypto@vger.kernel.org`
- `linux-ext4@vger.kernel.org`
- `linux-fsdevel@vger.kernel.org`
- `linux-hwmon@vger.kernel.org`
- `linux-iio@vger.kernel.org`
- `linux-integrity@vger.kernel.org`
- `linux-kernel@vger.kernel.org`
- `linux-media@vger.kernel.org`
- `linux-mips@vger.kernel.org`
- `linux-modules@vger.kernel.org`
- `linux-next@vger.kernel.org`
- `netdev@vger.kernel.org`
- `linux-nfs@vger.kernel.org`
- `linux-parisc@vger.kernel.org`
- `linux-pci@vger.kernel.org`
- `linux-renesas-soc@vger.kernel.org`
- `linux-rtc@vger.kernel.org`
- `linux-security-module@vger.kernel.org`
- `linux-sgx@vger.kernel.org`
- `linux-trace-devel@vger.kernel.org`
- `linux-watchdog@vger.kernel.org`
- `linux-wireless@vger.kernel.org`

During the research, we noticed that some mailing list archives' data contains errors. We analyzed patches in the LKML and compared them with patches in the other mailing lists' archives. We noticed that 14.1% of the mails have different headers when loading them from different mailing lists [120]. When analyzing this issue, we noticed that the archive of the LKML contains all headers found in other mailing lists' mails, but in other mailing lists, some headers are missing. We further noticed that only historical data is affected by this issue.

Nobody intentionally archived mailing lists from the beginning. The archives are based on inboxes from developers or `gmane.org`. Some mailing list archives are based on the records from `gmane.org` which is known for changing the mails before persisting them. We believe that broken inboxes from the developers and invalid archival records from Gmane are responsible for this issue.

Extraction of Patches In the selected time frame, about 3 million mails were sent to the analyzed mailing lists. Figure 1.4 shows the extraction of the about $800k$ relevant patches.

1. Of the $3M$ sent only $1.15M$ are patches, according to PaStA's patch detection.
2. Of the remaining $1.15M$ patches, we excluded all patches which were not patching the Linux kernel but another project (e.g. EXT4 userland tools).
3. Of the remaining $1.11M$ patches, we excluded all patches sent by bots, all patches which were process-related (e.g. linux-next, git-pull), and patches for the stable versions (stable-review).

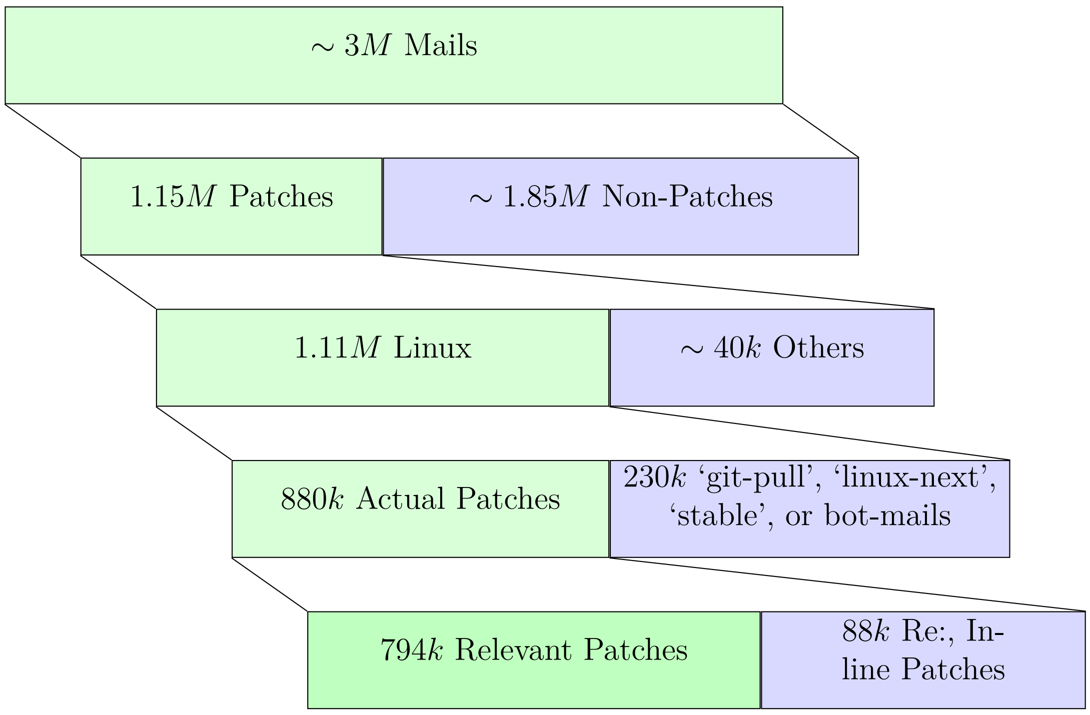

Figure 1.4: Extraction of Patches

4. Of the remaining $880k$ patches, we excluded the patches which were sent in a reply, because there is another mail in the thread which likely prevents our requirements to be met. Most of the patches in replies are commentatory patches or code snippets.

1.4.1.2 Git Repository

Ramsauer uses Github to host PaStA. To include the Linux repository conveniently in PaStA [124], we use Linus' git repository from GitHub (`https://github.com/torvalds/linux`). The repository on GitHub is identical to the one on `kernel.org`.

This git repository contains data since 2005. As the repository at `kernel.org` is the official repository, we assume the data to be complete and correct.

1.4.1.3 Namsor

Namsor (`https://namsor.com`) is an online service which determines the `gender`, and `country of origin` based on the name of a person. Namsor is a big data company in the domain of onomastics. The service, provided by Namsor, was used in a paper by Science-Metrix [78]. Information about the quality of the results can be found in a document [25] provided by Namsor.

1.4.2 PaStA

PaStA [122] is a backronym for `Patch Stack Analysis`. It is an open source tool for analyzing patches. The tool is written in `Python3` and published under `GPL-2.0`. PaStA can compare patches and link similar patches from different sources. It is developed by Ralf Ramsauer (Ostbayerische Technische Hochschule Regensburg)[15].

The tool was initially developed to map the out-of-tree development, such as PreemptRT on the upstream kernel commits [125]. Since mailing lists are semantically similar to out-of-tree patch stacks, this feature was added. Therefore patches from mailing lists can now be mapped to the upstream kernel.

The tool was enhanced in the course of this thesis. We added a feature to detect ignored patches. Furthermore, the feature to export the dataset based on the analysis of PaStA was developed.

1.4.3 Measurements of Patches

In this section, we describe all data of the dataset. We extracted the data from the mailing lists (see section 1.4.1.1) and the git repository (see section 1.4.1.2). We selected the measurements based on expert knowledge, and the paper published by German et al. [81].

The dataset is a `csv`-file that contains the columns named below. Each patch is a line identifiable by `patch-id`. For creating the author/subsystem/mailing list dataset, the following data structure contains the patch data. There is a list of patches. For each patch there is a dictionary where the column names are the keys with corresponding values.

1.4.3.1 Patch-ID

This measurement was elected based on expert knowledge.

Description The `patch-id` is the patch's mail `message-id`. The ID is structured like this: `< <local-id> @ <domain-part> >` [130]. The `domain`-part is the global identifier of the sending mail server. The `local-id`-part is the ID of the mail on the sending mail server. Both parts combined are globally unique.

The client itself can determine the `message-id`, thus one can find several examples where the requirements are not met. For example:

[15]`https://github.com/rralf`

- Duplicate `message-ids`
- Incorrect `domain`-part
- No `message-id`

We handle these exceptions the following way:

Duplicate `message-ids` Only one mail is analyzed, we ignore the other duplicated mails.
Incorrect `domain-part` We ignore this case, as this does not affect the analysis.
No `message-id` We ignore this mail, as it cannot be identified.

Motivation To work with the patches' data, an identifier is required. The `message-id` is a unique identifier for any mail [131]. As there can only be one patch per mail, the mail's ID can be used for the patch as well.

How to measure By reading the `message-id` header, we determine the mail's ID. As discussed above, the mail's id of a patch's mail can be used as `patch-id`.

How to validate In this thesis, we assume the readout of the ID has no issues, except the said ones. The error handling is executed, as described above. No further validation is conducted.

1.4.3.2 Name and Mail of Contributor

This measurement was elected based on German's paper [81] and expert knowledge.

Description The name of the author of the patch combined with the author's mail address used to send the patch. If the `from`-header only reveals the mail-address, the name is empty.

Motivation As we intend to reveal discrimination, we need information about the author of a patch. To have an identifier for the authors, we use the mail's `from` header.

How to measure By reading the `from`-header of the mail, we can determine the name and the mail-address of the author. If the header contains both name and mail-address, we use the convention to distinguish both parts. The name is the first part, and the mail-address is between the `less-than sign` (`<`) and the `greater-than sign` (`>`) after the name. If there is only a mail-address, the name is set an empty string.

How to validate In this thesis, we assume the readout of the `from`-header has no issues. We are aware that malicious people can fake the header. But we do not know any case where that happened before. We are also aware that it is not mandatory to publish one's name on LKML. The name can be blank, or the name can be faked. In this thesis, we assume that a name stated is the correct name.

1.4.3.3 Subject

This measurement was elected based on expert knowledge.

Description The subject of the patch's mail.

Motivation To simplify the handling of the patches, the dataset contains the subject of each patch. The subject can be used as pseudo identifier. There is no need that the subject is unique, but it helps when talking about patches, because it is more descriptive and recognizable than the mail's id. Furthermore, the subject enables researchers to apply NLP[16]-techniques on the data.

How to measure The subject of the patch is the subject of the patch's mail. Thus, we trivially read the `subject` header of the mail.

How to validate In this thesis, we assume the readout of the header is always correct.

1.4.3.4 Time

This measurement was elected based on German's paper [81] and expert knowledge.

Description A `date-time`-string in the `RFC5322` [130] format with offset (`2019-09-11 20:22:52+02:00`).

Motivation To conduct analysis based on the time-dimension, we added the time when the patch was submitted to the mailing list to the data.

How to measure Mail have a `date`-header containing the time stamp when the mail was sent. PaStA parses the header, and takes care of the different formats.

[16]natural language processing

How to validate In this thesis, we assume the readout of the `date`-header has no issues. We are also aware that there are uncountable different types of date-time-strings in the header. PaStA makes a best effort and parses most of the date-time-strings. The timestamp `0` is used if the date-time-string cannot be parsed.

1.4.3.5 Kernel Version and Release Candidate

This measurement was elected based on German's paper [81] and expert knowledge.

Description The latest kernel version and release candidate when the patch was submitted.

Motivation To simplify analyses of the Linux kernel development cycle, we add corresponding information; the kernel version and the release candidate the patch was sent to.

How to measure Based on the `time`-data, we determined the latest release(-candidate)'s tag.

The Linux kernel development cycle consists out of the merge window (two weeks after any release) and the stabilization phase. In the stabilization phase, there are usually seven unstable releases, so called release candidates. See section 1.2.1.3.

Release candidates have the format:

`v<kernel-version>-rc<release-candidate number>` (v5.2-rc1)

And stable releases have this format

`v<kernel-version>` (v5.1)

If the patch was submitted in the merge window, the value of `rc` is 0; if it is released in the stabilization phase, this value is the release candidate's number.

If a patch is sent during the merge window, the current tag is a major release (e.g. `v5.1`). However, the patch is sent during the development of the succeeding kernel version (e.g. `5.2`). Thus, the `kernel version` of the patch stated is `5.2`, and the `release candidate` of the patch stated is `0`.

In contrast, if a patch is sent during the stabilization phase of the kernel development, the current tag is a release candidate (e.g. `v5.2-rc1`). Since the patch is sent during the development of the stated kernel version (e.g. `v5.2`), the kernel version can trivially be read out. Thus, the `kernel version` of the patch stated is `5.2`, and the `release candidate` of the patch stated is `1`.

How to validate As the measurement of the kernel version and the release-candidate is based on previously validated measurements and the processing is trivial, spot test 1 was conducted. The evaluation of the test is shown in table 1.2.

	kernel version	elease candidate
correct	25	25
incorrect	0	0

Table 1.2: Kernel Version and Release Candidate Measurement - Results Spot Test 1

1.4.3.6 Recipients

This measurement was elected based on and expert knowledge.

Description The value `recipients` is a list of the recipients of the patch. There are some occurrences that the mail in the repository has no `to`-header [52]. In there cases the value `recipients` is empty. The value `#recipients` states how many recipients the patch has. Both persons and lists are counted.

Motivation We want to enable researchers to analyze the effect of the (number of) recipients.

How to measure PaStA extracts the recipients from the mail headers (`to` and `cc`). Mails are separated by `','` [130].

How to validate As the measurement is read out from PaStA, spot test 1 is conducted. The evaluation of the test is shown in table 1.3.

	recipients
correct	25
incorrect	0

Table 1.3: Recipients Measurement - Results Spot Test 1

1.4.3.7 List and Non-List Recipients

This measurement was elected based on expert knowledge.

Description The list and non-list recipients with respective numbers. We only extracted mailing lists associated with the kernel.

Motivation To improve the analysis based on the recipients, we distinguish between non-list recipients (e.g. maintainers) and lists (e.g. LKML).

How to measure We use the identified recipients and check each recipient if it is a known list. If so, we add the recipient to the list recipients, otherwise as non-list recipients. If the mail has no list recipient the LKML is added as list recipient.

How to validate As this is trivial, spot test 1 is conducted. The evaluation of the test is shown in table 1.4.

	list recipients	non-list recipients
correct	25	25
incorrect	0	0

Table 1.4: Lists and Non-Lists Recipients Measurement - Results Spot Test 1

The spot test complies with our expectations. We noticed some mailing lists in the non-lists recipients, however these lists are not related to the kernel development.

1.4.3.8 ID of first Mail in Thread

This measurement was elected based on expert knowledge.

Description The ID of the first mail in a thread.

Motivation As there can be multiple patches per thread, the `id of first mail in thread` helps to cluster these threads.

How to measure We use PaStA's thread feature to recognize the thread and identify the first message.

How to validate As the measurement is based on PaStA features, spot test 1 is conducted. The evaluation of the test is shown in table 1.5.

	id of first mail in thread
correct	24
incorrect	1

Table 1.5: ID of First Mail in Thread Measurement - Results Spot Test 1

The spot test complies with our expectations. It is not unusual that the threads are broken.

1.4.3.9 Is first Mail in Thread

This measurement was elected based on German's paper [81].

Description This measurement was suggested by German.

Motivation A preceding mail might raise awareness for the patch and cause a different processing of the patch.

How to measure The patch's ID is compared to the `id of first mail in thread`.

How to validate As this feature is only based on previously validated measurements, no further validation is required. The evaluation of the test is shown in table 1.6.

	is first mail in threat
correct	24
incorrect	1

Table 1.6: Is First Mail in Thread Measurement - Results Spot Test 1

The spot test complies with our expectations. If the tread-detection worked this measurement was correct as well.

1.4.3.10 Length of Thread

This measurement was elected based on German's paper [81].

Description The number of mails sent in the thread.

Motivation German proposes to count the mail in a thread before the patch itself was sent. As we are analyzing ignored patches and assume the patch not to be followed by any mail, we count all mails in the thread.

How to measure We use PaStA's thread feature to collect the thread of the patch. And count all nodes of the returned thread.

How to validate As we only use PaStA-features and trivial processing, we conducted spot test 1. The evaluation of the test is shown in table 1.7.

The spot test complies with our expectations.

	length of thread
correct	23
incorrect	2

Table 1.7: Lenght of Thread Measurement - Results Spot Test 1

1.4.3.11 Upstream

This measurement was elected based on German's paper [81] and expert knowledge.

Description If the patch is upstream, `1` is stated otherwise `0`.

Motivation We need the information if a patch is upstream for the ignored patch analysis.

How to measure PaStA compares the patches submitted to the mailing lists with commits in the Linux git repository. If they are similar [125], they are marked as upstream. This marks not only the accepted patches as upstream but similar patches as well (e.g. previous versions of the patch).

How to validate As this information is determined by PaStA, spot test 1 is conducted. The evaluation of the test is shown in table 1.8.

	upstream
correct	24
incorrect	1

Table 1.8: Upstream Measurement - Results Spot Test 1

The spot test complies with our expectations as the heuristic of PaStA is good but not perfect.

1.4.3.12 Ignored

This measurement was elected based on expert knowledge.

Description If the patch is ignored, `1` is stated otherwise `0`.

Motivation Since this thesis analyses ignored patches, this information is key for further analyses.

How to measure We check if the patch meets the requirements (see section 1.2.1.4). To check if a patch is upstream, we use the information from `upstream`. To check if a patch has foreign responses, we use PaStA's thread feature. We iterate over all nodes in the thread, if there is another author besides the patch's author, the patch is not ignored.

To check the related patches, we need to analyze two groups, other versions of the patch and other patches of the patch series:

To collect all patch versions, we use PaStA's similar patches feature. We assume that different versions of the patch are in one cluster. We further assume that only the versions of a patch are in on such a cluster.

To collect all patches of the patch series, we first check if a patch is part of a patch series. To do so, we apply a regular expression (`'[0-9]+/[0-9]+\]'`) on the subject of a mail. If it matches, the patch is probably part of a patch series and we iterate the thread to collect the other parts of the patch series. The subject of all patches in a patch series have to contain information about the patch series. The subject starts with `[PATCH <number of patch>/<number of patches in the series>]`.

If there is one patch in the collection which is not ignored, the whole collection is not ignored.

How to validate This measurement was validated in spot test 2 as this is a new feature. The evaluation of the test is shown in table 1.9.

	ignored	not-ignored
correct	11	486
incorrect	3	0

Table 1.9: Ignored Measurement - Results Spot Test 2

The spot test complies with our expectations. The implementation is robust as it does not identify patches as ignored by mistake. However, the implementation misses patch series and upstream patches.

1.4.3.13 Size of Patch

This measurement was elected based on German's paper [81] and expert knowledge.

Description The number of lines of the patch.

Motivation To enable assessing the patches, we added the number of lines changed to the dataset.

How to measure PaStA identifies the diffs in the patches by applying regular expressions. Then the diff is analyzed and the lines changed are counted [123].

How to validate As the measurement is read out from PaStA, spot test 1 is conducted. The evaluation of the test is shown in table 1.10.

	size
correct	25
incorrect	0

Table 1.10: Size of Patch Measurement - Results Spot Test 1

1.4.3.14 Number of Lines Added

This measurement was elected based on German's paper [81] and expert knowledge.

Description The number of lines of code added by the patch.

Motivation To improve the assessment of the patch, we added the number of lines added to the dataset.

How to measure To count the lines added in a patch, we use PaStA's `diffstat` feature, which counts the line in a patch starting with a `+`.

How to validate As retrieving the information is trivial, spot test 1 is conducted. The evaluation of the test is shown in table 1.11.

	lines added
correct	7
incorrect	18

Table 1.11: Lines Added Measurement - Results Spot Test 1

The spot test does not comply with our expectations. However, the results of this metric were 5 to 10% below correct value. Thus, the analyses are still valid as they only state a trend.

1.4.3.15 Number of Lines Removed

This measurement was elected based on German's paper [81] and expert knowledge.

Description The number of lines of code removed by the patch.

Motivation To improve the assessment of the patch, we add the number of lines removed to the dataset.

How to measure To count the lines added in a patch, we use PaStA's `diffstat` feature, which counts the line in a patch starting with a `-`.

How to validate As retrieving the information is trivial, spot test 1 is conducted. The evaluation of the test is shown in table 1.12.

	lines removed
correct	18
incorrect	7

Table 1.12: Lines Removed Measurement - Results Spot Test 1

1.4.3.16 Number of Files Touched

This measurement was elected based on German's paper [81] and expert knowledge.

Description The number of files affected by the patch.

Motivation To improve the assessment of the patch, we added the number of affected files to the dataset.

How to measure PaStA extracts the number of files changed by analyzing the diff shipped with the Patch.

How to validate As the measurement is read out from PaStA, spot test 1 is conducted. The evaluation of the test is shown in table 1.13.

	files touched
correct	25
incorrect	0

Table 1.13: Files Touched Measurement - Results Spot Test 1

1.4.3.17 Subsystems

This measurement was elected based on expert knowledge.

Description List of subsystems affected by the analyzed patch. The subsystem 'THE REST' was removed as any patch affects this subsystem.

Motivation Dan Williams [165, 44] proposed to introduce subsystem profiles, as the subsystems are very diverse. To enable research tackling the diversity of subsystems, the information is added.

How to measure PaStA has a feature which parses the `MAINTAINERS`-file. The `MAINTAINERS`-file in the version related to the patch is parsed, and the subsystems affected are extracted.

How to validate The list is extracted by PaStA spot test 1 is executed. The evaluation of the test is shown in table 1.14.

	subsystems
correct	21
incorrect	4

Table 1.14: Subsystems Measurement - Results Spot Test 1

PaStA's parse algorithm sometimes missed a subsystem. The missed subsystems were subsystems hierarchically between `THE REST` and the most specific subsystem.

1.4.3.18 Mailing Lists

This measurement was elected based on expert knowledge.

Description A list of the mailing lists the suggested by the `MAINTAINERS`-file.

Motivation The motivation why to extract the mailing lists is comparable to the motivation to extract the subsystem (Section 1.4.6.1)

How to measure PaStA has a feature which parses the `MAINTAINERS`-file and extracts the affected subsystems.

How to validate As the list is extracted by PaStA spot test 1 is executed. The evaluation of the test is shown in table 1.15.

1.4.3.19 Maintainers

This measurement was elected based on expert knowledge.

	mailing lists
correct	7
incorrect	18

Table 1.15: Mailing Lists Measurement - Results Spot Test 1

Description List of maintainers responsible for the patch according to the `get_maintainers.pl`-script.

Motivation Linux grew so big that it is not possible for one person to maintain the whole system alone. As a result, there are several persons involved. They are called maintainers. These persons are responsible for one or more subsystems.

How to measure In the first approach, PaStA executed the script to get the related maintainers. As often calling a script is time-intensive, we decided to make PaStA parse the `MAINTAINERS`-file on its own. In the current implementation, the subsystem and the maintainers are determined by checking the files modified. For each file, the subsystems, which matches the file-descriptor (`F:`), are extracted from the Maintainers file [17].

How to validate As this is a PaStA-feature, spot-test 1 was conducted. The results of the spot test were the same as the results of the subsystems' spot test.

1.4.3.20 Correct Maintainers

This measurement was elected based on expert knowledge.

Description The recipients of the mail is compared to the MAINTAINERS' output. The value is true if there is one correct list or one correct maintainer, otherwise false.

Motivation The `get_maintainers.pl`-script suggests recipients for any patch. A patch is supposed to be sent to all maintainers of the most specific subsystems, according to the process. As we can not determine which subsystem is the most specific, we chose the second metric.

How to measure We compare the maintainers from `maintainers` with the recipients from the mail header.

[17]There are "Keywords" (`K`) and "Files and directories with regex patterns" (`N`) to identify subsystems as well, but this is not implemented.

How to validate As this feature is only based on previously validated measures, it is not further validated. The evaluation of the test is shown in table 1.16.

	correct maintainers
correct	23
incorrect	2

Table 1.16: Correct Maintainer Measurement - Results Spot Test 1

These results correlate with results of the subsystem measurement. However, the correct maintainers measurement is more often correct as the constraints are weaker and some subsystems share maintainers.

1.4.3.21 Signed-off-by, Acked-by, and Co-developed-by

This measurement was elected based on expert knowledge.

Description `Signed-off-by` is a list of all `signed-off-by`-tags. `Acked-by` and `Co-developed-by` are respective lists. `#Signed-of-by`, `#Acked-by`, and `#Co-developed-by` counts the elements in the lists.

Motivation The `signed-off`-tag states "that he or she [the developer] has the right to submit the patch for inclusion into the kernel." [119] Thus no `signed-off`-tag should indicate that the patch is not upstream. As a result, it might be ignored right away. If there is a larger amount of tags, this means multiple developers were involved in the development of the patch. The analyses below will show if this affects the likeliness to be ignored.

The `Co-developed-by`-tag "states that the patch was co-created by several developers; it is a used to give attribution to co-authors" [119].

The `Acked-by`-tag "indicates an agreement by another developer (often a maintainer of the relevant code) that the patch is appropriate for inclusion into the kernel." [119]

How to measure We use PaStA's patch-object, which contains the patch-message. In the patch-message, the `signed-off`-tags can be found. To extract the tags, we iterate over the lines of the message and extract all lines containing `signed-off-by` [119].

How to validate As we only use PaStA-features and trivial processing, we conducted spot test 1. The evaluation of the test is shown in table 1.17.

	Signed-off-by	Acked-by	Co-developed-by
correct	25	25	25
incorrect	25	25	25

Table 1.17: Tags Measurements - Results Spot Test 1

1.4.4 Measurements of Authors

To create the dataset for analyzing the authors, one needs the patch-data as described in section1.4.3. We describe the source code extensively. As a result we can avoid an additional spot test. Nevertheless, we conduct a plausibility check. For simplicity, we group the patches by the author (The definition of `add_or_create` can be found in the appendix A.1):

```
def prepare_authors(patch_data):
  authors = dict()
  for patch in patch_data:
    add_or_create(authors, patch['from'], [patch])
  return authors
```

Afterwards, one iterates over the returned dictionary and calls the `get_author_information` method:

```
  prepared_author_data = prepare_authors(patch_data)
  author_data = list()
  for author, patches in prepared_author_data.items():
    list.append(get_author_information(author, patches))
```

In the function `get_author_information`, a dictionary is created with information about the author.

```
def get_author_information(author, patches):
  data = dict()
  …
```

1.4.4.1 Name and Mail

This measurement was elected based on expert knowledge.

Description `name` is the name of the author. `mail` is the author's mail. `author` is the combination of `name` and `mail`, which can be used as an identifier for an author.

Motivation This measurement is required to identify the author. The `name` of the author is required for further analyses (see [`sex` and `ethnics`]). The `mail` of the author is required for further analyses (see section 1.4.4.4).

How to measure The name of the author can be determined by reading the `from` header of the mail. If the header contains both name and mail-address, we use the convention to distinguish both parts. The name is the first part, and the mail-address is between the `less-than sign` (<) and the `greater-than sign` (>) after the name. If there is only the mail-address, the name is set an empty string. This is done by PaStA when creating the patch dataset. As a result, we can simply do:

```
data['author'] = author

# we select the first patch and
data['name'] = patches[0]['from_name']
data['mail'] = patches[0]['from_mail']
```

How to validate We are aware that some developers use multiple mail addresses for kernel development. This is, for example, done to distinguish private and company work or when a developer changes employer. In this thesis, we ignore these cases as the handling of this issue requires the knowledge of all developers using multiple addresses. We decided this is too much effort.

As this data is only based on previously validated data we will not conduct further verification.

1.4.4.2 Experience of the Author

This measurement was elected based on German's paper [81] and based on expert knowledge.

Description `patch experience` is the number of patches sent to the analyzed mailing lists from the author. `ignored patch experience` is the number of patches sent to the analyzed mailing lists from the author, which were ignored.

Motivation The `ignored patch experience` measurement is key for this thesis. The `patch experience` measurements is required for comparing and normalizing the `ignored patch experience`.

How to measure `patch experience` is the number of patches sent by the author. `ignored patch experience` is the number of patches sent by the author which were ignored.

```
for patch in patches:
  add_or_create(data, 'patch experience')
  if patch['ignored']:
    add_or_create(data, 'ignored patch experience')
```

1.4.4.3 Number of Mails Sent

This measurement was elected based on German's paper [81].

Description `#mails sent` is the number of mails, not only patches, sent from one author.

Motivation The number of mails sent to the mailing lists might assume the awareness and the involvement of the author in the community.

How to measure Count all mails from the author.
The `repo.mbox.message_ids` and `repo.mbox.get_messages` methods are implemented in PaStA.

```
all_mails = repo.mbox.message_ids(allow_invalid=True)
  for mail_id in all_mails:
    mails = repo.mbox.get_messages(mail_id)
    if len(mails) is 0:
      continue
    if author == email_get_from(mails[0])
      add_or_create(data, '#mails sent')
```

One can execute this before iterating over the author to improve the performance.

1.4.4.4 Organization and TLD

This measurement was elected based on expert knowledge and is inspired by the "2017 State of Linux Kernel Development" report from the Linux foundation [45]

Description `company` is the `host`-part of the `domain-part` of the mail address of the author. `tld` is the `tld`-part of the `domain-part` of the mail address of the author.

`TLD` stands for `Top Level Domain`. The TLD is the most generic part of an URL, for example `.com`, `.org`, or `.de`.

Motivation Most of the heavily involved contributors are working for companies. As these developers often use their company mail-address. The mail-addresses can give a hint on the involvement of companies. The URL-part, if not inexpressive (e.g. `.com`, `.net`, `.org`), can indicate the homeland of the company. The Linux Foundation [45] published several statistics based on the part of the mail address after the `@`, the so called `domain-part`.

How to measure The mail address extracted from the `from`-header of the mail is split by `@`. The second part is used for this analysis. we are aware of TLDs like `.co.uk`. With our algorithm, we are able to get the `.uk`-part for the country analysis, and the `.co`-part is added to the company name. This does not affect further analyses, as all mails from the company are combined with the `.co`-part.

```
# Extract '@company.com'
glob = re.findall('@.+', patches[0]['from_mail'])
if glob:
  splits = glob[0].split('.')
  # All segments but the last are the company part
  data['company'] = '.'.join(splits[:-1])[1:]
  # The last segment is the tld
  data['tld'] = splits[-1]
```

1.4.4.5 Sex and Country of Origin

This measurement was elected based on expert knowledge.

Description

`sex` we are aware of the broad spectrum of genders, however in this thesis, we will use a discretized, binary spectrum. This means, we distinguish between male, female, and other. The Namsor Api returns the gender identified combined with the probability ($0.5 to 1$). We select a threshold of 0.67. Any developer with a probability above 0.67 is added to the respective gender group (male or female) the developers below the threshold are added to the third group (other) as we cannot safely determine the gender.

`country` In addition to the gender Namsor can indicate what country the developer is from.

Motivation To detect discrimination in the Linux kernel development, we analyze the socio-ethnic characteristics of the authors.

How to measure We determine the characteristics using the Namsor-API. For more information see section 1.4.1.3. Due to licensing issues, we cannot publish the results.

How to validate See Namsor section.

1.4.5 Measurements of Lists

To create the dataset for analyzing the lists, one needs the patch data as written in section 1.4.3. As we depict the source code detailedly, we save the spot test, but conduct a plausibility check. For simplicity, we group the patches by the lists (The definition of `add_or_create` can be found in the appendix A.1). Patches are only associated with lists if the patch is sent to the list and the patch is related to the list according to the `MAINTAINERS`-file.

```
def prepare_lists(patch_data):
  lists = dict()
  for patch in patch_data:
    for list in patch['lists']:
      if list in patch['recipients']:
        add_or_create(lists, list, [patch])
  return lists
```

Afterward, one iterates over the returned dictionary and calls the `get_list_information` method:

```
  prepared_list_data = prepare_lists(patch_data)
  list_data = list()
  for list, patches in prepared_list_data.items():
    list.append(get_list_information(list, patches))
```

In the function `get_list_information`, a dictionary is created with information about the list.

```
def get_list_information(list, patches):
  data = dict()
  …
```

1.4.5.1 Name of List

This measurement was elected based on expert knowledge.

Description This measurement is the name of the list. It can be used for identifying the list.

Motivation To work with the lists' data, an identifier is required.

How to measure The `MAINTAINERS`-file states a name per list. This name is unique and can be used as identifier.

```
  data['list'] = list
```

1.4.5.2 Number of Total and Ignored Patches

This measurement was elected based on expert knowledge.

Description `total` is the number of total patches sent affecting the list. `ignored` is the number of ignored patches sent affecting the list.

In addition, one can separate the numbers by time (by year, kernel version).

Motivation `total`, `ignored`, and `upstream` can be used to get a first impression of the size of the list. Additionally, `total` can be used to normalize the `ignored` and the `upstream` values.

When one add the time dimension to the measurements (e.g. `total-<year>`, and `ignored-<year>`), one can conduct time-series analysis as Yin [170] suggests.

How to measure We iterate over all patches affecting the list.

- For each patch, we increase `total`
- For each patch with the `ignored`-flag, we increase `ignored`
- Each measurement can be separated by time

```
for patch in patches:
  add_or_create(data, 'total')
  if patch['ignored']:
    add_or_create(data, 'ignored')
```

1.4.6 Measurements of Subsystems

To create the dataset for analyzing the subsystems, one needs the patch-data as described in section1.4.3. We describe the source code extensively Thereby, we avoid the spot test. Nevertheless, we conduct a plausibility check. For simplicity, we group the patches by the subsystems.

```
def prepare_subsystems(patch_data):
  subsystems = dict()
  for patch in patch_data:
    for subsystem in patch['subsystems']:
      subsystem_list = get_most_current_maintainers(subsystem).list
      if subsystem.list in patch['recipients']:
        add_or_create(subsystems, subsystem, [patch])
  return subsystems
```

Afterwards, one iterates over the returned dictionary and calls the `get_subsystem_information` method:

```
  prepared_subsystem_data = prepare_subsystems(patch_data)
  subsystem_data = subsystem()
  for subsystem, patches in prepared_subsystem_data.items():
    subsystem.append(get_subsystem_information(subsystem, patches))
```

In the function `get_subsystem_information`, a dictionary is created with information about the subsystem.

```
def get_subsystem_information(subsystem, patches):
  data = dict()
  …
```

1.4.6.1 Name of Subsystem

This measurement was elected based on expert knowledge.

Description This measurement is the name of the subsystem. It can be used for identifying the subsystem.

Motivation To work with the subsystems' data, an identifier is required.

How to measure The `MAINTAINERS`-file states a name per subsystem. This name is unique and can be used as identifier.

```
data['subsystem'] = subsystem
```

1.4.6.2 Total and Ignored Patches

This measurement was elected based on expert knowledge.

Description `total` is the number of total patches sent affecting the subsystem. `ignored` is the number of ignored patches sent affecting the subsystem.

When one add the time dimension to the measurements (e.g. `total-<year>`, and `ignored-<year>`), one can conduct time-series analysis as Yin [170] suggests.

Motivation `total`, and `ignored` can be used to get a first impression of the size of the subsystem. Additionally, `total` can be used to normalize the `ignored` values.

`total`, and `ignored` with the time-dimension can be used for analyses over time.

How to measure We iterate all patches affecting the subsystem.

- For each patch, we increase `total`
- For each patch with the `ignored`-flag, we increase `ignored`
- Each measurement can be separated by time

```
for patch in patches:
    add_or_create(data, 'total')
    if patch['ignored']:
        add_or_create(data, 'ignored')
```

1.4.6.3 Status

This measurement was elected based on expert knowledge.

Description (Almost) each entry in the `MAINTAINERS`-file has a status. If available, the status proposed. Otherwise, an empty string is proposed.

Some entries in the `MAINTAINERS`-file have multiple statuses. They are proposed comma-separated.

Possible statuses are [102]:

- Supported
- Maintained
- Odd Fixes
- Orphan
- Obsolete

The status proposed is the subsystem's status in the current version if the subsystem is existing in this version. Otherwise, the status of the latest appearance of the subsystem in the `MAINTAINERS`-file is used.

Motivation The status of the subsystem is an indicator of how good a subsystem is maintained. As the status might correlate with the ignored patch frequency, the status is key information for the analyses.

How to measure Entries in the `MAINTAINERS`-file may contain a line starting with `'S:'`. This tag is succeeded by the status-keyword, which is extracted. PaStA extracts the statuses as a list, thus, we have to join them. To extract the status from the subsystem from the latest `MAINTAINERS`-file version containing the subsystem the function `get_most_current_maintainers` (The definition can be found in the Appendix; see section A.2) is required.

```
data['status'] = ', '.join(map(lambda s: s.value,
get_most_current_maintainers(subsystem).status))
```

1.5 Analysis

In this section, we will elaborate on the previously described datasets. First, we quantify the phenomenon of ignored patches. Second, we search for characteristics of patches that are ignored, followed by the analysis of authors. Concludingly, we examine the mailing list and the subsystem dataset. In each subsection, we present respective analyses and discuss the insights gained in the analyses.

1.5.1 Quantification of Ignored Patches

1.5.1.1 Quantification

In total, developers submitted about 792k patches to the LKML in our analyzed time frame. From the 792k patches, 18k were ignored; about 2,3%.

1.5.1.2 Developement over Time

In this section, we analyze the data mined based on the time, as Yin suggests [170]. To answer this RQ, we first determined the values per year:

2011	3.2%	**2013**	2.2%	**2015**	1.6%	**2017**	1.8%
2012	2.7%	**2014**	2.0%	**2016**	2.4%	**2018**	1.2%

We were surprised by the decreasing trend, so we plotted the data over time (see figure 1.5). The graph shows the total/ignored patches per week in absolute values. The smooth lines are `t-based approximations` [65] of the respective lines. The `y`-scale is a `sqrt`-scale, as suggested by Tukey [155]. This description applies to all following graphs, if not stated differently.

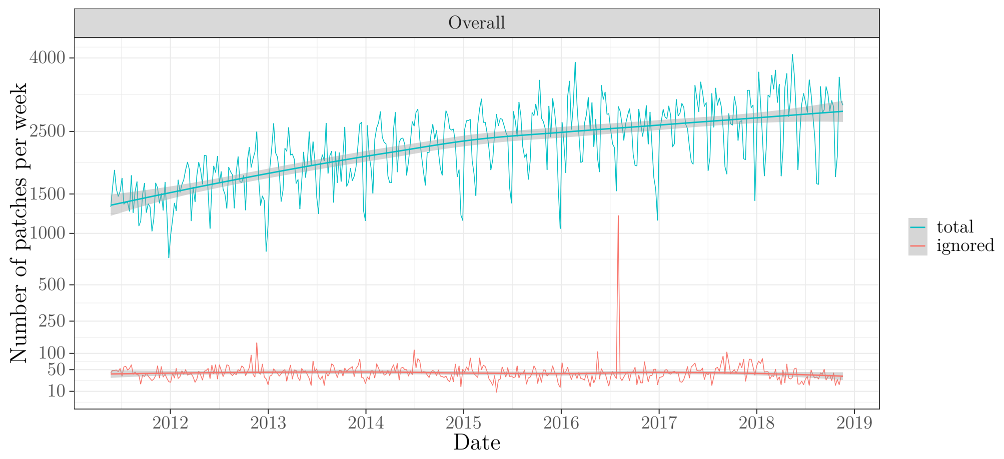

Figure 1.5: Ignored Patches over Time (uncleaned)

One can see one spike in the third quarter of 2016. This spike is one patch series with almost 1300 patches. Due to technical errors (by the sender), the patch series was not sent in one thread. Thus, our patch series detection failed.

This patch series replaces a numeric value stating permissions of drivers with the corresponding macros (see example below). The patch series was rejected with the comment that people working in this domain can read the numeric value more easily than the macros [2].

```
-module_param_array(index, int, NULL, 0444);
+module_param_array(index, int, NULL, S_IRUSR | S_IRGRP | S_IROTH);
```

For our further analyses, we exclude this author. Besides this patch series, the developer submitted one patch (the patch was sent three times), which was not ignored due to our metrics (requirement 3 is not met; one of the three patches was not ignored). As a result, the exclusion does not affect our data noticeably. After cleaning the data, the graph in figure 1.6 is plotted.

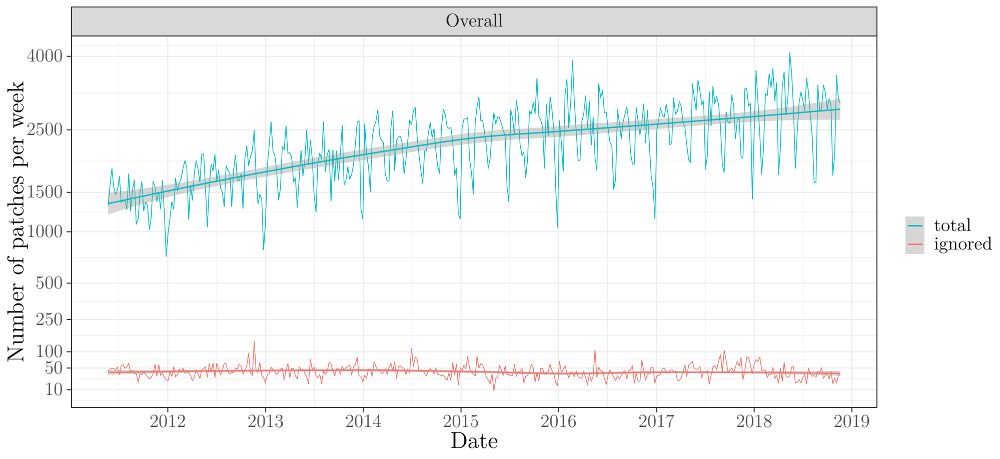

Figure 1.6: Ignored Patches over Time (cleaned)

The graph (figure 1.6) shows an increasing amount of contributions sent to the analyzed mailing lists. In the analyzed time frame, they grew from below 1500 contributions per week to 3500 contributions per week. The maximum number of patches sent to the mailing lists in a week is 4080. Besides, one can see the spike downwards at the end of the year. The observations correlate with Christmas/holidays. One can further see reoccurring downward peaks 5 to 6 times per year. In figure 1.7, vertical lines were added at the release dates of a kernel version. One can easily see the correlation between the spikes and the releases.

After the release the frequency of patches is decreasing. This matches the process documented. After a release the merge window opens. In this time frame the maintainers merge the new features into Linus' upstream repository. Because, the maintainers are engaged working on the merging, they are less open to new patches.

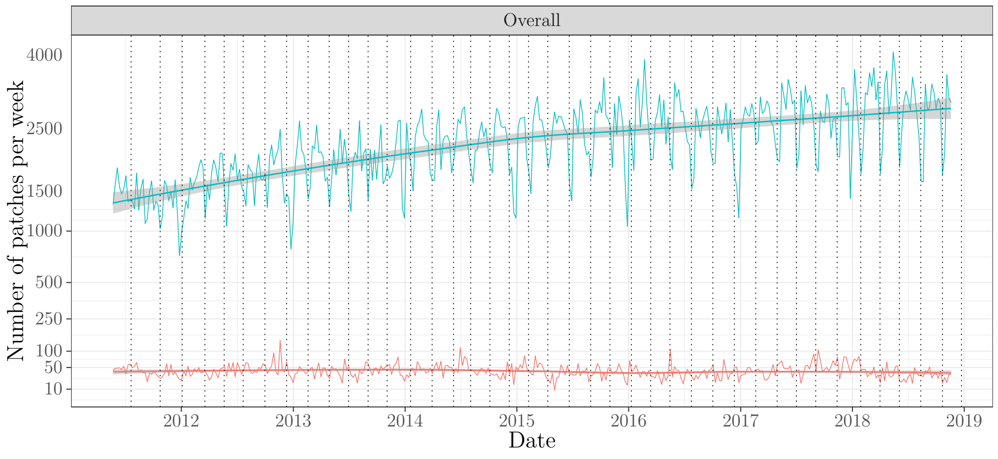

Figure 1.7: Ignored Patches over Time with Releases

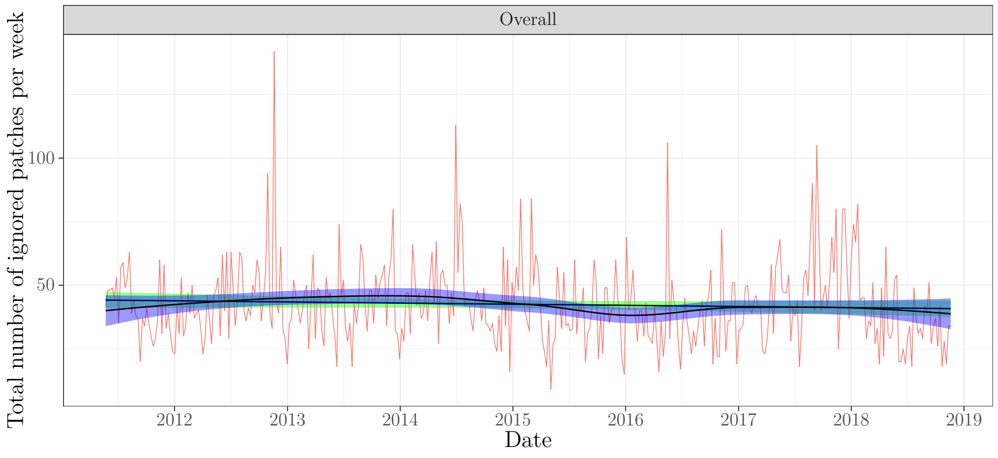

Figure 1.8: Only Ignored Patches over Time

When focusing on the ignored patches (see figure 1.8), one can see the consistency of the number of ignored patches. In the analyzed time frame, the number of ignored patches per week is quite constant The number of ignored patches is mostly between 25 and 75 per week.

One can see spikes of over 100 ignored patches per week. These spikes are (multiple) patch series. In 2014, for example, there was a patch series with close to 50 patches which was ignored [82]. In 2017, there were about 100 patches by one author ignored. The patches were separate patches, some were grouped in small patch series [56].

Figure 1.9 shows the weeks placed by the number of patches sent in the week (x-axis) and the number of ignored patches (y-axis). The weeks' distribution is

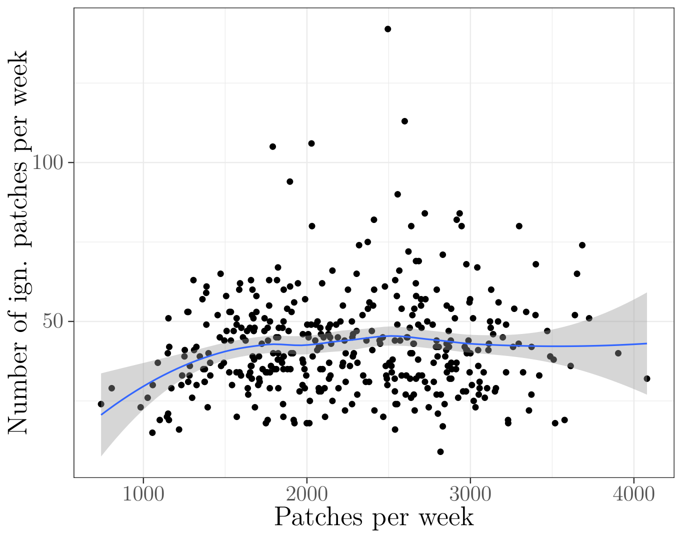

Figure 1.9: Weeks by Number of Patches Sent and Number of Ignored Patches

not correlated as the number of ignored patches is only very little based on the number of total patches.

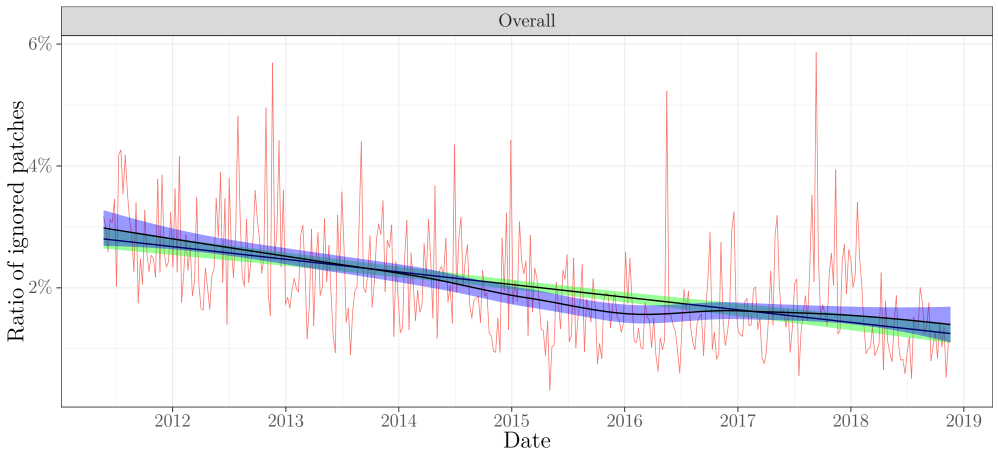

Figure 1.10: Ratio of Ignored Patches over Time

At the beginning of the section, we stated the ratio of ignored patches in the years. Figure 1.10 shows the ratio of ignored patches per week over the analyzed time frame. The numbers stated match the graph; the number is decreasing. There is a trivial explanation: The number of total patches increases, and the number of ignored patches remains constant. Thus, the ratio is decreasing.

Linux Kernel Release Cycle Does it matter when a patch is sent during the release cycle?

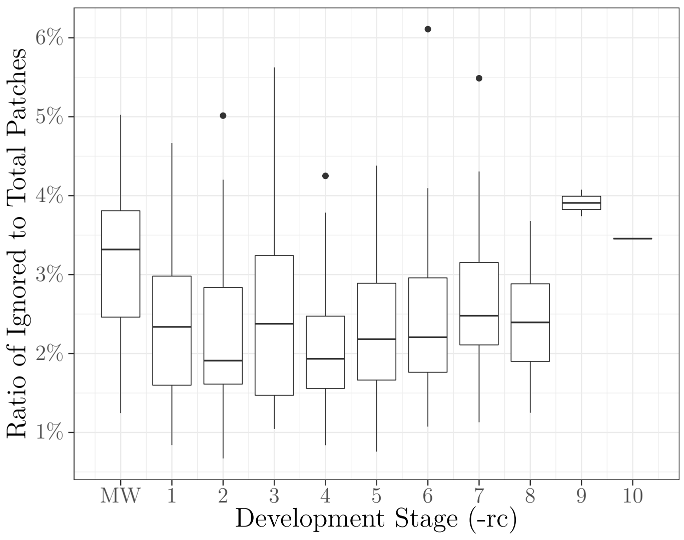

Figure 1.11: Distribution of Ratio Ignored to Total Patches Grouped by Development Stage

In figure 1.11, one can see the ratio of ignored patches grouped by the phase in the kernel development. The boxplot shows that it is mainly independent of the development stage if a patch is ignored. There is a higher chance of being ignored if the patch is sent during the merge window. An explanation for this can be that the maintainers have a higher workload at this time (as explained above). Thus, they have less time to care about new patches and patches are more easily ignored.

The number of measured points of `-rc8`, `-rc9`, and `-rc10` is not equal to the other stages. This is because usually, a new kernel is released after `-rc7`. As a result, there are not many `-rc8`s, `-rc9`s, and `-rc10`s.

1.5.1.3 Summary

In this section, we learned about the occurrence of ignored patches to answer RQ1 (How many patches are ignored in the Linux kernel development? And How is the rate of ignored patches developing over time?):

From 792k patches, 18k (2.3%) were ignored in the analyzed time frame. The ratio of the ignored patches is continuously decreasing as the number of patches is increasing, and the amount of ignored patches is constant. The number of patches ignored per week is almost independent of the number patches sent in the week. Patches sent in the MW, are a little more likely to be ignored than patches sent in the other development stages.

1.5.2 Characteristics of Ignored Patches

1.5.2.1 Size

As the first step, we want to analyze the size of patches. In the analyses, we found out that the size is no suitable measurement to distinguish ignored and not-ignored patches.

Size of Patch To get a first impression on the size of patch measurement, we did some basic analyses (see 1.18).

quantile	not-ignored	ignored
0.00	0	0
0.25	6	3
0.50	20	8
0.75	65	27
1.00	93269	26161

Table 1.18: Size of Patch - Five Number Summaries [155]

The average size of a not-ignored patch is 98.5 lines of code (loc), the average size of an ignored patch is 66.4 locs. The size of a patch is measured by counting the lines of the patch (without patch message). Because, the size of patch detection failed at some points. As a result, some patches (588) claim to have the patch-size `0`. As these are no patches with extremes, they will not affect further analyses.

The most comprehensive patches are affecting whole drivers. Some are staging in drivers [59]. Some are moving drivers [105]. Some are staging out drivers [61]. All patches changing more than $50,000$ locs were sent to the `netdev` mailing list.

The most comprehensive ignored patches are comparable to the abovementioned categories (staged in [49], staged out [60], or moved [57]).

In figure 1.12 the number of not-ignored and ignored patches (y-axis) with the size (x-axis) is plotted. The Pearson-correlation-coefficient of the ratio of both numbers and the corresponding size of the patches (see section D for the source code) is -0.04. The value of the coefficient close to zero expresses that the size of patch measurement is no characteristic of an ignored patch. All Pearson-correlation-coefficients in the succeeding subsections are calculated this way.

Number of Lines Added To get a first impression on the size of patch measurement, we did some basic analyses (see 1.19).

The average number of lines added in a not-ignored patch is 66.5 locs, the average number of chunk added in an ignored patch is 41.4 locs.

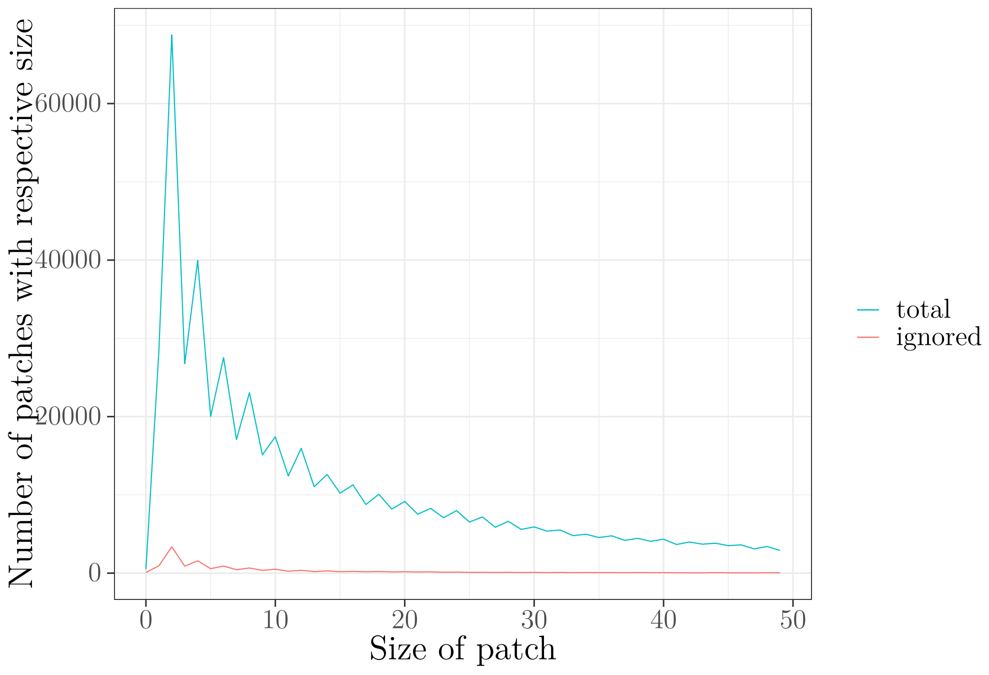

Figure 1.12: Number of Patches with respective Size

quantile	not-ignored	ignored
0.00	0	0
0.25	3	1
0.50	11	4
0.75	39	15
1.00	44379	10027

Table 1.19: Number of Lines Added - Five Number Summaries [155]

As expected, the patches with the highest number of added lines are patches adding driver support [50] or moving drivers [105]. Like noticed during the analysis of the lines of code changed in a patch, the ignored patches adding the most code can be grouped in the same abovementioned categories (add driver [49], move driver [57]).

The Pearson-correlation-coefficient of the ratio of the numbers of patches with an equal number of lines added and the corresponding number of lines added is -0.07. The value of the coefficient close to zero expresses that the new-lines measurement of the patch is no characteristic of an ignored patch.

Number of Lines Removed To get a first impression on the size of patch measurement, we did some basic analyses (see 1.20).

quantile	not-ignored	ignored
0.00	0	0
0.25	0	1
0.50	2	2
0.75	11	6
1.00	63036	23429

Table 1.20: Number of Lines Removed - Five Number Summaries [155]

The average number of lines removed in a not-ignored patch is 20.0 locs, the average number of chunk removed in an ignored patch is 17.7 locs.

When analyzing the patches removing the most files, one recognizes the same categories as before:

- Not ignored patch staging out a driver [60]
- Not ignored patch moving a driver [105]
- Ignored patch staging out a driver [60]
- Ignored patch moving a driver [57]

The Pearson-correlation-coefficient of the ratio of the numbers of patches with an equal number of lines removed and the corresponding number of lines removed is -0.05. The value of the coefficient close to zero expresses that the lines-removed measurement of the patch is no characteristic of an ignored patch.

Number of Files Touched To get a first impression on the size of patch measurement, we did some basic analyses (see 1.21).

quantile	not-ignored	ignored
0.00	0	0
0.25	1	1
0.50	1	1
0.75	3	1
1.00	1206	812

Table 1.21: Number of Files Touched - Five Number Summaries [155]

The average number of files touched by a not-ignored patch is 2.6 files, the average number of files touched in an ignored patch is 2.0 files. Because, the size of patch detection failed at some points as a result, some patches (382) claim to have the patch-size `0`. As these are no patches with extremes, they will not affect further analyses. Most of the patches are (re)moving drivers or (semi-)automated[18].

- Add License tags to drivers (automated) [88, 87]
- Remove import after removing a file (automated) [74]
- Remove toggle from configuration [36]
- Move video-drivers [156]

When analyzing the ignored patches, one finds two categories (move driver, false positive):

- Move documentation/driver [29, 28]
- False-positive (undetected process mails) [66, 89]
- False-positive (undetected patch series due to missing reference) [117]

The Pearson-correlation-coefficient of the ratio of the numbers of patches with an equal number of files touched and the corresponding number of files touched is -0.13. The small value of the coefficient expresses that the number of files touched measurement slightly indicates the ignoring of the patch.

1.5.2.2 Recipients

In this subsection, we will investigate the effects of the recipients (`to:` and `cc:` header). We found that a high number of recipients implies that a patch is not ignored and that patches sent to the LKML are more likely be ignored. However, we advise to follow the `get_maintainers.pl`-script and not to send the patch to uninvolved developers, maintainers or worse mailing lists just to increase the number of recipients.

Number Recipients To get a first impression on the size of patch measurement, we did some basic analyses (see 1.22).

quantile	not-ignored	ignored
0.00	0	0
0.25	2	2
0.50	5	4
0.75	8	7
1.00	275	51

Table 1.22: Number of Recipients - Five Number Summaries [155]

The average number of recipients of a not-ignored patch is 5.8 recipients, the average number of files touched in an ignored patch is 5.5 recipients. Due to missing `to` headers in the mail (e.g. [52]), some patches (466) have no recipients.

The Pearson-correlation-coefficient of the data series and the corresponding number of recipients of the patches is -0.07. The value of the coefficient close to zero

[18]Probably all patches are automated, but only some contributors stated this explicitly.

expresses that the number of recipients measurement of the patch is no characteristics of an ignored patch.

Number of Non-List Recipients To get a first impression on the size of patch measurement, we did some basic analyses (see 1.23).

quantile	not-ignored	ignored
0.00	0	0
0.25	0	1
0.50	3	3
0.75	6	5
1.00	238	47

Table 1.23: Number of Non-List Recipients - Five Number Summaries [155]

The average number of non-list recipients of a not-ignored patch is 4.2 recipients, the average number of recipients of an ignored patch is 3.9 recipients.

The Pearson-correlation-coefficient of the data series and the corresponding number of non-list recipients of the patches is -0.10. The small value of the coefficient expresses that the number of non-list recipients measurement of the patch indicates the ignoring of the patch.

Number of List Recipients To get a first impression on the size of patch measurement, we did some basic analyses (see 1.24).

quantile	not-ignored	ignored
0.00	1	1
0.25	1	1
0.50	1	1
0.75	2	2
1.00	37	11

Table 1.24: Number of List Recipients - Five Number Summaries [155]

The average number of list recipients of a not-ignored patch is 1.6 recipients, the average number of list recipients of an ignored patch is 1.6 recipients.

The Pearson-correlation-coefficient of the data series and the corresponding number of list recipients of the patches is -0.70. The value of the coefficient expresses that the number of list recipients measurement of the patch is a significant characteristics of an ignored patch.

As over 50% of the patches are only addressed to one list this is no good criterion. The number of list recipients measurement can be used as disqualifier but not for the identification of ignored patches.

Sent to LKML To get a first impression on the size of patch measurement, we did some basic analyses (see 1.25).

	not-ignored	ignored	total
true	378021	11886	389907
false	396531	5922	402453
total	774552	17808	**792360**

Table 1.25: Sent to LKML - Confusion Matrix

Two third of all ignored patches were sent to the LKML but only half of the not-ignored patches were sent to the LKML. 3% of the mails sent to the LKML were ignored, however, the ratio of ignored mails not sent to the LKML is 1.5%.

This indicates that patches sent to the LKML are more likely to be ignored than patches not sent to the LKML. Patches sent to the LKML might be ignored more often as the LKML is a high traffic mailing list with over 300 mails per day [106].

Correct Maintainers To get a first impression on the size of patch measurement, we did some basic analyses (see 1.26).

	not-ignored	ignored	total
true	685437	15722	701159
false	89115	2086	91201
total	774552	17808	**792360**

Table 1.26: Sent to Correct Maintainers - Confusion Matrix

11.7% of the ignored patches were not sent to the correct maintainers, similar 11.5% of the not-ignored patches were not sent to the correct maintainers. 2.3% of the patches not sent to the correct maintainers were ignored, similar 2.2% of the patches sent to the correct maintainers were ignored-

Surprisingly, we see that patches are equally likely ignored independent of the correct addressing of the patch.

1.5.2.3 Tags

In this subsection, we will investigate the effects of the tags found in the patch. We found that a high number of tags implies that a patch is not ignored. As the number of tags is determined by the development process of the patch, one cannot easily influence the tags. Furthermore, there is a default situation (one `signed-off-by`, none `acked-by`, and none `co-developed-by`). As a result, a patch with more tags than the default situation is likely not ignored, but the default situation does not help us to distinguish.

Number of Signed-Off-Bys To get a first impression on the size of patch measurement, we did some basic analyses (see 1.27).

quantile	not-ignored	ignored
0.00	0	0
0.25	1	1
0.50	1	1
0.75	1	1
1.00	230	6

Table 1.27: Number of Signed-Off-Bys - Five Number Summaries [155]

The average number of signed-off-by tags of a not-ignored patch is 1.2 tags, the average number of signed-off-by tags in an ignored patch is 1.1 tags.

The Pearson-correlation-coefficient of the data series and the corresponding number of signed-off-bys of the patches is -0.51. The value of the coefficient expresses that the number of signed-off-bys measurement of the patch is a significant characteristic of an ignored patch.

As over 75% of the patches have only one tag or less this is no good criterion. The number of signed-off-bys can be used as disqualifier but not for the identification of ignored patches.

Number of Acked-Bys To get a first impression on the size of patch measurement, we did some basic analyses (see 1.28).

quantile	not-ignored	ignored
0.00	0	0
0.25	0	0
0.50	0	0
0.75	0	0
1.00	101	7

Table 1.28: Number of Acked-Bys - Five Number Summaries [155]

The average number of acked-bys by a not-ignored patch is 0.1 tags, the average number of acked-bys in an ignored patch is 0.0 tags.

The Pearson-correlation-coefficient of the data series and the corresponding number of acked-bys of the patches is -0.20. The small value of the coefficient expresses that the number of acked-bys measurement of the patch indicates the ignoring of the patch.

As over 75% of the patches have no acked-by tag this is no good criterion. The number of acked-bys can be used as disqualifier but not for the identification of ignored patches.

Number of Co-Developed-Bys To get a first impression on the size of patch measurement, we did some basic analyses (see 1.29).

quantile	not-ignored	ignored
0.00	0	0
0.25	0	0
0.50	0	0
0.75	0	0
1.00	5	1

Table 1.29: Number of Co-Developed-Bys - Five Number Summaries [155]

The average number of co-developed-bys by a not-ignored patch is 0.0 tags, the average number of co-developed-bys in an ignored patch is 0.0 tags.

The Pearson-correlation-coefficient of the data series and the corresponding number of co-developed-bys of the patches is -0.77. The value of the coefficient expresses that the number of co-developed-bys measurement of the patch is a significant characteristic of an ignored patch.

As over 75% of the patches have no co-developed-by tag this is no good criterion. The number of co-developed-bys can be used as disqualifier but not for the identification of ignored patches.

1.5.2.4 Summary

In this section, we learned about the characteristics of ignored patches to answer RQ2 (What are the unique characteristics of ignored patches in the Linux kernel development?):

We found that some metrics indicate if a patch is ignored or not, but the metrics are not expressive enough to distinguish ignored or not-ignored patches. Small patches are more likely to be ignored than big patches and that a patch with many recipients is less likely to be ignored and that patches sent to the LKML are more likely to be ignored. However, we suggest developers: send the patch to the recipients proposed by the `get_maintainers.pl`-script (including the LKML) and related other maintainers and developers. Big patches are less likely ignored but probably more likely rejected. A high number of tags means a patch is unlikely to be ignored. But most of the patches only have a `signed-off-by`-tag when they are submitted to the mailing lists. Thus, the number of tags is no useful metric as well to identify ignored patches. A summary of the indicators is: the more effort is in a patch (collaboration of developer (e.g. multiple tags, recipients), bigger changes (e.g. by lines of code, recipients)) the less likely it is ignored.

There is no characteristic of the patch, that enables us to reliably recognize ignored patches. These evaluations show that ignored patches are distributed almost independent of specific characteristics. In the following section, we want to analyze if patches of some authors are more likely to be ignored than others.

1.5.3 Analyses of Authors

In this section, we will analyze the authors of the patches. We will look at the experience of authors and identify authors who are often ignored. Afterwards, we analyze if the `tld` or the `organization` identifiable by the mail address is correlated with being ignored. Concludingly, we analyze if there is discrimination of authors based on their gender or country of origin.

In the gender and country analyses, one notices that the total number of patches is not the total number of patches one can find in the previous analyses. We excluded some contributors from this analysis, as not all contributors stated a name. Thus, we were not able to determine the gender or the country of origin. There were about 38k contributions without a name stated (about 5%). We assume that the omission of the developers without a stated name does not introduce bias.

1.5.3.1 Experience

Figure 1.13 shows a plot of all authors who contributed to the kernel in the stated time frame. The plot places a dot per author based on the total patches sent (x-axis) and the ignored patches (y-axis). The plot shows that it is a common phenomenon to have ignored patches.

There are nine contributors with over 100 ignored patches, they account for about 3000 ignored patches (17% of all ignored patches). The developers who are often ignored can be grouped into the following clusters:

- Maintainers [73]
- Minor contributions (`check_patch.pl` issues, typos) [86, 169]
- (Semi-)Automatically created patches [146]
- Others [92]

Figure 1.14 shows a plot of all authors who contributed to the kernel in the stated time frame. The plot places a dot per author based on the mails sent by the author (x-axis) and the ignored patches (y-axis). The plots show that the

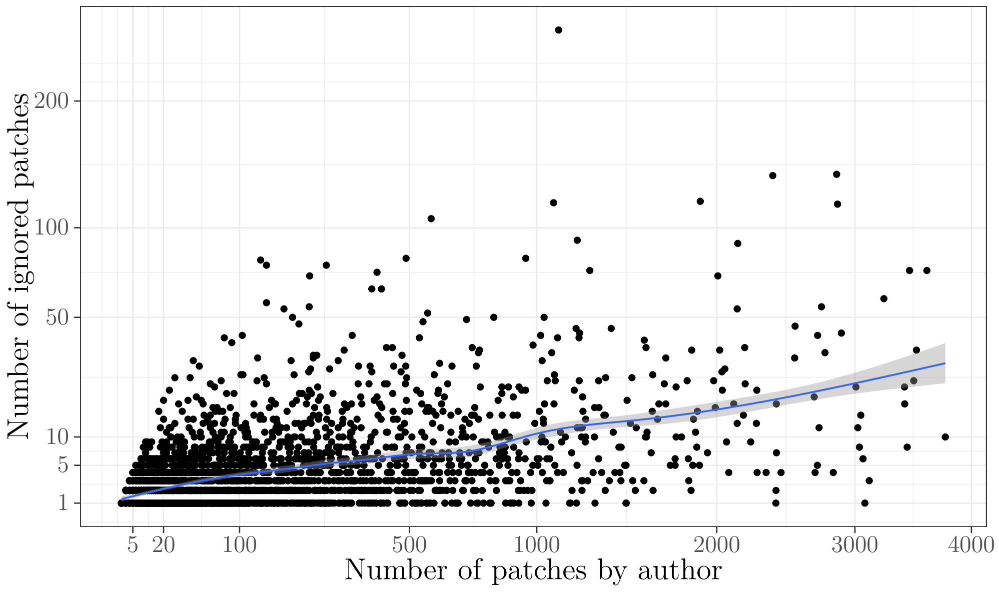

Figure 1.13: Total and Ignored Patches per Author by Patches Sent

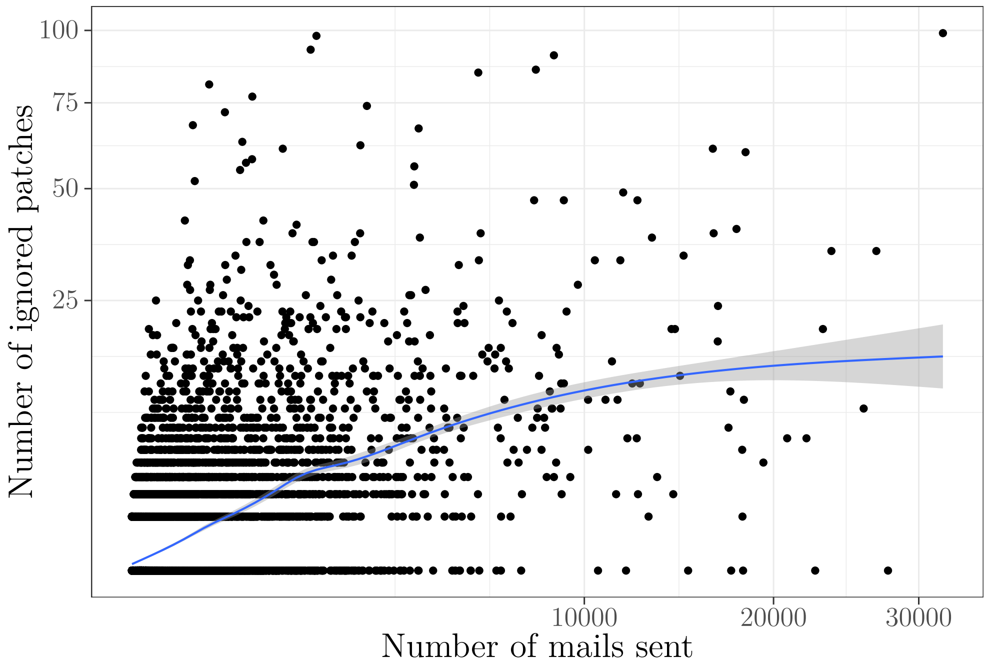

Figure 1.14: Total and Ignored Patches per Author by Mails Sent

average amount of patches, trivially, increases with the number of contributions to the mailing list. However, figure 1.15 shows that the ratio of ignored patches is decreasing with more involvement. The plot places a dot per author based on

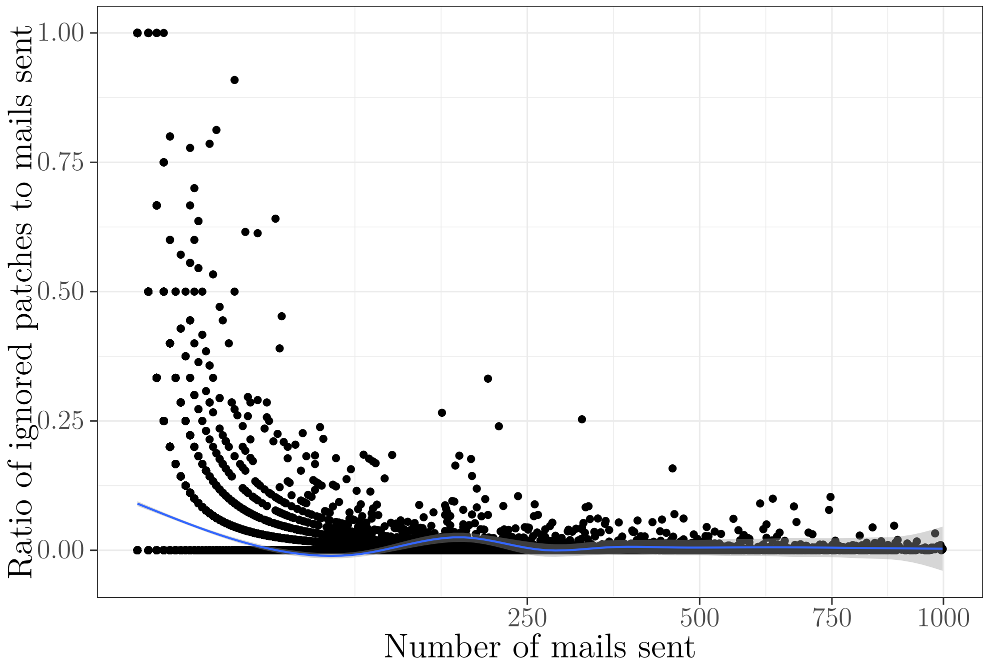

Figure 1.15: Ratio of Ignored Patches to Mails Sent by Author by Mails Sent

the mails sent by the author (x-axis) and the ratio of ignored patches to the mails sent (y-axis).

1.5.3.2 TLD

As figure 1.16 shows, the top TLDs in the Linux kernel development are:

1. `.com`
2. `.org`
3. `.de` - Germany
4. `.net`
5. `.au` - Australia

Most of the TLDs' ratios of ignored patches to total patches are between 1% and 3%. However, when looking at the TLDs with more than 1000 patches sent in the analyzed time frame one finds some with a significantly higher ratio:

- `.se` - 8%
- `.cn` - 9%

The high ratio of ignored patches from Sweden is attributable to automatically created patches [146]; without these patches, the ratio is 1%. We did not find a sound reason why patches from China are more often ignored than others. Further, when conducting the χ^2-test (see appendix), one finds out that the

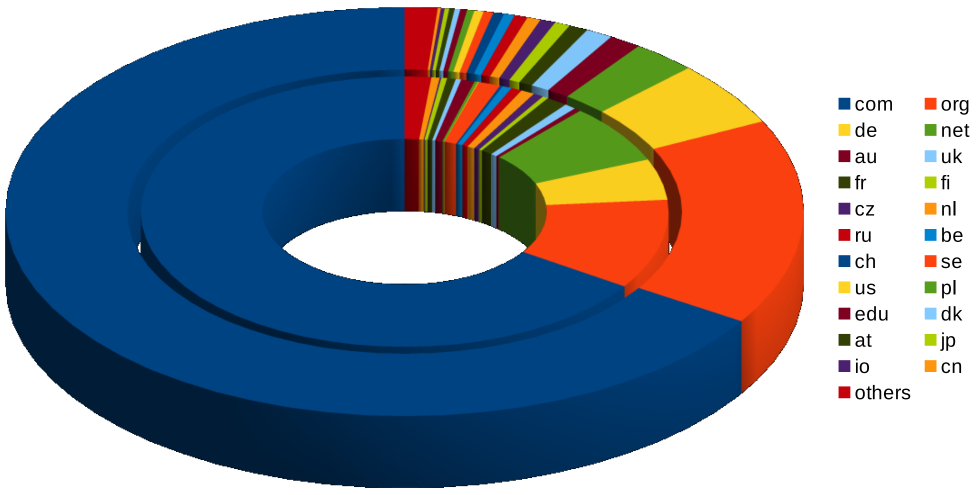

Figure 1.16: Pie Chart Based on the Total and Ignored Patches Sent from a Mail-Address' tld

`.cn`-tld and the ignoring of patches correlates. This could be an indicator for discrimination, further research is required.

1.5.3.3 Organization

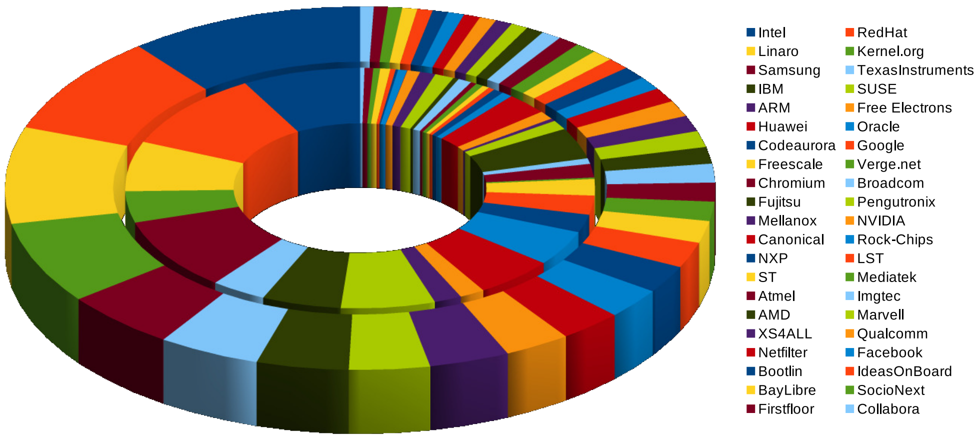

Figure 1.17: Pie Chart Based on the Total and Ignored Patches Sent from a Mail-Address' Domain

In figure 1.17 one can see the organizations contributing the highest numbers of patches to the Linux kernel. Expectably, companies like Intel, RedHat, and Suse are represented on this chart.

When calculating the ratios of the ignored patches per organization, one notices that most of the organizations' ratios are significatly below the generic ratio of 2.3% of all patches sent to the mailing lists. This can be explained by more experienced developers However, there are two companies with a significantly higher ratio of ignored patches. One company employs a person known to send often ignored typo-fixes. When we exclude the author the company becomes ordinary. For the other company[19] we did not find an explanation why the patches are more often ignored.

1.5.3.4 Gender

To get a first impression on the issue, we build the confusion matrix[20] (see table 1.30).

	not-ignored	ignored	total
male	711141	16045	727186
not-male	26527	903	27430
total	737668	16948	**754616**

Table 1.30: Gender - Five Number Summaries [155]

One can instantly see, that most contributions are done by men. Trivially, the majority of ignored patches are contributed by men as well.

However, when looking at the conditional probabilities one notices the following:

$$P(male \mid ignored) = 2.2\%$$
$$P(not\ male \mid ignored) = 3.3\%$$

The conditional probability of being ignored when being not-male is 50% higher compared to when being male.

When conducting a χ^2-test (in the appendix the calculation can be found), we found out that being ignored and being not male are correlating. The insight, combined with the assumption that female and diverse developers' contributions are of the same quality as male's, indicate the discrimination of non-male developers.

We think assessing the gender by name is not the most accurate method, however, when collaborating over a mailing list the name is the only indication one has. To use the same indication to estimate the gender evokes to make the same mistakes. This improves the quality of the results.

[19]We do not write the company's name to not denounce them.

[20]To achieve a better data base, we only distinguish between male and not-male in the analysis.

1.5.3.5 Country

As first step, we will analyze the regions of origin of the contributors (see table 1.31. One noticed that North America is missing in the table below. This can be explained by the fact that the Norther American states, Australia, and New Zealand are primarily inhabited by people with names from one of the other regions, mostly form the European cultural area.

region	ignored	total	ratio
Africa	143	2656	0.05
Asia	6283	217785	0.03
Europe	10471	533772	0.02
Latin America	51	403	0.13

Table 1.31: Ignoring by Region

Comparable to the analysis if developers are discriminated due to their gender, this analysis reveals that the conditional probability of some regions are significantly higher than the generic probability that a patch is ignored (2.3%). When conducting the χ^2-test, one can see that being ignored and the region are depending variables.

When looking at the countries, the values are either on an average level or the countries have a too small dataset for comparison.

1.5.3.6 Ignored First Submission

In their research, Steinmacher et al. [144] and Jensen et al. [80] show that the first interaction of the community with new members is key for the further activity of the newcomer. Williams [166] showed that the exclusion and ignoring amplifies negative emotions. As a result, the ostracizing by ignoring might discourages volunteers to continue their participation in the project. In this subsection, we analyze if it affects further activity if the first patch is ignored (see table 1.32).

	single contribution	multiple contributions
first is ignored	459	596
first is not ignored	3454	10467

Table 1.32: Abandonment of Project if First Contribution is Ignored

From the confusion matrix, one can derive the following conditional probabilities:

$$P(first\ is\ ignored \mid multiple\ contributions) = 56.7\%$$

$$P(first\ is\ not\ ignored \mid multiple\ contributions) = 75.2\%$$

According to the χ^2-test both variables are statistically correlating (see section C.1 in the appendix for the elaborated test).

We can agree with Steinmacher et al. and Jensen et al. that the first response has an influence on the further activity of the newcomer. We were able to show, that ignoring the first contribution correlates with the abandonment of the project.

1.5.3.7 Summary

In this section, we learned about ignored patches from the author's point of view to answer RQ3 (What discrimination is taking place in the Linux kernel development by ignoring patches?):

In this section, we identified two classes of patches (minor patches, and automatically created patches) which are often ignored. We found indications for discrimination based on the gender and the mail-address' top-level-domain. Further research is required to examine these claims. Concludingly, we found indications that if the first contribution is ignored the contributor abandons the project.

1.5.4 Analyses of Mailing Lists

As we analyze many mailing lists, we will select interesting clusters in the following section and describe each cluster. The global trend is that the frequency of patches sent is increasing, and the frequency of ignored patches is stagnating or even decreasing.

1.5.4.1 Growing Mailing Lists

At first, we want to look at the mailing lists that grow and thereby face the issue of a maintainer overload. Figure 1.18 shows the plots of the mailing lists `arm`, `btrfs`, `fsdevel`, and `netdev`. All four mailing-lists grew (2011-2018): the `arm`-list grew from 200 to 500 patches per week[21]; the `btrfs`-list grew from 20 to 50 patches per week[22]; the `fsdevel`-list grew from 30 to 100 patches per week[23]; the `netdev`-list grew from 150 to 500 patches per week[24].

However, in all mailing lists, the frequency of ignored patches stayed the same or even decreased (see `btrfs`). The abovementioned insights suggest that the

[21]Due to the increasing importance of ARM in smartphones, cloud applications, and single-board computers (Raspberry PI) the ARM-architecture gained more attention.

[22]BTRFS is a new filesystem which is in the kernel since 2007. Thus, BTRFS is under active development and gained users since.

[23]Big Data applications, cloud storage, etc. need enormous memory capacities with improved performance. To achieve both, the file-systems are continuously improved.

[24]The usage of Linux in cloud applications strengthens the need for sophisticated network support. As a result, the mailing list grew.

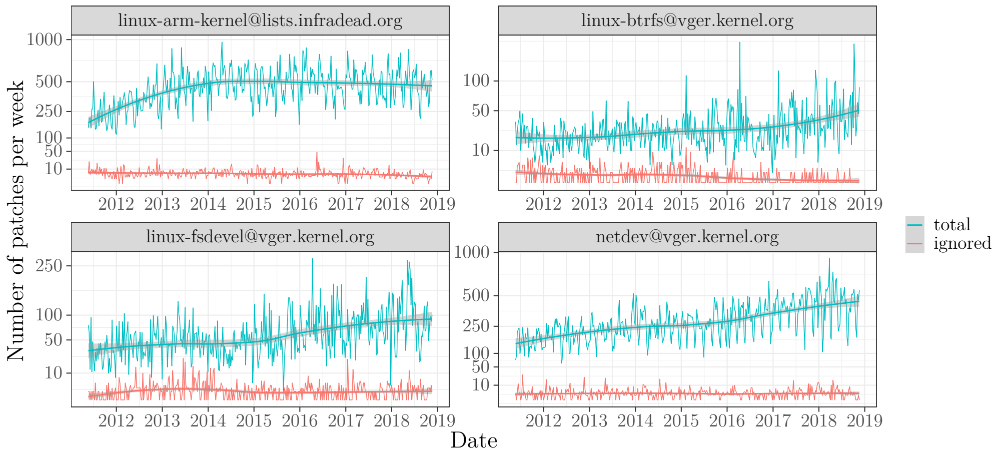

Figure 1.18: Ignored and Total Patches of growing Mailing Lists

process lived in the kernel development scales. Despite the increasing number of patches sent per week, the number of ignored patches per week is stagnating. This contradicts the insights from Wolfgang Sang [141, 140].

To conciliate both findings, we suggest more research on the content of the answers from the maintainers. An explanation could be that it takes more time to answer, maintainers are answering some patches automatically or that the feedback is less sophisticated. As we did not analyze the patches' content and the responses, we cannot underpin this hypothesis.

1.5.4.2 Effect of the Number of Maintainers and Reviewers

In the abovementioned analysis, we were not able to show the overload of maintainers, respectively, the Linux kernel development process scales (according to our metric). According to Zhou et al. [171] increasing the number of maintainers improves the manageable workload. In figure 1.19 and 1.20, the history of the `arm`- and the `bluetooth`-mailing lists are plotted. One can see the increase of maintainers who are active in the mailing lists. In Zhou's paper, the authors state that the manageable workload scales with factor one half. The golden line is the square-root of the number of maintainers.

Until 2013, the activity on the `arm` mailing list is increasing. The observation matches the insights from Zhou et al. [171]. However, the activity on the `arm` mailing list stagnates between 2014 and the end of the analysis, but the number of maintainers is still increasing. This would, according to Zhou et al., reduce the workload of the maintainers. There is a constant trend in the whole analyzed time frame that the number of ignored patches is decreasing. The trend before 2014 and from 2014 is continuous.

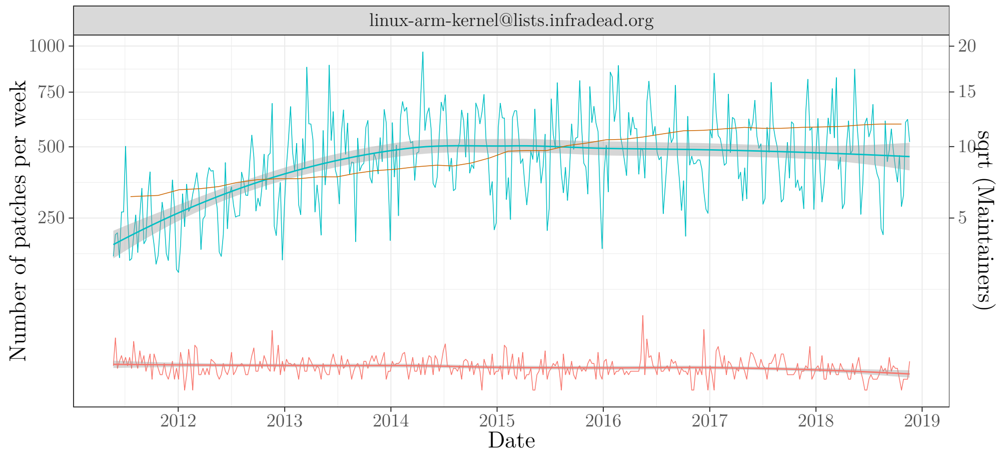

Figure 1.19: Patches per Week on ARM Mailing List with Maintainers over Time

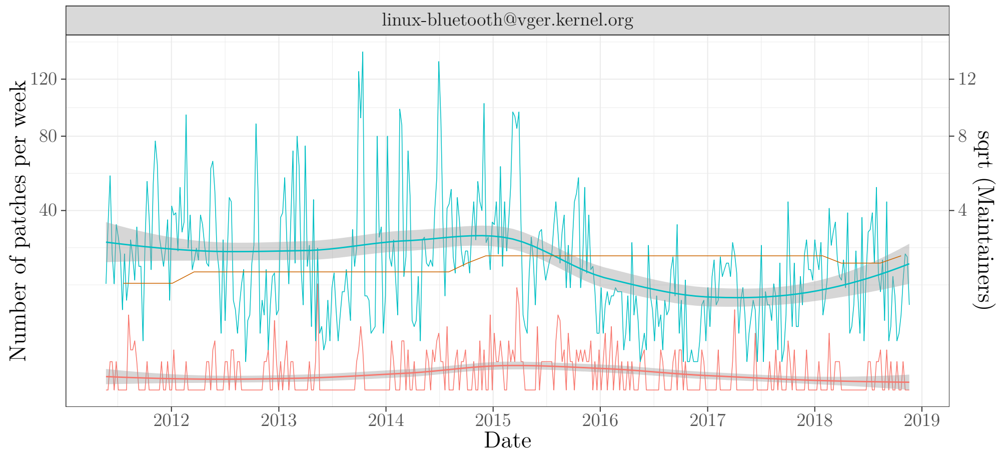

Figure 1.20: Patches per Week on Bluetooth Mailing List with Maintainers over Time

Looking at the plot of the `bluetooth` mailing list the average number of ignored patches correlates with the number of total patches between 2011 and 2015. In 2015 the activity of the mailing list collapses, and the number of ignored patches is decreasing deferred as well. The additional and the quitting maintainers cannot be noticed in the ignored-patches plot. There are only two small peaks of ignored patches (∼ July of 2011 and ∼ Dec of 2014) that correlate with a new maintainer in the team. However, this is no consistent phenomenon as the peak is missing in October 2018 when Sean Wang became a maintainer.

1.5.4.3 Similar Frequency of Ignored Patches

The `arm`, `fsdevel`, and the `media` mailing lists have a similar frequency of ignored patches, each mailing list ignored about 5 patches per week. The interesting thing is, that all three mailing lists have a different frequency of total patches sent (`arm`: 500 patches per week; `fsdevel`: 100 patches per week; `media`: 130 patches per week). This suggests that the `arm` mailing list's process can cope with more patches.

1.5.4.4 Ratio of Ignored to Total Patches of Mailing Lists

Other mailing lists' frequency of ignored patches is less than the abovementioned five patches per week; with one exception the LKML itself. All mailing lists with a lower frequency of ignored patches have a lower frequency of total patches than the `arm`-mailing list as well.

The ratio of ignored patches is usually below 5% in a mailing list. Due to the low frequency of all patches sent on most of the mailing lists even small patch series that are ignored end up as a high ratio of ignored patches (see figure 1.22). There are some mailing lists, like `cifs`, that receive down to zero patches per week. In figure 1.21 one can see the plots of the `block` and `mips` mailing list. The `mips` list is an example of a low traffic mailing list with about 20 patches per week. The `block` list is an example for a new mailing list, it was introduced in 2015 but the traffic grew to over 50 patches per week.

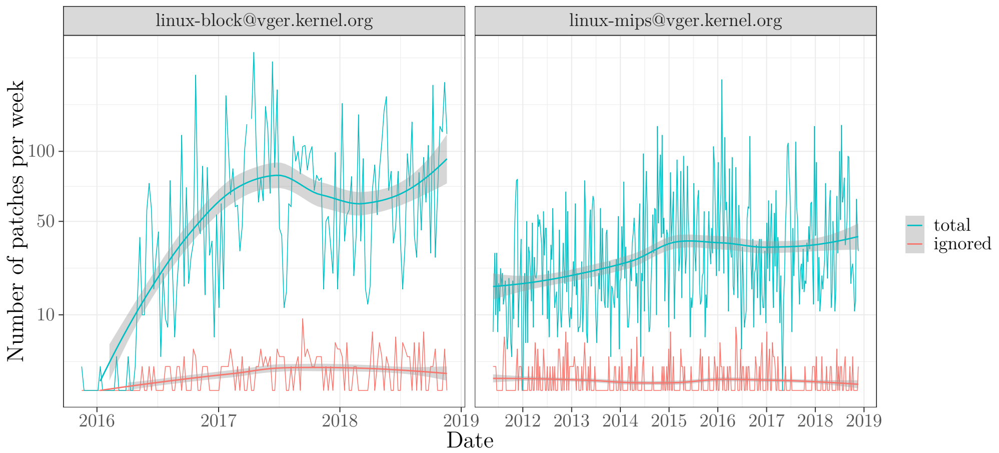

Figure 1.21: Number of Total and Ignored Patches per Week of `block` and `mips` Mailing Lists

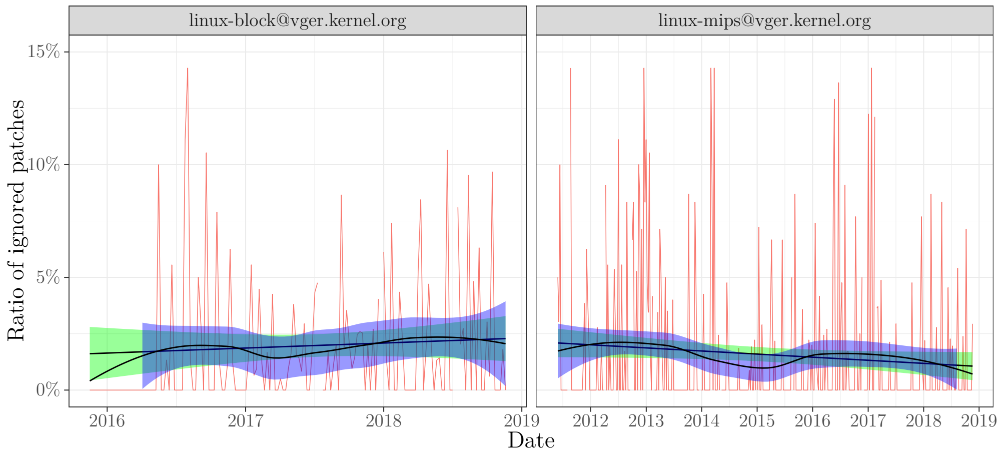

Figure 1.22: Ratio of Ignored to Total Patches of 'block' and 'mips' Mailing Lists

1.5.4.5 Summary

In this section, we learned about ignored patches from the mailing list's point of view to answer RQ4a (What is the information about subsystems/mailing lists that can be derived from the ignored patches data?):

We see that all mailing lists have ignored patches. The baseline of ignored patches that can be found in all mailing lists strengthens our the insights of the independent distribution of patches. In addition, we see, that some mailing lists have a lower ratio of ignored patches than others. Further, the analyses show that the number of maintainers has no significant impact on the ignored patch ratio.

1.5.5 Analyses of Subsystems

In the analyzed time frame, there were 2088 subsystems in the Linux kernel. 868 of these subsystems received over 100 contributions. The smaller subsystems are ignored in this analysis, as there is too few data for statistical analysis. We will select interesting clusters in the following section and describe each cluster.

1.5.5.1 Subsystems by Size

All analyzed subsystems have an average ignored-ratio of 2.2%. We split the subsystems into two groups; one with more than 1960 (five patches per week) contributions one with less. The big-groups' average ignored-ratio is 1.7%, the small groups' average ignored-ratio is 2.2%. The five number summaries can be found in table 1.33

quantile	not-ignored	ignored
0.00	0,1%	0,0%
0.25	1,0%	0,6%
0.50	1,5%	1,5%
0.75	1,9%	3,0%
1.00	7,3%	13,8%

Table 1.33: Subsystems by Size - Five Number Summaries [155]

In the big-group, the subsystem with the lowest ignored-ratio is `KERNEL VIRTUAL MACHINE FOR ARM64 (KVM/arm64)` with 4 ignored patches out of 2746 total patches. The low ratio can be explained by the low ignored-ratio of the `arm` mailing list which can be explained by the high number and the high stake of contributors and maintainers which are paid by a company for the work on the kernel[25].

In the big-group, the subsystem with the highest ignored-ratio is `SOUND` with 1196 ignored patches or 16480 total patches. The high ratio of ignored patches can be explained by the high ratio of volunteers working on the `linux-media` mailing list[26].

Similar to the insights from the list-analyses, we can see that big subsystems are relatively ignoring fewer patches. However, there are small subsystems equally performing.

1.5.5.2 Unmaintained and Removed Subsystems

As the ignoring of a patch is the result of a not responding maintainer, we identified orphaned subsystems. There was the hypothesis that the ratio of ignored patches would significantly increase after a subsystem becomes orphaned.

Most of the orphaned subsystems are too small to be able to detect anything. For example, the patch frequency of `PMC SIERRA MaxRAID DRIVER` is so low (one patch every three weeks) that one cannot detect anything due to outliers.

Some subsystems, like `OMAP USB SUPPORT` have a higher patch frequency. Even after being orphaned the subsystem received patches that were accepted. In figure 1.23, one can see the activity of the subsystem. It became orphaned in 2016. Even when the subsystem is orphaned, there is some activity [138]. This

[25]There are about 130 maintainers and reviewers (effective *v*4.9) associated with the `arm` mailing list; only 20% of them are not obviously associated with a company (Not using a company mail-address).

[26]There are about 60 maintainers and reviewers (effective *v*4.9) associated with the `media` mailing list; 40% of them are not obviously associated with a company (Not using a company mail-address).

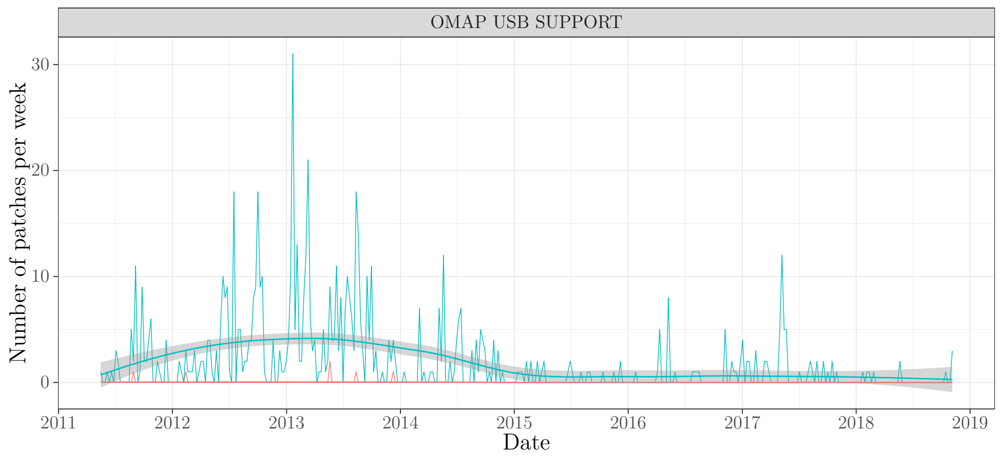

Figure 1.23: The orphaned `OMAP USB SUPPORT` subsystem

shows the resilience of the kernel community structure that even in an orphaned subsystem is activity.

1.5.5.3 Summary

In this section, we learned about ignored patches from the subsystem's point of view to answer RQ4b (What is the information about subsystems/mailing lists that can be derived from the ignored patches data?):

The analyses show us that small subsystems usually have a higher ratio of ignored patches than big subsystems. This can be explained by a better-trained process of the maintainers of the big subsystems. Moreover, some subsystems have maintainers who are paid for the work. We notice that we cannot distinguish the subsystem's status using the ignored patches metric as neither the relative nor the absolute number changes in a detectable pattern.

The lesson learned is that the analysis of most subsystems is not statistically sound as the subsystems are too small (by patch-frequency) to cope with statistic outliers.

1.6 Validation

In this section, we discuss the validation of the previous results. Like the research process, we will split the validation into two parts. First, we recapitulate the validation of the measurements. Second, we look at the validation of the analyses. Afterwards, we discuss further validation.

1.6.1 Validation of Measurements

To validate the measurements, we conducted two spot tests.

First, we conducted a small spot test to gain an understanding of the quality of all measurements. This spot test is based on a manual analysis of 25 entities.

Second, we conducted a more sophisticated spot test to be able to precisely examine the quality of the ignored patch measurement. This spot test is based on a manual analysis of 500 entities.

The manually checked entities were selected at random. We used the service from `random.org` to get a list of 500 random integers[27]. The patches at the line number equal to the integers were extracted and manually checked. The result of the spot tests can be found in the section 1.4.3 in the according subsection of each measurement.

We are not validating the sections with the reference-implementation because we are not measuring but processing data. There is no need of a validation because the excellence of the data is depending on the previously validated data and the precisely described process.

1.6.2 Further Validation

The quality of research conducted in this thesis was discussed in the section 1.3. We discussed quality criterions (Construct Validity, Internal Validity, External Validity, Reliability), and Yin's [170] Principles of Data Collection (Multiple Sources of Evidence, Case Study Database, Chain of Evidence, and Characteristics of Electronic Data Sources). Section 1.7 discusses the weaknesses of sources of evidence and the weaknesses of the research methodology.

For further validation of the research, we suggest the following: The thesis focuses on third-degree data sources [170, 93]. For a better balance of the methods the use of first-degree and second-degree sources have to be intensified in further research. Participant Observation, and Interviews [93] might be the best additional sources for this research.

[27]To reproduce the selection, the integers can be accessed here: `https://www.random.org/integers/?num=500&min=1&max=792464&col=1&base=10&format=plain&rnd=id.ignored-patches`

1.7 Limitations

1.7.1 Limitations of the Research Method

1.7.1.1 Generalizability

The case study design can be applied to all cases that fulfill the requirements (see section1.3.1.1) and go with the theoretical concepts (see section 1.2.1). However, the results of the analysis can change when analyzing other cases. This is possible due to a different CoC, tooling, or community (-structure).

According to Stol [145] field studies are not statistically generalizable.

1.7.1.2 Reliability

We used openly available data-sources, described the process from the data-source to the interpretation of the analyses, and we provided all source code used. Thus, all mistakes in the research design and the research itself can easily be detected.

There are two threads to the reliability. First, Namsor as a data-source cannot be validated because we have no information of the gender of the developers. Second, due to the missing archives of several mailing lists, we cannot analyze the whole software development process.

1.7.1.3 Construct Validity

By splitting the research process into two parts, we omitted the risk of a biased data-selection. There is the threat of a bad selection of analyses. We coped with this by a peer review of the analyses.

1.7.1.4 Internal Validity

As we conduct exploratory research, we have no underlying assumptions and can omit this test with the approval of Yin [170].

1.7.2 Weaknesses of the Data Sources

According to Lethbridge [93], all used sources are qualified to be used for exploratory research. We will discuss the source's weaknesses in the following subsections.

Lethbridge [93] identifies the sheer amount of data as a weakness of empirical research. This is no disadvantage of our research as we use computer-aided tools for the analysis. This enables us to analyze much data with care. Furthermore, Lethbridge noted that while using third degree sources, one has to keep the development system and process changes in mind. Fortunately, the Linux kernel development system and the process did not change in the analyzed time frame.

Analysis of Electronic Databases of Work Performed According to Lethbridge, the weakness of this source is the uncontrollable quality and quantity of the data. Some mailing lists archives (e.g. `arm`) are not correct (some headers are missing). When checking the LKML's archive, we found no missing headers. As we mainly retrieve our data from the LKML, the analyses can cope with this. As all mails are available in the archives, the weakness of potentially bad quantity will not affect the research.

Analysis of Tool Logs Lethbridge claims that the available data is often dispersed across multiple tools. In the Linux kernel development several tools are used. Some subsystems use specific mailing lists some use other source code management tools, and some have special bug-tracker. Thus all data is available on the mailing lists and Linus' git repository.

Documentation Analysis The required time consumption is no weakness in the results. The claimed inconsistency of the documentation is valid. Though we are analyzing mostly metadata created by the tools we are only little affected. As a result, claimed inconsistencies did not affect our research.

Yin[170] suggests other categories similar to Lethbridge's. Documentation and Archival Records are combined as they share most of the weaknesses.

Retrievability As all sources used (except Namsor) are publicly available at the specified URLs, all data can be retrieved.

Biased Selectivity As the sources are complete in the given period, and we simply select all data, there cannot be any selectivity bias. Some mailing lists have no complete records for the given period to prevent any bias, we do not include these mailing lists in our data collection process.

Reporting Bias As (almost) all commits in the Linux kernel git repository can be linked to mails from the mailing lists (almost) all patches were reported. An analysis by Ramsauer et al. [128] identified 24 off-list patches[28] (about 1800 total commits, 1.3%) in the Linux kernel development (between `v5.1-rc1` and `v5.1`). Thus, our data collection technique can collect (nearly) all patches.

Access All data (analyzed) related to the development and the developed source code is openly available [48].

Accessibility Due to Privacy Reasons (Archival Records only): All data (analyzed) related to the development and the developed source code is openly available.

One can see, that our research copes well with all weaknesses of the source of evidence.

[28]An off-list patch is a patch that has been included in Linus' git repository and has never been sent to any public mailing list [128].

1.7.3 Intended Limitations

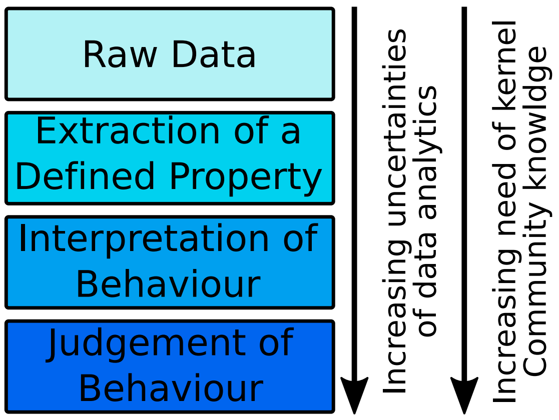
Raw Data
Extraction of a Defined Property
Interpretation of Behaviour
Judgement of Behaviour
Increasing uncertainties of data analytics
Increasing need of kernel Community knowldge

Figure 1.24: Data Aggregation

In this thesis, we want to compile the dataset, show how to alter the dataset to simplify analyses of authors, subsystems, lists, and conduct analyses of the data. Our analysis is supposed to show the actual condition objectively, not to judge or blame. Based on the insights, the community can change at their own discretion. Figure 1.24 shows the data aggregation process which is discussed below.

Raw Data We use the raw data publicly available (see section 1.4.1).

Extraction of a Defined Property To extract the data we use PaStA with our extension (see section 1.4.2) and the Namsor-service (see section 1.4.1.3).

Interpretation of Behavior In the analysis section, we interpret the data. The interpretation is based on the previously extracted data in combination with some knowledge of the community.

Judgment of Behavior As the validation of the interpretation, and in-depth community knowledge is missing, we will not judge the behavior. We will *not* judge about any contributor, the community, or the process.

According to Tukey's [155] analogy, we did the work of the police and the detectives, as we found evidence. The explanation and interpretation have to be done separately. Our interpretations have to be used with care. They are hypothesis and there is no underlying explanatory case study.

1.7.4 Further Work

We want to suggest further work in this section, that hat not be done in this thesis, because it is not in the scope of the thesis

1.7.4.1 Dataset

We did the first reusable dataset of the patches sent in the Linux kernel development. There are some extensions we were not able to implement:

Categorization of Patches We suggest creating a framework to categorize patches. Some categories might be: Bug-Fix (major-fix, remove-compiler-warning, typo-fix, security-fix), New-Feature, Performance-Improvement, Refactoring. Another category worth to detect is if a patch was created with tool-assistance (like coccinel). The categorization of the patches has to be automatable. We did not develop such a framework as this was not in the scope of this thesis.

Patchwork-ID of Patches Patchwork is the web-frontend of the LKML. In Patchwork, one can see patches with the discussion and the tool processes the discussion and extracts statuses like who acked a patch. Adding the Patchwork-ID of a patch would simplify the manual work with the dataset. We were not able to implement this feature as it is not possible to search a patch by `message-id`. This feature is not yet implemented.

Upstream Commit Hashes A column with the detected upstream hashes should be added to the dataset. As upstream patches cannot be ignored, by definition, this was not in the scope of the thesis.

Dataset Based on all Mailing Lists Our dataset is only based on a selection of mailing lists. The selected mailing lists were chosen because these are the only mailing lists with complete (all patches from the first one are available) data. When there are more mailing lists with sufficient data (e.g. Ramsauers' mailing list archive [121]), the dataset should be extended.

Record of all Mails The dataset currently only contains information about patches. In a future version of the dataset all patches should be recorded. We did not record only patches because it is easier to measure patches. A sophisticated measurement of a non-patch mail requires NLP to imply any information of the patch.

Create Database To further simplify working with the patch data, we suggest to migrate the dataset from a `.csv`-file to a database. This avoids some serialization issues, we faced. Additionally, this simplifies partial loading of the data. Currently, one has to load the file and drop the unnecessary data.

1.7.4.2 Ignored-Patch Analysis

Use Additional Measurements When the dataset is extended, the analyses should be repeated. The additional information like the categorization or NLP-techniques might be of merit.

Analysis per Maintainer We did the analyses from the developer's point-of-view. The analyses can be conducted from the maintainer's point-of-view.

The problem of creating the dataset per maintainer is that maintainers have to be tracked over time. Some maintainers changed the mail-address while being a maintainer. Additionally, one maintainer is not only working for one subsystem but many. This requires a sophisticated tracing of maintainers and knowledge about former employers. Both exceeded the manageable effort.

Predict Ignored Patches The insights gained in this thesis, and in the further work can be used to implement a predictor if a patch will be ignored. The feature might be implemented in the `check_patch.pl`-script to be neatly embedded in the process.

Notifier The ability to detect ignored patches can be used to implement a reminder. The reminder watches the mailing lists to identify ignored patches and reminds developers to resend the patch. Before launching this tool, it needs to be discussed with the community, and developers have to be able to opt-in/-out.

Safety Certification The insights gained in this thesis could be extended and used in the safety-certification process for the Linux kernel. As abovementioned, ignoring patches is a violation of the review process. Patches that have been identified as ignored have to be further analyzed if they contradict the requirements of the certification (e.g. generic certification: IEC 61508 [77]; or certification for automotive: ISO 26262 [79]).

Analysis of other Projects In the thesis, we state lots of numbers. As this is the first research of the ignored patch phenomenon, there is a lack of comparables. We suggest conducting further analysis on other projects like BSD (another operating system), GNU-projects (comparable tooling), or Angular (different domain, different tooling). The insights gained in these projects would help to learn more about the effect of tooling and other community types.

Explanatory Study This thesis represents the detective's work [155] of analyzing the ignored-patch phenomenon. In a second step, an explanatory study is required to back the new insight gained in this study.

1.8 Conclusion

When we started to conduct our research about ignored patches in the Linux kernel development process, we noticed the lack of data required for our research. Thus, we started to create a comprehensive, reusable dataset. In an iterative process, we selected measurements. After the creation of the dataset, we conducted a two-step validation. First, we did a small spot test of trivial measures. Second, we did a large spot test of the non-trivial measures. Based on the created dataset we simplified research of the Linux kernel development process.

When the dataset was created, we were able to conduct our analyses and tackle the RQs.

Quantification of the Ignored Patches Phenomenon (RQ1) In the measured time frame (v3.0 to v4.20), there were 18k patches of 792k patches ignored; about $2,3\%$. Over the time frame, there was a steady trend of decreasing ratio of ignored to total patches. The frequency of ignored patches is constant and the frequency of all patches is increasing.

Characteristics of Patches (RQ2) We found no characteristic of ignored patches that is statistically sound. However, some trends crystallized (the more effort is in a patch, the less likely the patch is ignored). In the author's analysis, we detect two types of patches that are often ignored. As we could not automatically identify these types of patches, there is no analysis of the types of patches.

Authors (RQ3) Some authors are ignored more often due to the type of patches they sent. Further, we found indications that discrimination takes place by ignoring patches.

Mailing Lists (RQ4a) The evaluations show that ignored patches are equally distributed, independent of specific characteristics of the mailing lists. Besides this, we detected that larger mailing lists have ignored fewer patches; maybe due to better workflows.

Subsystems (RQ4b) The analyses of the subsystems showed that a subsystem is a too-small unit of analysis to make statistically backed statements. However, we were able to detect the trend that large subsystems (by patch-frequency) are ignoring fewer patches than smaller subsystems. This insight is consistent with the insight from the mailing list analysis.

In the analyses, we found out that the ignored patches can be grouped into three clusters.

1. Automatically created patches
2. Minor contributions (`check_patch.pl` issues, typos)
3. Others

If a patch is automatically created or a minor contribution, they are not consistently ignored, but they are more likely to be ignored. High-traffic mailing lists like `lkml`, `arm`, or `netdev` ignore a smaller part of the received patches. We assume the high-traffic mailing lists have better processes to answer all patches.

The made hypotheses still have to be validated in an explanatory study. Tukey [155] says:

> Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone – as the first step

The conducted research is the first analysis of ignored patches. We found two categories of patches that are more likely ignored and indicators for discrimination; there were no other statistical abnormalities. In addition to the analyses, we created a dataset that simplifies further research of the Linux kernel development process. A one-sentence-summary of the insights of this thesis is: “Only submit patches you worked hard on!”

2 Elaboration

2.1 Changes to the Thesis Goals

We started on this thesis, to work towards the certification of Linux for automotive. During the research, we noticed that we are not able to extract the data we wished for (e.g. identify unmaintained subsystems). During the literature review for patch evolution, we identified the following types of patches:

- accepted patches
- rejected or in-discussion patches
- instant accepted patches
- ignored or pending patches

When exploring the literature for insights about ignored patches, we noticed that there is no research of this phenomenon yet. As a result, we decided (as agreed with the supervisor (Capraro)) to change the focus of the research from patch-evolution to the specific types of patches-evolution, ignored patches. Therefore, this thesis analyses ignored patches in the Linux kernel development.

2.2 Decisions

2.2.1 PaStA - Alternatives

For our research, we need a tool with two features. First, the tool has to mine the mailing-lists. Second, the tool needs to know if a patch is upstream. There are tools to analyze git repositories (e.g. Coming[1] and PyDriller[2]). There are tools to work with mailing lists mail2git[3]). However, we needed a tool capable of combining both tasks and with the ability to link those data sources. As there is no alternative tool to Pasta the selection of the tooling for the data collection was simple.

[1]`https://github.com/SpoonLabs/coming`
[2]`https://github.com/ishepard/pydriller`
[3]`https://github.com/coyotebush/mail2git`

2.2.2 χ^2-Test

We want to check if two variables are depending. The variables contain categorial data. There are two established tests for this situation:

- χ^2-test
- Fisher's exact test

Since, we have a high number of observations, we chose the χ^2-test.

2.3 Beside Work

Besides the work described above, I work(ed) on some extra thesis related tasks that are not part of the research itself. In this section, the tasks are described.

2.3.1 ELCE/OSSE and LPC

Since July 2019 I'm working on the thesis. In September of 2019, we presented the preliminary results of the thesis in Lisbon on the Linux Plumbers Conference [127]. Because, there was too little time between the beginning of the thesis and the conference, I was not able to present the results myself. Thus, peers presented the results. Nevertheless, I partly drafted the slide deck that was presented at the conference.

Later at the end of October 2019, there was the Embedded Linux Conference Europe / Open Source Summit Europe in Lyon, France. As there was enough time for the organization, I was able to participate in the conference. Additionally, together with Ralf Ramsauer, I was presenting the results of the research [55].

2.3.2 Maintenance of Maintainers

During the data-collection, we noticed some issues in the maintainer-file. We detected five categories of issues:

- Duplicated file-descriptors
- Invalid file-descriptors (due to moved or removed files)
- Missing file-/keyword-descriptors
- Missing status descriptors
- Formatting issues

We are currently implementing a script to detect these errors and automatically create a patch resolving the issues[4]. As the script still is in an early development stage. It is not published yet.

[4]We are aware that automatically created patches are more likely to be ignored.

2.3.3 Continuously Updated Dataset

In this thesis, we proposed a dataset of all patches from the Linux kernel. We want to publish the dataset and continuously add new patches. As the process is quite bulky at the moment, further work has to be done to create a continuous process to extract the data from the mailing-lists and the repository.

2.3.4 Kernel Maintainer Handbook

Before we conducted the analyses of the mailing-lists and subsystems, we planned to work on the proposed [165, 44] Linux kernel maintainer handbook. We planned to contribute insights if a patch for any subsystem is likely to be ignored and propose additional help on how this can be avoided (e.g. split patch into smaller subsets and resend each). However, during the analyses we noticed two things. First, the subsystems itself is too small to be able to make statistically backed statements. Second, the phenomenon of ignored patches is not related to subsystems but more a kernel-wide phenomenon.

2.4 Acknowledgments

I want to thank Dr. Lukas Bulwahn for being a discussion partner and for guidance during the process. Further, I want to thank Ralf Ramsauer for the work and the help with PaStA.

Appendix A Python Helper Functions

A.1 `add_or_create`

To create the data set, we used a function called `add_or_create` which id defined below:

```
def add_or_create(d, k, v=1):
    """
    Add value to dict if key exists; otherwise create \
        key with given value.

    :param d: Dictionary which should be modified.
    :param k: Key of the dictionary's entry which should \
        be modified.
    :param v: Value that should be added or set.
    """
    if k in d:
        d[k] += v
    else:
        d[k] = v
```

A.2 `get_most_current_maintainers`

The function `get_most_current_maintainers` returns the subsystem object of the latest occurrence. The function requires two external variables:

- `maintainers_cache` is a dictionary; this value is not required for the functionality but significantly enhances the performance
- `maintainers` is a dictionary with the kernel versions as keys and the related maintainers-objects as values

```
def get_most_current_maintainers(subsystem):
    """
    Search for newest entry of given subsystem in maintainers files

    :param d: Subsystems of interest
    :return: Queried subsystem-object
    """
    #is subsystem cached?
    if subsystem in maintainers_cache.keys():
        return maintainers_cache[subsystem]

    # sort maintainers files by ages (asc.)
```

```
tags = sorted(maintainers.keys(), reverse=True)

for tag in tags:
    try:
        maintainers_cache[subsystem] = \
            maintainers[tag].subsystems[subsystem]
        return maintainers_cache[subsystem]
    # continue searching for object if not found
    except KeyError:
        continue
raise KeyError('Subsystem ' + subsystem +  'not found')
```

Appendix B Example Output get_maintainers.pl

```
ORINOCO DRIVER
L: linux-wireless@vger.kernel.org
W: http://wireless.kernel.org/en/users/Drivers/orinoco
W: http://www.nongnu.org/orinoco/
S: Orphan
F: drivers/net/wireless/intersil/orinoco/

NETWORKING DRIVERS (WIRELESS)
M: Kalle Valo kvalo@codeaurora.org
L: linux-wireless@vger.kernel.org
Q: http://patchwork.kernel.org/project/linux-wireless/list/
T: git git://git.kernel.org/pub/scm/linux/kernel/git/\
        \kvalo/wireless-drivers.git
T: git git://git.kernel.org/pub/scm/linux/kernel/git/\
        kvalo/wireless-drivers-next.git
S: Maintained
F: Documentation/devicetree/bindings/net/wireless/
F: drivers/net/wireless/

NETWORKING DRIVERS
M: David S. Miller davem@davemloft.net
L: netdev@vger.kernel.org
W: http://www.linuxfoundation.org/en/Net
Q: http://patchwork.ozlabs.org/project/netdev/list/
T: git git://git.kernel.org/pub/scm/linux/kernel/git/\
        davem/net.git
T: git git://git.kernel.org/pub/scm/linux/kernel/git/\
        davem/net-next.git
S: Odd Fixes
F: Documentation/devicetree/bindings/net/
F: drivers/net/
F: include/linux/if\_*
F: include/linux/netdevice.h
F: include/linux/etherdevice.h
F: include/linux/fcdevice.h
F: include/linux/fddidevice.h
F: include/linux/hippidevice.h
F: include/linux/inetdevice.h
F: include/uapi/linux/if\_*
F: include/uapi/linux/netdevice.h
```

```
THE REST
M: Linus Torvalds torvalds@linux-foundation.org
L: linux-kernel@vger.kernel.org
Q: http://patchwork.kernel.org/project/LKML/list/
T: git git://git.kernel.org/pub/scm/linux/kernel/git/\
        torvalds/linux.git
S: Buried alive in reporters
F: *
F: */
```

Appendix C χ^2-Test Execution

The *Chi*^{2} test is conducted after the instructions by `crashkurs-statistik.de`.

C.1 Ignored First Submission

In this test (equations 2.1 to 2.7), we asses if the ignoring of the first contribution correlates with the abandonment of the project by the contributor. In the tables below, "ignoring the first contribution" is abbreviated as "ignored". "1" indicates the abandonment of the project by the contributor whereby "<1" indicates a continuous activity.

Contingency table	1	>1	sum
ignored	459	596	1055
not-ignored	3454	10467	13921
sum	3913	11063	**14976**

(2.1)

Expected values	1	>1	sum
ignored	256.66	779.34	1055
not-ignored	3637.34	10283.66	13921
sum	3913	11063	**14976**

(2.2)

Test values	1	>1
ignored	121.95	43.13
not-ignored	9.24	3.27

(2.3)

$$\chi^2 = 177,59 \tag{2.4}$$

$$Degrees\ of\ freedom : 1 \tag{2.5}$$

$$Significance\ level : 1\% \tag{2.6}$$

$$Critical\ value : 6,635 \tag{2.7}$$

As the χ^2 value is higher than the critical value, the variables `ignoring the first contribution` and `abandonment of the project by the contributor` are correlating.

C.2 Discrimination of Contributors with `.cn` Mail-Address

In this test (equations 2.8 to 2.14), we asses if the ignoring of contributions correlates with the `.cn`-tld in the mail address of the contributor.

Contingency table	.cn	not .cn	sum
ignored	90	17718	17808
not-ignored	1036	773516	774552
sum	1126	791234	**792360**

(2.8)

Expected values	.cn	not .cn	sum
ignored	25.31	17782.69	17808
not-ignored	1100.69	773451.31	774552
sum	1126	791234	**792360**

(2.9)

Test values	.cn	not .cn
ignored	165.38	0.23
not-ignored	3.80	0.01

(2.10)

$$\chi^2 = 169,43 \tag{2.11}$$

$$Degrees\ of\ freedom : 1 \tag{2.12}$$

$$Significance\ level : 1\% \tag{2.13}$$

$$Critical\ value : 6,635 \tag{2.14}$$

As the χ^2 value is higher than the critical value, the variables `ignored` and `.cn tld in mail address` are correlating.

C.3 Discrimination of Non-Male Contributors

In this test (equations 2.15 to 2.21), we asses if the ignoring of contributions correlates with the gender.

Contingency table	non-male	male	sum
ignored	903	16905	17808
not-ignored	26527	748025	774552
sum	27430	764930	**792360**

(2.15)

Expected values	non-male	male	sum
ignored	616.48	17191.52	17808
not-ignored	26813.52	747738.48	774552
sum	27430	764930	**792360**

(2.16)

Test values	non-male	male
ignored	133.16	4.78
not-ignored	3.06	0.11

(2.17)

$$\chi^2 = 141,11 \tag{2.18}$$

$$Degrees\ of\ freedom : 1 \tag{2.19}$$

$$Significance\ level : 1\% \tag{2.20}$$

$$Critical\ value : 6,635 \tag{2.21}$$

As the χ^2 value is higher than the critical value, the variables `ignored` and `gender` are correlating.

Appendix D Pearson-Correlation-Coefficient of Size – Source Code

```
import pandas as pd
all_patches = pd.read_csv('Data/1913/characteristics.csv')
not_ignored_patches = all_patches[~all_patches['ignored']]
ignored_patches = all_patches[all_patches['ignored']]

tpg = not_ignored_patches.groupby(['#locs']) \
    .count()['id'].sort_index()
ipg = ignored_patches.groupby(['#locs']) \
    .count()['id'].sort_index()

corr = list()

for j in range(0, tpg.index.max()):
    try:
        t = tpg.loc[j]
    except:
        t = 0
    try:
        i = ipg.loc[j]
    except:
        i = 0
    corr.append({'idx': j, 'tot':t, 'ign':i})

pd.DataFrame(corr).set_index(['idx']).corr('pearson')
```

References

[1] *[0001/1285] Replace numeric parameter like 0444 with macro*. 2016. URL: `https://lore.kernel.org/patchwork/patch/704048/` (visited on 13/11/2019).

[2] *[0029/1285] Replace numeric parameter like 0444 with macro*. 2016. URL: `https://lore.kernel.org/patchwork/patch/702697/#889616` (visited on 19/11/2019).

[3] *[1/5] locking/percpu-rwsem, lockdep: Make percpu-rwsem use its own lockdep_map*. 2019. URL: `https://lore.kernel.org/patchwork/patch/1152988/` (visited on 13/11/2019).

[4] *2. How the development process works*. 2019. URL: `https://www.kernel.org/doc/html/latest/process/2.Process.html` (visited on 14/11/2019).

[5] *2.5. Staging trees*. 2019. URL: `https://www.kernel.org/doc/html/latest/process/2.Process.html#staging-trees` (visited on 14/11/2019).

[6] *A guide to the Kernel Development Process*. 2019. URL: `https://www.kernel.org/doc/html/latest/process/development-process.html` (visited on 23/09/2019).

[7] Mark Aberdour. 'Achieving quality in open-source software'. In: *IEEE software* 24.1 (2007), pp. 58–64.

[8] *ABIT UGURU 1,2 HARDWARE MONITOR DRIVER – MAINTAINERS – v5.3*. 2019. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n240` (visited on 23/09/2019).

[9] *About Linux Kernel*. 2017. URL: `https://www.kernel.org/linux.html` (visited on 06/11/2019).

[10] Mohammad AlMarzouq et al. 'Open source: Concepts, benefits, and challenges'. In: *Communications of the Association for Information Systems* 16.1 (2005), p. 37.

[11] *AMAZON ANNAPURNA LABS THERMAL MMIO DRIVER – MAINTAINERS – v5.3*. 2019. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n753` (visited on 23/09/2019).

[12] *AMD DISPLAY CORE – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n785 (visited on 23/09/2019).

[13] *Android (operating system).* 2019. URL: https://en.wikipedia.org/wiki/Android_(operating_system) (visited on 23/09/2019).

[14] *ARM/SAMSUNG MOBILE MACHINE SUPPORT – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n2229 (visited on 23/09/2019).

[15] *ASPEED VIDEO ENGINE DRIVER – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n2641 (visited on 23/09/2019).

[16] Vlastimil Babka. *[2/3] mm/filemap: initiate readahead even if IOCB_NOWAIT is set for the I/O.* 2019. URL: https://lore.kernel.org/patchwork/patch/1036978/ (visited on 27/11/2019).

[17] *BCACHE (BLOCK LAYER CACHE) – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n2938 (visited on 23/09/2019).

[18] Stefan Beiersmann. *Suse entwickelt Linux-Kernel für Microsoft Azure.* 2018. URL: https://www.zdnet.de/88340115/suse-entwickelt-linux-kernel-fuer-microsoft-azure/ (visited on 23/09/2019).

[19] Angela Bohn et al. 'Content-based social network analysis of mailing lists'. In: *The R Journal* 3.1 (2011), pp. 11–18.

[20] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[21] Ben Bowers. *Infinity I-Kitchen sports Linux-based touch screen computer, kitchen sink still not included.* 2010. URL: https://www.engadget.com/2010/11/30/infinity-i-kitchen-sports-linux-based-touch-screen-computer-kit/ (visited on 23/09/2019).

[22] David Bretthauer. 'Open source software: A history'. In: (2001).

[23] Maximilian Capraro, Michael Dorner and Dirk Riehle. 'The patch-flow method for measuring inner source collaboration'. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR).* IEEE. 2018, pp. 515–525.

[24] Maximilian Capraro and Dirk Riehle. 'Inner source definition, benefits, and challenges'. In: *ACM Computing Surveys (CSUR)* 49.4 (2017), p. 67.

[25] Elian Carsenat. 'Inferring gender from names in any region, language, or alphabet'. In: (Nov. 2019). DOI: 10.13140/RG.2.2.11516.90247.

[26] *Census.gov.* 2019. URL: https://www.census.gov/en.html (visited on 24/10/2019).

[27] Mauro Carvalho Chehab. *[04/12] media: v4l uAPI docs: adjust some tables for PDF output.* 2017. URL: https://lore.kernel.org/patchwork/patch/827835/ (visited on 27/11/2019).

[28] Mauro Carvalho Chehab. *[2/3,media] rename most media/video usb drivers to media/usb.* 2012. URL: https://patchwork.kernel.org/patch/1317851/ (visited on 13/12/2019).

[29] Mauro Carvalho Chehab. *doc_rst: rename the media Sphinx suff to Documentation/media.* 2016. URL: https://patchwork.kernel.org/patch/9221097/ (visited on 13/12/2019).

[30] Mauro Carvalho Chehab. *Linux Kernel patch submission checklist.* 2019. URL: https://www.kernel.org/doc/html/latest/process/submit-checklist.html (visited on 14/11/2019).

[31] Daming D Chen et al. 'Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.' In: *NDSS.* 2016, pp. 1–16.

[32] *CLEANCACHE API – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n4010 (visited on 27/11/2019).

[33] *Code of conduct.* 2019. URL: https://en.wikipedia.org/wiki/Code_of_conduct (visited on 23/09/2019).

[34] *Combined diff format.* 2019. URL: https://git-scm.com/docs/git-diff#_combined_diff_format (visited on 18/12/2019).

[35] *Common German Names for Boys and Girls.* 2019. URL: http://german.about.com/library/blname_topDE.htm (visited on 24/10/2019).

[36] Kees Cook. *Remove CONFIG_EXPERIMENTAL.* 2012. URL: https://lore.kernel.org/patchwork/patch/322802/ (visited on 13/12/2019).

[37] Jonathan Corbet. *Development statistics for the 5.0 kernel.* 2019. URL: https://lwn.net/Articles/780271/ (visited on 05/11/2019).

[38] Jonathan Corbet. *How 4.4's patches got to the mainline.* 2016. URL: https://lwn.net/Articles/670209/ (visited on 13/11/2019).

[39] Jonathan Corbet. *How patches get into the mainline.* 2009. URL: https://lwn.net/Articles/318699/ (visited on 13/11/2019).

[40] Jonathan Corbet. *Linux Kernel – What this document is about?* 2008. URL: https://www.kernel.org/doc/html/latest/process/1.Intro.html#what-this-document-is-about (visited on 23/09/2019).

[41] Jonathan Corbet. *Patch flow into the mainline for 4.14.* 2017. URL: https://lwn.net/Articles/737093/ (visited on 13/11/2019).

[42] Jonathan Corbet. *Some 4.4 development statistics.* 2015. URL: https://lwn.net/Articles/668870/ (visited on 05/11/2019).

[43] Jonathan Corbet. *Statistics from the 5.2 kernel — and before.* 2019. URL: https://lwn.net/Articles/791606/ (visited on 05/11/2019).

[44] Jonathan Corbet. *Toward a kernel maintainer's guide.* 2018. URL: https://lwn.net/Articles/772882/ (visited on 16/12/2019).

[45] Jonathan Corbet and G Kroah-Hartman. '„2017 Linux Kernel Development Report “'. In: *A Publication of The Linux Foundation* (2017).

[46] Kevin Crowston and James Howison. 'The social structure of open source software development teams'. In: (2003).

[47] Kevin Crowston and Barbar Scozzi. 'Bug fixing practices within free/libre open source software development teams'. In: *Journal of Database Management (JDM)* 19.2 (2008), pp. 1–30.

[48] Kevin Crowston et al. 'Free/Libre open-source software development: What we know and what we do not know'. In: *ACM Computing Surveys (CSUR)* 44.2 (2012), p. 7.

[49] Bálint Czobor. *[1/1] drivers: input: touchscreen: Initial support for ATMEL_MXTS touchscreen.* 2013. URL: https://lore.kernel.org/patchwork/patch/391357/ (visited on 01/12/2019).

[50] David Daney. *[PATCH 2/8] MIPS: OCTEON: Update register definitions.* 2012. URL: https://www.linux-mips.org/archives/linux-mips/2012-04/msg00197.html (visited on 02/12/2019).

[51] *Developer Certificate of Origin.* 2006. URL: https://developercertificate.org/ (visited on 17/10/2019).

[52] Martin Devera. *[PATCH 1/2] Add NXP LPC32XX SPI driver.* 2015. URL: https://lore.kernel.org/lkml/E1ZEEtz-0000pe-7R@luxik.cdi.cz/ (visited on 14/12/2019).

[53] *DEVICE DIRECT ACCESS (DAX) – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n4842 (visited on 23/09/2019).

[54] Jake Edge. *A new kernel tree: linux-staging.* 2008. URL: https://lwn.net/Articles/285594/ (visited on 14/11/2019).

[55] *ELCE - The List is our Process: An Analysis of the Kernel's Email-based Development Process.* 2019. URL: https://osseu19.sched.com/event/TPGw/ (visited on 19/12/2019).

[56] Markus Elfring. *ttusb_dec: Fine-tuning for some function implementations.* 2017. URL: https://lore.kernel.org/patchwork/project/lkml/list/?series=325049&state=*&archive=both (visited on 13/12/2019).

[57] Michael Ellerman. *[PATCH] powerpc/perf: Move perf core & PMU code into a subdirectory.* 2012. URL: https://lore.kernel.org/linuxppc-dev/1329790742-19308-1-git-send-email-michael@ellerman.id.au/ (visited on 18/12/2019).

[58] Wes Felter et al. 'An updated performance comparison of virtual machines and linux containers'. In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2015, pp. 171–172.

[59] Larry Finger. *[4/8] staging: r8822be: Add code for halmac sub-driver.* 2017. URL: http://patchwork.ozlabs.org/patch/802822/ (visited on 01/12/2019).

[60] Larry Finger. *[PATCH 9/9] staging: r8821ae: Remove remaining files from old version of driver.* 2014. URL: https://lore.kernel.org/linux-wireless/1393973635-24799-10-git-send-email-Larry.Finger@lwfinger.net/ (visited on 01/12/2019).

[61] Larry Finger. *staging: r8192ee: Remove staging driver*. 2014. URL: http://patchwork.ozlabs.org/patch/395008/ (visited on 01/12/2019).

[62] Brian Fitzgerald. 'The transformation of open source software'. In: *MIS quarterly* (2006), pp. 587–598.

[63] Javier Franch Gutiérrez et al. 'Managing risk in open source software adoption'. In: *ICSOFT 2013: Proceedings of the 8th International Joint Conference on Software Technologies*. 2013, pp. 258–264.

[64] Inc. Free Software Foundation. *GNU GENERAL PUBLIC LICENSE*. 1991. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/LICENSES/preferred/GPL-2.0 (visited on 11/11/2019).

[65] ggplot2. *Smoothed conditional means*. 2019. URL: https://ggplot2.tidyverse.org/reference/geom_smooth.html (visited on 13/12/2019).

[66] Thomas Gleixner. *time/timers: Type cleanups*. 2016. URL: https://lore.kernel.org/patchwork/patch/745879/ (visited on 13/12/2019).

[67] Richard Gooch. *I want to post a Great Idea (tm) to the list. What should I do?* 2009. URL: http://vger.kernel.org/lkml/#s3-17 (visited on 14/11/2019).

[68] Richard Gooch. *The linux-kernel mailing list FAQ*. 2009. URL: http://vger.kernel.org/lkml/ (visited on 23/09/2019).

[69] Georgios Gousios, Eirini Kalliamvakou and Diomidis Spinellis. 'Measuring developer contribution from software repository data'. In: *Proceedings of the 2008 international working conference on Mining software repositories*. ACM. 2008, pp. 129–132.

[70] Thomas C Greene. *Ballmer: 'Linux is a cancer'*. 2001. URL: https://www.theregister.co.uk/2001/06/02/ballmer_linux_is_a_cancer/ (visited on 23/09/2019).

[71] Sebastian Grüner. *Die Kernel-Maintainer skalieren nicht*. 2016. URL: https://www.golem.de/news/linux-und-containercon-2016-die-kernel-maintainer-skalieren-nicht-1610-123687.html (visited on 06/11/2019).

[72] Anja Guzzi et al. 'Communication in open source software development mailing lists'. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 277–286.

[73] Gerd Hoffmann. *[v2,1/6] virtio-gpu: add virtio_gpu_queue_ctrl_buffer_locked*. 2015. URL: https://lore.kernel.org/patchwork/patch/599728/ (visited on 13/12/2019).

[74] David Howells. *[RFC,00/29] Disintegrate and kill asm/system.h*. 2012. URL: https://lore.kernel.org/patchwork/cover/290648/ (visited on 13/12/2019).

[75] Frank Hübner. *1-TeraFLOPS-Linux-Server: Supercomputer bleibt noch länger auf der ISS*. 2019. URL: https://www.computerbase.de/2019-02/teraflops-linux-server-supercomputer-iss/ (visited on 23/09/2019).

[76] *Hyper-V CORE AND DRIVERS – MAINTAINERS – v5.3.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.3#n7439 (visited on 23/09/2019).

[77] *Functional safety of electrical/electronic/programmable electronic safety-related systems – Software requirements.* Standard. Geneva, CH: International Electrotechnical Comission, Mar. 2010.

[78] ScienceMetrics Inc. 'Analytical Support for Bibliometrics Indicators'. In: (2018). URL: www.science-metrix.com/sites/default/files/science-metrix/publications/science-metrix_bibliometric_indicators_womens_contribution_to_science_report.pdf.

[79] *Road vehicles – Functional safety – Management of functional safety.* Standard. Geneva, CH: International Organization for Standardization, Dec. 2018.

[80] Carlos Jensen, Scott King and Victor Kuechler. 'Joining free/open source software communities: An analysis of newbies' first interactions on project mailing lists'. In: *2011 44th Hawaii international conference on system sciences.* IEEE. 2011, pp. 1–10.

[81] Yujuan Jiang, Bram Adams and Daniel M German. 'Will my patch make it? and how fast?: Case study on the linux kernel'. In: *Proceedings of the 10th Working Conference on Mining Software Repositories.* IEEE Press. 2013, pp. 101–110.

[82] Lee Jones. *[01/47] ARM: sti: Add BCH (NAND Flash) Controller support for STiH41x (Orly) SoCs.* 2014. URL: https://lore.kernel.org/patchwork/patch/460997/ (visited on 13/12/2019).

[83] Danial Jordan. *[RFC,v2,3/8] mm: convert lru_lock from a spinlock_t to a rwlock_t.* 2018. URL: https://lore.kernel.org/patchwork/patch/984204/ (visited on 27/11/2019).

[84] Eirini Kalliamvakou et al. 'Measuring Developer Contribution From Software Repository Data.' In: *MCIS* 2009 (2009), 4th.

[85] KernelNewbies. *Kernel Newbies – Guide.* 2019. URL: https://kernelnewbies.org/Linux_Kernel_Newbies (visited on 23/09/2019).

[86] Colin King. *jffs2: fix indentation issue.* 2019. URL: https://lore.kernel.org/patchwork/patch/1153288/ (visited on 13/12/2019).

[87] Greg Kroah-Hartman. *[v2] USB: add SPDX identifiers to all remaining files in drivers/usb/.* 2017. URL: https://lore.kernel.org/patchwork/patch/847275/ (visited on 13/12/2019).

[88] Greg Kroah-Hartman. *USB: add SPDX identifiers to all files in drivers/usb/.* 2017. URL: https://lore.kernel.org/patchwork/patch/842653/ (visited on 13/12/2019).

[89] Herton Ronaldo Krzesinski. *Re: [3.5.y.z extended stable] Linux 3.5.7.3.* 2013. URL: https://lore.kernel.org/lkml/1358777909-26850-2-git-send-email-herton.krzesinski@canonical.com/ (visited on 13/12/2019).

[90] Victor Kuechler, Claire Gilbertson and Carlos Jensen. 'Gender differences in early free and open source software joining process'. In: *IFIP International Conference on Open Source Systems*. Springer. 2012, pp. 78–93.

[91] Tom Lendacky. *[v2,1/2] x86/cpu/AMD: Make LFENCE a serializing instruction*. 2018. URL: https://lore.kernel.org/patchwork/patch/871041/ (visited on 23/09/2019).

[92] Christophe Leroy. *[v4,01/21] powerpc/8xx: Declare SPRG2 as a SCRATCH register*. 2014. URL: https://lore.kernel.org/patchwork/patch/501456/ (visited on 13/12/2019).

[93] Timothy C Lethbridge, Susan Elliott Sim and Janice Singer. 'Studying software engineers: Data collection techniques for software field studies'. In: *Empirical software engineering* 10.3 (2005), pp. 311–341.

[94] Mauro Carvalho Chehab Linus Torvalds Jonathan Corbet. *Submitting patches: the essential guide to getting your code into the kernel*. 2019. URL: https://www.kernel.org/doc/html/latest/process/submitting-patches.html (visited on 14/11/2019).

[95] Mauro Carvalho Chehab Linus Torvalds Jonathan Corbet. *Using Reported-by:, Tested-by:, Reviewed-by:, Suggested-by: and Fixes:* 2019. URL: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#using-reported-by-tested-by-reviewed-by-suggested-by-and-fixes (visited on 14/11/2019).

[96] Mauro Carvalho Chehab Linus Torvalds Jonathan Corbet. *When to use Acked-by:, Cc:, and Co-developed-by:* 2019. URL: https://www.kernel.org/doc/html/latest/process/submitting-patches.html#when-to-use-acked-by-cc-and-co-developed-by (visited on 14/11/2019).

[97] *Linux Foundation Training Prepares the International Space Station for Linux Migration*. 2018. URL: https://training.linuxfoundation.org/solutions/corporate-solutions/success-stories/linux-foundation-training-prepares-the-international-space-station-for-linux-migration/ (visited on 23/09/2019).

[98] *Linux kernel*. 2019. URL: https://en.wikipedia.org/wiki/Linux_kernel (visited on 23/09/2019).

[99] *Linux Kernel Git Repository /Documentation/process*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/process (visited on 23/09/2019).

[100] *Linux Klausel*. 2019. URL: https://de.wikipedia.org/wiki/Linux-Klausel (visited on 23/09/2019).

[101] *Linux v0.98-rc1*. 1992. URL: http://www.oldlinux.org/Linux.old/Linux-0.98/sources/system/linux-0.98.1.tar.gz (visited on 23/09/2019).

[102] *List of maintainers and how to submit kernel changes*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS (visited on 10/11/2019).

[103] Robert Love et al. *Linux kernel development second edition.* Novell Press, 2005.
[104] LWN. *The LWN.net FAQ.* 2019. URL: `https://lwn.net/op/FAQ.lwn#general` (visited on 23/09/2019).
[105] Saeed Mahameed. *[for-next] net, IB/mlx5: Reorganize driver file layout.* 2017. URL: `http://patchwork.ozlabs.org/patch/714555/` (visited on 01/12/2019).
[106] *Majordomo lists at VGER.KERNEL.ORG.* 2019. URL: `http://vger.kernel.org/vger-lists.html#linux-kernel` (visited on 13/12/2019).
[107] Amy Harmon; John Markoff. *Internal Memo Shows Microsoft Executives' Concern Over Free Software.* 1998. URL: `https://archive.nytimes.com/www.nytimes.com/library/tech/98/11/biztech/articles/03memo.html` (visited on 23/09/2019).
[108] Catherine Marshall and Gretchen B Rossman. *Designing qualitative research.* Sage publications, 2014.
[109] Paul B Menage. 'Adding generic process containers to the linux kernel'. In: *Proceedings of the Linux symposium.* Vol. 2. Citeseer. 2007, pp. 45–57.
[110] Becka Morgan and Carlos Jensen. 'Lessons learned from teaching open source software development'. In: *IFIP International Conference on Open Source Systems.* Springer. 2014, pp. 133–142.
[111] Namsor. *Namsor.* 2019. URL: `https://www.namsor.com/` (visited on 23/09/2019).
[112] Stas Negara et al. 'Mining fine-grained code changes to detect unknown change patterns'. In: *Proceedings of the 36th International Conference on Software Engineering.* ACM. 2014, pp. 803–813.
[113] Deb Nicholson. *Solving the Linux kernel code reviewer shortage.* 2016. URL: `https://opensource.com/business/16/10/linux-kernel-review` (visited on 06/11/2019).
[114] Opensource.com. *What is open source?* 2016. URL: `https://opensource.com/resources/what-open-source` (visited on 13/11/2019).
[115] OSI. *The Open Source Definition (Annotated).* 2019. URL: `https://opensource.org/osd-annotated` (visited on 07/12/2019).
[116] Michael Quinn Patton. *Qualitative research and evaluation methods . Thousand Oakes.* 2002.
[117] Bill Pemberton. *[261/493] sound: remove use of ___devinit.* 2012. URL: `https://patchwork.kernel.org/patch/1774911/` (visited on 13/12/2019).
[118] Bruce Perens et al. 'The open source definition'. In: ().
[119] *Posting Patches.* 2019. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/5.Posting.rst#n211` (visited on 16/12/2019).
[120] Ralf Ramsauer. *FYI: Broken Headers in Public Inboxes.* 2019.
[121] Ralf Ramsauer. *linux-mailinglist-archives.* 2019. URL: `https://github.com/linux-mailinglist-archives` (visited on 07/12/2019).

[122] Ralf Ramsauer. *PaStA*. 2019. URL: `https://github.com/lfd/PaStA` (visited on 23/09/2019).

[123] Ralf Ramsauer. *PaStA Patch.py*. 2019. URL: `https://github.com/lfd/PaStA/blob/master/pypasta/Repository/Patch.py#L164` (visited on 16/11/2019).

[124] Ralf Ramsauer. *PaStA Resources Linux*. 2019. URL: `https://github.com/lfd/PaStA-resources/tree/master/linux` (visited on 23/09/2019).

[125] Ralf Ramsauer, Daniel Lohmann and Wolfgang Mauerer. 'Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks'. In: *Proceedings of the 12th International Symposium on Open Collaboration*. ACM. 2016, p. 4.

[126] Ralf Ramsauer, Daniel Lohmann and Wolfgang Mauerer. 'The list is the process: reliable pre-integration tracking of commits on mailing lists'. In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press. 2019, pp. 807–818.

[127] Ralf Ramsauer, Wolfgang Mauerer and Lukas Bulwahn. *The list is our process: An analysis of the kernel's email-based development process*. 2019. URL: `https://linuxplumbersconf.org/event/4/contributions/555/` (visited on).

[128] Ralf Ramsauer et al. *The List is Our Process! – An analysis of the kernel's email-based development process*. Linux Foundation, 2019.

[129] ES Raymond. *The Cathedral and The Bazaar, 1st edn. Tim O'Reilly*. 1999.

[130] Peter W. Resnick. *RFC 5322*. 2008. URL: `https://tools.ietf.org/html/rfc5322` (visited on 23/09/2019).

[131] Peter W. Resnick. *RFC 5322 – Identification Fields*. 2008. URL: `https://tools.ietf.org/html/rfc5322#section-3.6.4` (visited on 23/09/2019).

[132] Dirk Riehle. 'The economic motivation of open source software: Stakeholder perspectives'. In: *Computer* 40.4 (2007), pp. 25–32.

[133] Jeffrey Robertsa, Il-Horn Hann and Sandra Slaughter. 'Communication networks in an open source software project'. In: *IFIP International Conference on Open Source Systems*. Springer. 2006, pp. 297–306.

[134] Gregorio Robles et al. 'Women in free/libre/open source software: The situation in the 2010s'. In: *IFIP International Conference on Open Source Systems*. Springer. 2016, pp. 163–173.

[135] Per Runeson et al. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[136] Rusty Russel. *Unreliable Guide To Hacking The Linux Kernel*. 2005. URL: `https://www.kernel.org/doc/htmldocs/kernel-hacking/index.html` (visited on 23/09/2019).

[137] Daniel Russo. 'Benefits of open source software in defense environments'. In: *Proceedings of 4th International Conference in Software Engineering for Defence Applications*. Springer. 2016, pp. 123–131.

[138] Wolfram Sang. *[2/3] mfd: tps65010: move header file out of I2C realm.* 20. URL: https://lore.kernel.org/patchwork/patch/790278/ (visited on 13/12/2019).

[139] Wolfram Sang. *[PATCH 3/3] rtc: mxc: remove UIE signaling.* 2011. URL: https://lore.kernel.org/lkml/1304523088-17039-4-git-send-email-w.sang@pengutronix.de/ (visited on 18/12/2019).

[140] Wolfram Sang. *I Still Think We Have a Scaling Problem.* 2016. URL: https://events.static.linuxfound.org/sites/events/files/slides/LCE16_ScalingProblem_WSang.pdf.

[141] Wolfram Sang. *Maintainer's Diary: We have a scaling problem.* 2013. URL: https://events.static.linuxfound.org/sites/events/files/slides/ELCE2013_WolframSang_WeHaveAScalingProblem.pdf.

[142] Dirk Riehle Sebastian Duda Maximilian Capraro. *NYT Case Study Protocol.* 2018.

[143] Andy Shevchenko. *[v2,1/2] mfd: lpc_ich: Enable watchdog on Intel Apollo Lake PCH.* 2017. URL: https://lore.kernel.org/patchwork/patch/755460/ (visited on 27/11/2019).

[144] Igor Steinmacher et al. 'Why do newcomers abandon open source software projects?' In: *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE).* IEEE. 2013, pp. 25–32.

[145] Klaas-Jan Stol and Brian Fitzgerald. 'The ABC of software engineering research'. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27.3 (2018), p. 11.

[146] Rickard Strandqvist. *staging: vt6655: device_main: Removed variables that is never used.* 2015. URL: https://lore.kernel.org/patchwork/patch/539056/ (visited on 13/12/2019).

[147] *The Debian Free Software Guidelines.* 2004. URL: https://www.debian.org/social_contract.html#guidelines (visited on 13/11/2019).

[148] Top500. *Top500 – June 2019.* 2019. URL: https://www.top500.org/lists/2019/06/ (visited on 23/09/2019).

[149] Linus Torvalds. *Copying.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/COPYING (visited on 11/11/2019).

[150] Linus Torvalds. *Linux kernel source tree – Github.* 2019. URL: https://github.com/torvalds/linux/ (visited on 23/09/2019).

[151] Linus Torvalds. *Linux kernel source tree – Kernel.org.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/ (visited on 23/09/2019).

[152] Linus Torvalds. *README.* 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/README.rst (visited on 11/11/2019).

[153] Linus Torvalds. *What would you like see most in minix?* 1991. URL: `https://groups.google.com/d/msg/comp.os.minix/dlNtH7RRrGA/SwRavCzVE7gJ` (visited on 23/09/2019).

[154] Parastou Tourani, Bram Adams and Alexander Serebrenik. 'Code of conduct in open source projects'. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, pp. 24–33.

[155] John W Tukey. 'Exploratory Data Analysis Addision-Wesley'. In: *Reading, MA* 688 (1977).

[156] Tomi Valkeinen. *[1/3] video: move fbdev to drivers/video/fbdev*. 2014. URL: `https://lore.kernel.org/patchwork/patch/445518/` (visited on 13/12/2019).

[157] Steven J. Vaughan-Nichols. *Linux powers world's fastest stock exchange*. 2009. URL: `https://www.computerworld.com/article/2467498/linux-powers-world-s-fastest-stock-exchange.html` (visited on 23/09/2019).

[158] Steven J. Vaughan-Nichols. *London Stock Exchange moves to Linux*. 2010. URL: `https://www.computerworld.com/article/2469538/london-stock-exchange-moves-to-linux.html` (visited on 23/09/2019).

[159] Georg Von Krogh. 'Open-source software development'. In: *MIT Sloan Management Review* 44.3 (2003), pp. 14–18.

[160] Georg Von Krogh and Eric Von Hippel. 'The promise of research on open source software'. In: *Management science* 52.7 (2006), pp. 975–983.

[161] Vivian Wagner. *The Flying Penguin: Linux In-Flight Entertainment Systems*. 2008. URL: `https://www.linuxinsider.com/story/The-Flying-Penguin-Linux-In-Flight-Entertainment-Systems-65541.html` (visited on 23/09/2019).

[162] Shu Wang. *cifs: release auth_key.response for reconnect*. 2017. URL: `https://lore.kernel.org/patchwork/patch/829128/` (visited on 27/11/2019).

[163] Steve Weber. *The success of open source*. Harvard University Press, 2004.

[164] *Wikipedia:Mailing lists*. 2019. URL: `https://en.wikipedia.org/wiki/Wikipedia:Mailing_lists` (visited on 13/11/2019).

[165] Dan Williams. *Towards a Linux Kernel Maintainer Handbook*. 2018. URL: `https://www.linuxplumbersconf.org/event/2/contributions/59/attachments/141/174/LPC_2018_-_Maintainer_Handbook.pdf` (visited on 16/12/2019).

[166] Kipling D. Williams. 'Social Ostracism'. In: *Aversive Interpersonal Behaviors*. Ed. by Robin M. Kowalski. Boston, MA: Springer US, 1997, pp. 133–170. ISBN: 978-1-4757-9354-3. DOI: `10.1007/978-1-4757-9354-3_7`. URL: `https://doi.org/10.1007/978-1-4757-9354-3_7`.

[167] Matt Wilson. *[v2] xen-netback: allow changing the MAC address of the interface*. 2013. URL: `https://lore.kernel.org/patchwork/patch/353940/` (visited on 27/11/2019).

[168] Simon Xiao. *[net-next] hv_netvsc: Add feature flags NETIF_F_IPV6_CSUM and NETIF_F_TSO6 for netvsc.* 2016. URL: `https://lore.kernel.org/patchwork/patch/643314/` (visited on 23/09/2019).

[169] Arvind Yadav. *[1/2,v2] misc: enclosure: Fix space before '[' error.* 2017. URL: `https://lore.kernel.org/patchwork/patch/856504/` (visited on 13/12/2019).

[170] Robert K. Yin. 'Case Study Research – Design and Methods'. In: (2014).

[171] Minghui Zhou et al. 'On the scalability of Linux kernel maintainers' work'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM. 2017, pp. 27–37.

[172] Mimi Zohar. *[RFC,8/9] ima: include tmpfs in ima_appraise_tcb policy.* 2015. URL: `https://lore.kernel.org/patchwork/patch/531075/` (visited on 27/11/2019).