The Sense of Logging in the Linux Kernel

Keyurbhai Hasmukhbhai Patel

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

August 2020

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Keyurbhai Hasmukhbhai Patel**

Entitled: **The Sense of Logging in the Linux Kernel**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Juergen Rilling

_____ External Examiner
Dr. Olga Ormandjieva

_____ Examiner
Dr. Juergen Rilling

_____ Supervisor
Dr. Abdelwahab Hamou-Lhadj

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2020

_____
Dr. Mourad Debbabi, Interim Dean
Gina Cody School of Engineering and Computer Science

Abstract

The Sense of Logging in the Linux Kernel

Keyurbhai Hasmukhbhai Patel

Logging is an important activity in software engineering. Developers use log data for a variety of tasks including debugging, performance analysis, detection of anomalies, and so on. Despite the importance of this data, the practice of logging still suffers from the lack of common guidelines and best practices. Recently, there have been studies that investigate the practice of logging in C/C++ and Java open-source systems. In this thesis, we complement these studies by presenting the first empirical study aimed at understanding the practice of software logging in the Linux kernel, which is perhaps the most elaborate open source development project in the computer industry. We achieve this by analysing the evolution history of the Linux kernel with a focus on three main aspects: the pervasiveness of logging code, the type of changes made to logging statements, and the rationale behind these changes. Our findings show that logging is pervasive as 3.73% of the Linux kernel source is logging code. We also found that 72.36% of the total number of files in the Linux kernel have at least one logging statement, while only 26.12% of functions are logged. Similar to other studies, the if-block gets the lion's share with 57% of the total number of logging statements. However, the distribution of logging statements across the Linux kernel subsystems and their components vary significantly with no apparent reason, suggesting that developers use different criteria when logging. We also observed that the use of logging has been gradually declining in recent years with a reduction of 9.27% from version v4.3 to version v5.3. This may be due to the availability

of other alternatives to logging such as the use of tracing (a more structured form of logging) and other dynamic analysis techniques. By manually investigating 900 commits that aim to fix or improve logging code, we found that the majority of changes are to fix spelling/grammar mistakes, fix incorrect log levels, and upgrade logging code to use new logging functions to improve the precision and consistency of log output. Based on these findings, we propose many recommendations such as the use of static analysis tools to detect defective logging code at commit-time, use of automatic spell/grammar checkers, adoption of common writing styles for log messages, a systematic review of logging code, and guidance on the use of log levels. We believe that these recommendations can serve as the basis for developing common logging guidelines, as well as better logging processes, tools, and techniques.

Acknowledgments

I would like to express my sincere appreciation to Dr Wahab Hamou-Lhadj for believing in me. His door has always been open to me. I have learnt a lot from you. I would never forget the invaluable help that you provided during my period at Software Research and Technology Lab.

I would like to extend my sincere thanks to the entire team at Ericsson Global AI Accelerator (GAIA) in Montreal, for their feedback and support. I would also like to thank Ericsson GAIA, MITACS, NSERC, and the Gina Cody School of Engineering and Computer Science at Concordia University for their financial support.

I wish to thank all the people from SRT Lab. Especially, MohammadReza Rejali and Mohammed Shehab.

To my friends, Payal, Vrund, Krunal, Mithil, Darshak, and Meet, I am also grateful for your support and help.

Thanks to my parents and sister. Without the financial support from my brother, Suhag, I would have never been able to complete my study in Canada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software logging is a practice that has long been used by developers to record information about a running system. Many studies have demonstrated the importance of log data in various areas including the diagnosis of system failures [Kha+18; El-+20], detection of vulnerabilities and malware infections [Yen+13; Zho+20], profiling of distributed systems [PCZ18], troubleshooting cloud computing environments [KG11; Mir+16], detecting anomalies [Ber+17; IKH18; OAS08], and reliability evaluation of web applications [TRL04].

Despite the importance of log analysis, the practice of logging remains largely *ad-hoc* with no recognized guidelines and systematic processes [YPZ12; Fu+14; Pec+15]. Software developers continue to insert logging statements in the source code without clear and sufficient guidance. Often the decisions on how and where to log are left to the discretion of the developers, resulting in inconsistencies even among developers working on the same projects. In a user study performed at Microsoft, Zhu et al. [Zhu+15] showed that around 68% of the participants find it hard to make decisions on how and where to log. Moreover, the lack of systematic and automated approaches for logging raises serious questions as to the validity of the generated log data as well as the efficacy of existing log analytic tools, which may impede failure diagnosis and other log-related tasks, threatening the stability of deployed software systems.

In recent years, we have seen the emergence of empirical studies investigating the practice of logging in different environments [Pec+15; CJ17b; Zhu+15; Zen+19; YPZ12]. To our knowledge, the first attempt to characterize logging practices in open-source projects was made by Yuan et al. [YPZ12]. The authors analyzed four server-side C/C++ projects and provided many findings with respect to the pervasiveness of logging code, how often logging code is changed, and what kind of changes are made to it. Chen et al. [CJ17b] conducted an extensive replication study, with a focus on 21 open-source projects written in Java.

In this thesis, we build on the studies of Yuan et al. [YPZ12] and Chen et al. [CJ17b] to present the first empirical study that focuses on understanding the practice of logging in the Linux kernel. The Linux kernel is considered as one of the greatest collaborative efforts in the computer industry, with more than a thousand experienced developers contributing to its growth on a regular basis. Many studies have examined the structure and evolution of the Linux kernel from different perspectives [Bag+18; Lot+10; PCW12; Lu+14; IF10], but a little is known about the logging practices followed by Linux kernel developers. Even with extensive documentation, existing Linux kernel development guidelines do not include guidelines for making logging decisions.

We thus in this thesis present the first empirical study that focuses on understanding the practice of logging in the Linux kernel, complementing the aforementioned studies. Similarly to previous studies [YPZ12; CJ17b], we explore three main aspects related to the practice of logging: (1) the pervasiveness of logging in Linux, (2) the types of changes made to logging statements over several releases, and (3) the rationale underlying changes to logging code. From this perspective, this thesis can be seen as a replication study with a focus on the Linux kernel, further contributing to the corpus of knowledge on the practice of logging in large systems. Studying how Linux kernel developers use logging will also reveal specific logging problems related to the Linux kernel and shed further light on the challenges that

developers face when performing logging activities in software engineering.

Our findings show that there is one line of logging code in every 27 lines of code in Linux v5.3 (the latest version when this study was conducted). The pervasiveness of logging; however, varies from one Linux kernel subsystem to another, with *filesystem* and *drivers* being the most logged subsystems. After investigating changes made to logging statements across 22 releases of the Linux kernel, we found that developers tend to modify logging code to improving the quality of the logging output by enhancing precision, conciseness, and consistency. We also found that about one-third of the total number of log level changes were between ERR and DEBUG, suggesting that developers have difficulties distinguishing between fatal and recoverable errors. Through a qualitative analysis of 900 commits that target fixes and improvement of logging code, we found that the reasons behind these changes are similar to those reported by other studies [Has+18; YPZ12; CJ19; CJ17a] such as ambiguous log messages, redundant information, logging the wrong variables, etc. We also found many issues that pertain to the Linux kernel such as revealing sensitive data and problems related to early logging. Based on these result, we propose many recommendations throughout the thesis that are not only applicable to the Linux kernel but can readily be generalized to any software system. Our long-term goal is to help design better processes, techniques, and tools for effective logging, and set the groundwork for establishing logging standards.

The remaining parts of the thesis are structured as follows. The current literature that focuses on *empirical research on logging practices* and *tools that provide logging suggestions* is discussed in Chapter 2. In Chapter 3, we provide an overview of the Linux kernel and how logging activities are performed in the Linux kernel. We present the study design in Chapter 4. The empirical study on logging practices in the Linux kernel is presented in Sections 5.1, 5.2 and 5.3. In Chapter 6, we discuss our findings and implications as well as threats to validity. We conclude the thesis in Chapter 7. Specifically in Section 7.1, we summarize our findings of the study. Moreover, in

Section 7.2, we discuss the directions for future work.

Chapter 2

Related Work

As discussed, although there are previous studies that aimed at understanding logging practices, they targeted systems with characteristics different from the Linux kernel. Our work is thus a replication study, focusing on an extremely relevant long-lived large-scale system. In this section, we provide an overview of the existing literature on software logging, which is complementary to our study. We first focus on *empirical research* on logging practices, and then on *tools* that provide logging suggestions.

2.1 Empirical Research on Logging Practices

A substantial amount of effort has been directed towards the understanding of many aspects of the logging practice in different contexts. In an attempt to understand the characteristics of software logging in open-source projects, Yuan et al. [YPZ12] carried out one of the first investigations to examine in detail the practice of software logging by mining evolution history of a project. Their subject systems included four widely popular C/C++ projects, which are named as follows: Apache httpd, OpenSSH, PostgreSQL, and Squid. While commenting on the pervasiveness of software logging, they reported that 3.30% of the total source code accounts for the logging code. The study also reported that developers often find it hard to get logging statement

right at the first try, as developers changed 36% of all log messages at least once as after-thoughts. Further, they break down the modification to logging code into three categories: *changes to verbosity level*, *changes to static content*, and *changes to variables*. They found that developers often struggle to get verbosity level right at the first try. They also reported that around three quarter (75%) of the log modifications to static content are fixing inconsistent and confusing log messages. They also designed simple verbosity level checker to help developers decide correct log level.

In order to confirm that findings reported by Yuan et al. [YPZ12] holds true or not for software systems written in Java programming language, Chen et al. [CJ17b] conducted a replication the study on popular Java projects which came from diverse domains. Several findings reported by Chen et al. [CJ17b] does not align with the findings reported by Yuan et al. [YPZ12]. For example, Yuan et al. [YPZ12] reported that around 67% of updates to logging code are consistent updates in C/C++ based software projects, while Chen et al. [CJ17b] found that only 41% of updates to logging code are consistent updates in Java-based software projects.

The study conducted by Pecchia et al. [Pec+15], for instance, investigated the practice of logging on the development process of critical software in a particular company. By analyzing more than 2 million de-parameterized log entries, they identified three main reasons for using logging code: state dump, execution tracing, and event reporting. Nevertheless, the usage of logging statements to that purpose presented limitations regarding the lack of standardization over key-value representations and missing contextual information, which could hinder the adoption of automated log analysis tools. The work of Shang et al. [Sha+14] focused on understanding the impact that logging code modifications have on these tools. The authors analyzed the evolution history of two large scale systems and found that 60% of execution logs were changed, which were not necessarily followed by modifications on log analysis tools. Between 10% to 70% of performed changes, however, were found

to be avoidable.

A study on the relationship between logging characteristics and code quality was conducted by Shang et al. [SNH15]. They studied two open-source software systems to find out that it is possible to relate post-release defects to some log-related metrics, even though there is no causal relationship between them. In fact, logging statements tend to be added where developers have concerns about their code. It thus suggests that more maintenance effort should be devoted to files with logging code as they are more susceptible to present problems. The quality of logging code was investigated by Hassani et al. [Has+18]. Their goal was to characterize log-related issues regarding aspects such as the number of files involved on the issue, the time required to fix it, and who provides the fix. In addition, they manually examined more than 500 log-related issues in order to identify common problems associated with logging code, which served as the basis for developing a tool to automatically detect issues such as spelling errors and empty catch blocks.

The study conducted by Zeng et al. [Zen+19], in turn, investigated the use of logging code in open-source Android apps. The authors find that there exists a difference between logging practice observed in mobile apps than those findings reported for desktop and server applications. In contrast to earlier findings, logging is less pervasive in mobile apps. They also found that developers do not actively maintain logging code in mobile applications, as compared to desktop and server counterparts. A recent qualitative study conducted by Li et al. [Li+20] aimed at understanding the benefits developers seek from software logging and the costs associated with it.

2.2 Where, What, and How to Log

There are many decisions that must be made by developers when logging. Focused on understanding where developers log, Fu et al. [Fu+14] performed an empirical

investigation on enterprise applications written in C#. They found that logging statements can be categorized according to five major groups: *assertion-check logging*, *return-value-check logging*, *exception logging*, *logic-branch logging*, and *observing-point logging*. They found that 39%~53% of the logging statements are placed to capture information when the software fails while 47%~61% of the logging statements are used to capture the normal execution flow. To complement their findings, the authors trained a decision tree model that suggests where to log based on contextual keywords extracted from code snippets. A similar tool, named *LogAdvisor*, was built by Zhu et al. [Zhu+15]. It helps developers by providing suggestions on where to log with a balanced accuracy of 84.6%~93.4%. Developers need to carry out logging activities while keeping the performance of the system in mind. Zhao et al. [Zha+17] addressed this issue by proposing a tool named *Log20* that could find an appropriate position for the logging statement to be placed within the given performance constraints.

Because there is no standardization over what information should be included in logs, developers insert logging statements into code in an *ad-hoc* manner. Sometimes, the information from these logs may not be sufficient to diagnose software failures. To handle this issue, Yuan et al. [Yua+12] built a tool called *LogEnhancer*. This tool is able to infer what information could be helpful to narrow down the root cause of failures and to include that information into existing logging code automatically. In order to help developers decide which variables should be included in the logging statement, a recent study by Liu et al. [Liu+19] proposed a recurrent neural network-based model which achieved an average MAP score of 0.84 on nine open-source Java projects. The work of He et al. [He+18] is also focused on what is logged. They manually analyzed 385 logging statements in order to understand the goal of the static text used in these statements. Their results suggest that static text is used essentially for describing program operation, error conditions, and high-level code semantics. To validate their finding, which indicated that there exists a repetitiveness in logging descriptions, they proposed an approach based on simple information

retrieval technique which can generate logging message automatically. The single attempt to address challenges of logging in Linux kernel was made by Senna Tschudin et al. [SLM15]. They proposed a machine learning-based approach to suggest the most appropriate logging function, using the evolution history of the Linux kernel. However, the approach consists of a conceptual description of a framework that still needs to be implemented and validated.

Many previous studies found that developers have a difficult time regarding how to log, especially when they are required to determine the correct log level to be used [YPZ12; CJ17b; Has+18; CJ17a]. Li et al. [LSH17] addressed this difficulty by proposing a prediction model that suggests the most appropriate log level to new logging statements using features such as average log level and log churn. In addition, five anti-patterns were identified by Chen et al. [CJ17a] in a study that analyzed many code-independent log updates from three open-source Java projects. A tool, named *LCAnalyzer* was developed in order to allow the automatic identification of these poor logging practices.

Chapter 3

Background

In this chapter, we present an overview of the Linux kernel project, its structure, the logging libraries used by Linux kernel developers, and how logging in Linux kernel is different from other projects written in C/C++.

3.1 The Linux Kernel

The Linux kernel is a free and open-source software distributed under the GPLv2[1] license. It is responsible for managing the interactions between hardware components and the higher-level programs that make use of them. Used across a wide range of computer systems, from mobile devices and server systems to supercomputers, the Linux kernel can be considered as one of the most important software components in the computing industry. In fact, we are surrounded by the Linux kernel in one way or another. Being developed by roughly fifteen thousand developers throughout its history, as of 2020, it contains around 18M lines of C code. This project is continuously evolving in order to meet both hardware manufacturers' requirements and end-user expectations. In its version 5.3, the Linux kernel received 14,605 commits from 1,881 developers, adding 837,732 and removing 253,255 lines of code, which represents an

[1] https://www.gnu.org/licenses/old-licenses/gpl-2.0.html

increase of 584,477 lines of code when compared to its previous version. [2]

The Linux kernel consists of five major subsystems[3]; each of them covering particular aspects of the project. Table 3.1 shows the system decomposition. The *core* subsystem is in charge of memory management, inter-process communication, management of I/O operations, among others. The *filesystem* subsystem is responsible for providing the file system interface as well as individual file system implementations. The *drivers* subsystem provides device and sound drivers as well as the implementation of cryptography algorithms used within the system and other security-related code. Finally, the *net* and *arch* subsystems are responsible for networking and architecture-specific code, respectively. In the following sections, we frequently refer to these subsystems while discussing our results.

Table 3.1: Linux kernel architectural decomposition

Subsystem	**Top level directories**
core	init, block, ipc, kernel, lib, mm, virt
filesystem	fs
drivers	crypto, drivers, sound, security
net	net
arch	arch

3.2 Logging in the Linux kernel

Log messages are used to record relevant information about a running system. These messages usually comprise three elements: (i) the *log level* of the event being recorded; (ii) a *static message* that describes that event; and, optionally, (iii) *variables* related to the logged event. A log message is typically generated by the execution of a function specifically created for that purpose. These functions can be either project-specific or provided by external logging libraries and frameworks.

[2] https://github.com/gregkh/kernel-history/
[3] https://github.com/gregkh/kernel-history/blob/master/scripts/genstat.pl

The Linux kernel provides to developers its own set of logging functions. One of the simplest ways to write a message to the kernel log buffer is by using the `printk()` function. It is the kernel's equivalent of `printf()`, with the difference that it allows developers to specify the log level of the event being recorded [CRK05]. An example of a log statement is as follows:

```
printk(KERN_ERR "Device initialized with return code %d\n", code);
```

where `KERN_ERR` corresponds to the log level, `"Device initialized with return code %d\n"` is the error message, and `code` is the corresponding variable.

There are eight log levels defined in the Linux kernel, representing different degrees of severity (see Table 3.2). `KERN_WARNING` is the default log level and is assigned to a message in case no log level is specified when calling `printk()`.

Table 3.2: The eight possible log levels defined in the Linux kernel.

Log level	**Description**
`KERN_EMERG`	System is unusable
`KERN_ALERT`	An action must be taken immediately
`KERN_CRIT`	Critical conditions
`KERN_ERR`	Error conditions
`KERN_WARNING`	Warning conditions
`KERN_NOTICE`	Normal but significant condition
`KERN_INFO`	Informational
`KERN_DEBUG`	Debug-level messages

Additional sets of logging functions were introduced in the Linux kernel v1.3.983 with the aim of making logging statements more concise. These functions incorporate log levels in their names. Therefore, in order to log a debug or informational message, instead of using the `printk()` function with the `KERN_DEBUG` and `KERN_INFO` levels as parameters, developers can use the `pr_debug()` and `pr_info()` functions, respectively. Another family of functions specifically designed for device drivers, e.g. `dev_dbg()` and `dev_info()`, automatically include device names in their outputs, making it easier to identify where log messages are originated from. Table 3.3 lists

both sets of functions and their corresponding log levels. Currently, some kernel components present their own logging functions, which are able to generate messages with service-specific information. Examples include network device and TI wl1251 drivers, which provide the `netdev_*()` and `wl1251_*()` families of logging functions, respectively. In this thesis, we may refer to logging functions and logging macros interchangeably.

Table 3.3: New sets of logging functions.

Log level	**`pr_*()` function**	**`dev_*()` function**
KERN_EMERG	pr_emerg()	dev_emerg()
KERN_ALERT	pr_alert()	dev_alert()
KERN_CRIT	pr_crit()	dev_crit()
KERN_ERR	pr_err()	dev_err()
KERN_WARNING	pr_warn()	dev_warn()
KERN_NOTICE	pr_notice()	dev_notice()
KERN_INFO	pr_info()	dev_info()
KERN_DEBUG	pr_debug()	dev_dbg()

Chapter 4

Study Design

4.1 Research Questions

In this study, our goal is to characterize how logging is carried out in the Linux kernel. To do so, we analyze the project considering three broader aspects: (i) how pervasive is logging in the Linux kernel and where logging code is located; (ii) how it changes; and (iii) why it changes. To achieve this goal, we focused on answering the following research questions.

RQ1 *What is the pervasiveness of logging in the Linux kernel?*

RQ2 *How does the logging code in the Linux kernel evolve?*

RQ3 *What are the characteristics of changes made to logging code as afterthoughts?*

By answering RQ1, we aim to understand whether logging is a practice consistently adopted by Linux kernel developers or not. Identifying where logging statements are located can also provide insights into the reasons they are placed in these locations. With RQ2, in turn, we focus on investigating the life cycle of logging statements, i.e. when they are created, updated or removed. These two research questions are similar to questions answered in previous studies [YPZ12; CJ17b].

This allow us not only to understand these aspects in a project with distinguished characteristics but also to contrast obtained results.

Previous research studies classified changes to logging code as code consistent and afterthought changes [YPZ12; CJ17b]. Consistent changes are those changes made to logging statements as a result of changes made in other parts of the code. Renaming a variable used in the logging statement in order to reflect the same modification in the code is an example of a code consistent change. Afterthought changes, in turn, are those changes made to logging statements in order to fix or improve the logging statements in question. In this thesis, we are interested in investigating afterthought changes to provide insight on the effort required to maintain logging code. By answering RQ3, we also aim to identify the reasons behind afterthought changes, which we hope can help researchers and practitioners develop techniques to prevent, detect and fix such changes automatically.

4.2 Subject Project

In Section 3 we introduced the Linux kernel project, which is the target of our study. We study the development history between v4.3 [1] and v5.3 [2] of the Linux kernel, comprising 285,045 commits. This corresponds to an interval of almost four years of software development, from November 2015 to September 2019. All releases used in this study are available in the official Linux kernel repository[3].

4.3 Identification of logging functions

Traditional approaches to automatically locate logging statements into the source code rely on logging functions that are known in advance (e.g., Log4J in Java) or use regular expression looking for variations of the term ”log”. Although these methods

[1] https://github.com/torvalds/linux/commit/6a13feb9
[2] https://github.com/torvalds/linux/commit/4d856f72
[3] https://github.com/torvalds/linux

have been successfully adopted in previous studies [SNH15; Zhu+15; CJ17b; YPZ12], we cannot apply them to the Linux kernel. This is because the Linux kernel uses a wide variety of functions and macros for logging purposes (see Section 3), and it is not unusual to find situations in which customized calls to these macros are implemented by particular services (e.g, `_enter` [4], `pr_pic_unimpl` [5]). Listing 4.1 presents an example of such a situation. A macro named `adsp_dbg()` is implemented with the aim of ensuring the invocation of `dev_dbg()` with a particular set of parameters and formatting (lines 2–3), thus preventing the need for the specification of such elements in further calls (line 5). This type of logging statements cannot be detected unless we know that `adsp_dbg()` is a logging function.

```
1 [...]
2 #define adsp_dbg(_dsp, fmt, ...) \
3         dev_dbg(_dsp->dev, "%s: " fmt, _dsp->name, ##__VA_ARGS__)
4 [...]
5 adsp_dbg(dsp, "Wrote %zu bytes to %x\n", len, reg);
6 [...]
```

Listing 4.1: Logging macro defined in /sound/soc/codecs/wm_adsp.c

In this study, to identify logging functions and macros in the Linux kernel, we take advantage of the pattern-based method presented by Senna Tschudin et al. [SLM15]. The authors proposed an identification method, which consists of three semantic patterns that, despite describing general properties when taken individually, are able to correctly characterize logging functions when combined, yielding a low number of false positives.

The first pattern states that logging functions should be called at least once inside an `if` block that ends with a `return` statement. The second pattern describes logging functions as those functions that have at least one string argument, representing a

[4] https://github.com/torvalds/linux/blob/v5.3/fs/afs/internal.h#L1449
[5] https://github.com/torvalds/linux/blob/v5.3/arch/x86/kvm/i8259.c#L37

log message. Finally, the third pattern requires logging functions to have a variable number of arguments. Listing 4.2 shows how macro `adsp_dbg()` satisfies all three patterns and thus can be considered as a logging function.

```
1 [...]
2 if (val == 0) {
3     adsp_dbg(dsp, "Acked control ACKED at poll %u\n", i);
4     return 0;
5 }
6 [...]
7 adsp_dbg(dsp, "Wrote %zu bytes to %x\n", len, reg);
```

Listing 4.2: A logging macro defined in /sound/soc/codecs/wm_adsp.c that satisfies the identification patterns.

Although, this approach yields good results, it misses logging functions with a fixed number of arguments (e.g., `PRINTK_2` [6], `PRINTK_4` [7]) or logging functions that lack variability in their use. These functions do not satisfy the third pattern. We, therefore, decided to relax the third pattern by including logging functions that contain a fixed number of arguments (i.e., the ones that are not overloaded). In addition to these patterns, we reviewed manually the resulting logging functions generated in the Linux kernel versions studied in this thesis to ensure their correctness. Once we know which functions/macros are used for logging, we simply write a script to detect all calls to these functions/macros to retrieve the specific logging statements.

[6] https://github.com/torvalds/linux/blob/v5.3/drivers/char/mwave/mwavedd.h#L79
[7] https://github.com/torvalds/linux/blob/v5.3/drivers/char/mwave/mwavedd.h#L89

Chapter 5

Empirical Study

In this chapter, we will present results addressing each research questions as stated in Section 4.1.

5.1 RQ1: What is the pervasiveness of logging in the Linux kernel?

To examine the pervasiveness of logging in the Linux kernel, we consider its latest available version when we conducted this study (v5.3). We extract a set of metrics to quantitatively assess the widespread usage of logging statements with respect to different levels of granularity (Linux kernel subsystems, files, and programming constructs) (see Fig. 5.1). We detail these metrics and how they are collected in Section 5.1.1 and present our results in Section 5.1.2.

5.1.1 Data Gathering and Extraction

To assess the pervasiveness of logging code in Linux kernel, we extracted the number of *lines of source code* (SLOC) and the number of *lines of logging code* (LLOC), similar to previous studies (e.g., Yuan et al. [YPZ12]). We only included `.c` files in our analysis. (Header files were not taken into account). Comments and empty

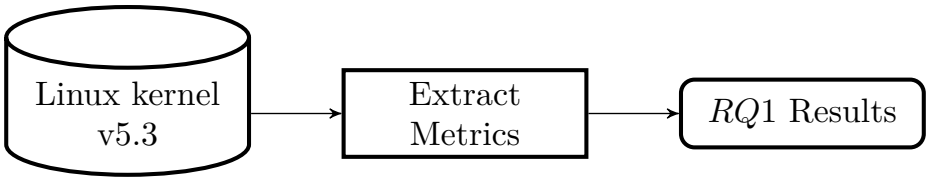


Figure 5.1: RQ1 data collection process

lines were removed when measuring SLOC. These metrics were collected considering different levels of granularity, namely (i) the overall system and subsystem level; (ii) the file level; and (iii) the programming construct level. The latter comprises functions, as well as `do-while`, `if`, `else`, `else-if`, `for`, `switch`, and `while` blocks. We also computed the *Log Density* [YPZ12; CJ17b; Zen+19] and *Log Ratio* metrics (see Equations 1 and 2). The log density measures the number of lines of source code per line of logging code, whereas the log ratio measures the number of lines of logging code per line of source code. For example, a log density of 10 would mean that for every ten lines of the source code, there is one line of the logging code. A log ratio of 0.2 means that 20% of the source code consists of the logging code. Table 5.1 summarizes the metrics used to measure the pervasiveness of logging in the Linux kernel.

$$Log\ Density = \frac{SLOC}{LLOC} \tag{1}$$

$$Log\ Ratio = \frac{LLOC}{SLOC} \tag{2}$$

5.1.2 Data Analysis

5.1.2.1 System and subsystem level

We analyzed the log density and log ratio in the Linux kernel and its subsystems. Table 5.2 shows the results. We found that from a total of 13,390,131 lines of source

Table 5.1: Metrics used to answer RQ1

Metric	**Description**
SLOC	Number of lines of source code excluding comments and empty lines
LLOC	Number of lines of logging code
Log Density	The number of lines of source code per line of logging code
Log Ratio	The percentage of number of lines of logging code per source line of code

code, 3.73% are lines of logging code. It is equivalent to a logging density of 27, which means that for every 27 lines of source code in the Linux kernel, we have one line of the logging code. This finding is in line with that from Yuan et al. [YPZ12], which reported an average log density of 30 for four C/C++ applications. On the other hand, this result differs from the study of Chen et al. [CJ17b], which reported an average log density of 50 for 21 Java applications. Although these results are not conclusive, they suggest that the language in which the project is written may affect in the prevalence of logging statements in the source code.

When looking at each subsystem individually, we obtain heterogeneous results. The log ratio ranges from 1.94% to 4.26%. A deeper look into the components of the Linux kernel subsystems shows important discrepancies. For example, we observe that the *init* component of *core* subsystem, which consists of only 2,935 SLOC, contains 192 lines of logging code (6.54% log ratio). This is considerably higher than the other components. This may be explained by the fact that the *init* component is responsible for the initialization of the console and other key kernel services such as the security framework, scheduler, memory allocation [Boo20]. For these services, it is important to log all possible errors in order to quickly debug potential failures. However, the idea that critical components are logged the most does not always hold. Take for example the *ipc* component from the *core* subsystem with the highest log density (498). This component, which contains only six files, is responsible for setting up the inter-process communication mechanisms on which the processes rely for communication with each other and coordination. Despite

Table 5.2: Summary of metrics collected from the entire system and its subsystems

Subsystem	Component	SLOC	LLOC	Log Density	Log Ratio
core	lib	106,577	2,296	46	2.15%
	kernel	208,000	4,396	47	2.11%
	mm	86,921	1,940	45	2.23%
	block	31,846	713	45	2.24%
	ipc	6,468	13	498	0.20%
	init	2,935	192	15	6.54%
	virt	16,839	184	92	1.09%
		459,586	9,734	47	2.12%
filesystem	fs	840,527	35,780	23	4.26%
drivers	drivers	9,358,913	384,269	24	4.11%
	sound	765,988	19,673	39	2.57%
	security	57,974	1,192	49	2.06%
	crypto	52,888	835	63	1.58%
		10,235,763	405,969	25	3.97%
net	net	746,263	14,490	52	1.94%
arch	arch	1,107,992	32,924	34	2.97%
Total		**13,390,131**	**498,897**	**27**	**3.73%**

being critical, this component is the least logged. Without further studies, we can only attribute these variations to the fact that different groups of developers are maintaining each subsystem, and there are no recognized (or common) guidelines on how to log.

> **Finding 1:** The log density in the Linux kernel v5.3 is 27, i.e., there is one line of logging code for every 27 lines of the source code. Logging code represents 3.73% of the total source code, with *filesystem* and *drivers* being the most logged subsystems with a log ratio of 4.26% and 3.97% and a log density of 23 and 25, respectively.

5.1.2.2 File level

Similar to the previous study [LSS15], we also calculate the log ratio at the file level. Note that we do not use log density here due to the fact that many files do not

contain logging statements. Log density will not make sense in such cases. The log ratio provides a more accurate measurement by assessing the pervasiveness of logging across the Linux kernel files. Figure 5.2 shows the results. We found that 94.34% of the total number of files in the Linux kernel v5.3 have a log ratio between 0% and 10%. More precisely, 7,134 files (27.64%) do not contain any logging statements, while a significant number of files (66.70%) have a log ratio greater than 0% and less than or equal to 10%. Files with a log ratio greater than 10% account for only 5.66% of the total number of files. We found a very small number of file with a log ratio close to 90%. We manually inspected these files and found that they contain mostly debugging routines. For example, `drivers/scsi/qla4xxx/ql4_dbg.c` contains functions, which dump relevant information about the Linux Host Adapter structure.

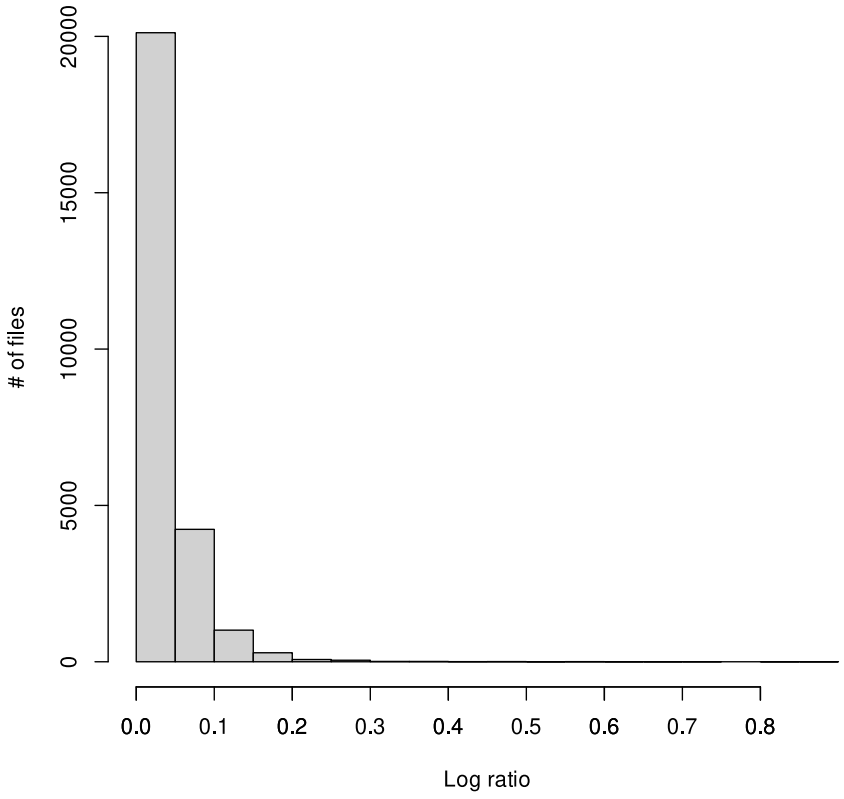


Figure 5.2: Value of log ratio at the file level (Total #files = 25,814)

Finding 2: We found that 72.36% of the total number of files in the Linux kernel have at least one logging statement.

5.1.2.3 Programming construct level

In this section, we measure the number of logging statements in program constructs of the Linux kernel including functions, `do-while`, `if`, `else`, `else-if`, `for`, `switch`, and `while`. Determining where logging statements are located can help understand the purpose they serve. There exist studies that examine the location of logging statements in source code, among which we found that the work of Pecchia et al. [Pec+15] very comprehensive in the sense that they measured the number of logging statements in the same program constructs as the ones listed above. Their work was on two large C/C++ industrial applications. We followed the same approach and compared our results to theirs.

Table 5.3 shows the distribution of logging statements in the kernel functions. We found that from a total of 476,522 functions, 352,045 (73.88%) do not have any logging statements. For the remaining functions (26.12%), 91.84% of these have the number of logging statements in the range of one to five. Less than 1% of the total number of functions have the number of logging statements more than eight.

Table 5.3: Counting of functions by their number of logging statements

#Logging Statements	#Functions
0	352,045 (73.88%)
1	64,553 (13.55%)
2	25,823 (5.42%)
3	12,432 (2.61%)
4	7,075 (1.48%)
5	4,432 (0.93%)
6	2,752 (0.58%)
7	1,902 (0.40%)
8	1,282 (0.27%)
≥ 9	4,226 (0.89%)
Total	476,522 (100.00%)

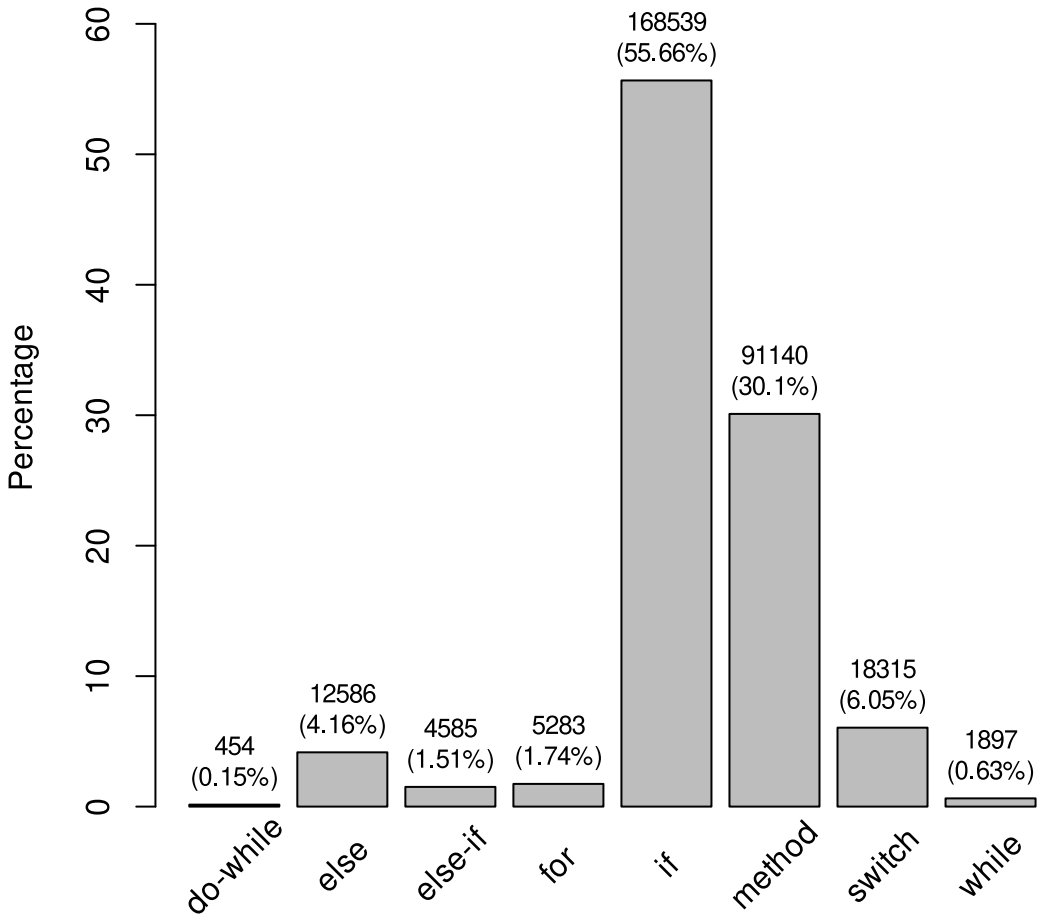


Figure 5.3: Distribution of logging statements across different program constructs

Figure 5.3 shows the distribution of logging statements in different program constructs. We found that 55.66% (168,539 out of 302,799) of logging statements are used inside the *if* block and 4,585 (1.51%) inside the *else-if*. Together, they represent 57.17%, which is similar to the study of Pecchia et al. [Pec+15] who reported that around 60% of the total logging statements are used inside the `if` blocks. These logging statements are typically used for logging errors after checking the return value of function calls.

The logging statements in the *else* block accounts for 4.16% (12,586 out of total 302,799 logging statements). The *switch* block, which is another control statement available in C, we found that they account for 6.05% of the total logging statements. The logging statements used directly inside loop controls represents only 2.52% of the total logging statements. The logging statements used directly inside functions (i.e., not in any of the program constructs) accounts for 30.10% of the total number of logging statements.

Finding 3: We found that only 26.12% of the functions in the Linux kernel have one or more logging statements. Further studies, such as the work from Li et al. [Li+18], are needed in order to understand whether there is any relationship between *topic* of source code and it having a logging statement.

Finding 4: 57.17% of the logging statements are used inside an *if* and *else-if* blocks, while 30.10% are used inside functions directly.

5.2 RQ2: How does the logging code in the Linux kernel evolve?

In the previous section, we showed that the use of logging code is widespread in the Linux kernel (3.73% of the Linux kernel source code is logging code). In this RQ, similar to previous studies [Li+19a; YPZ12; CJ17b], we want to understand how

often the logging code changes over various releases of the Linux kernel. We do this by measuring the number of logging statements that are added, deleted, or updated. In addition, we examine the type of updates made to logging statements such as changes to logging message, log level, and logged variables (dynamic part). Similar to RQ1, we compare our findings to the studies conducted by Yuan et al. [YPZ12] (C/C++ systems) and that of Chen et al. [CJ17b] (Java systems) whenever applicable.

5.2.1 Data Gathering and Extraction

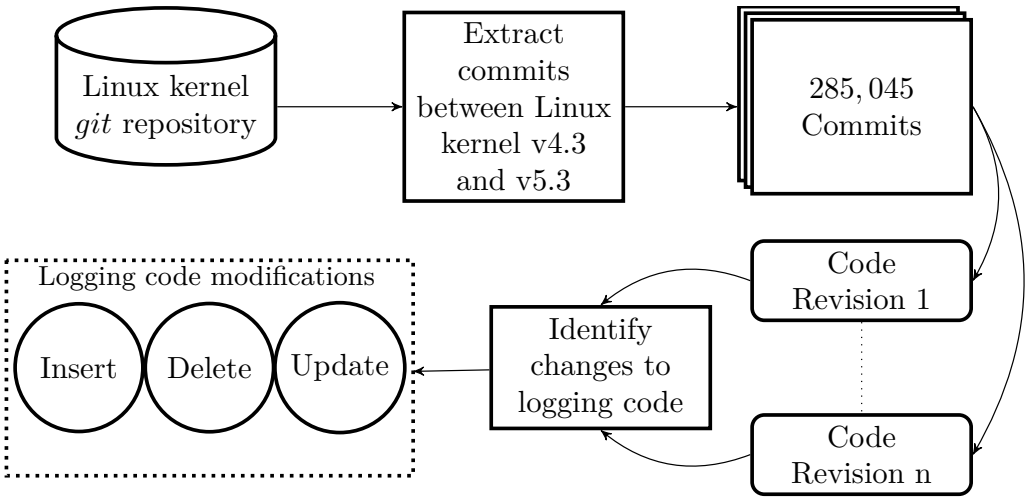


Figure 5.4: Retrieving changes to logging statements

5.2.1.1 Use of logging statements

To understand how the use of logging code has changed over various Linux kernel releases, we study the evolution history between the Linux kernel v4.3 and v5.3, which represents 22 releases covering the period from November 2015 to September 2019, with a total of 285,045 commits. Given the number of releases and commits, this dataset is representative of the changes made to logging statements over the years. For each release, we first identify the logging functions used in that release using the approach described in Section 4.3. We then retrieve calls to these functions and calculate the log ratio (LLOC/SLOC) for each release.

5.2.1.2 Retrieving changes to logging statements

To understand how logging code evolves in the Linux kernel, we study three types of modifications made to logging code—log insertion, log deletion, and log update—similarly to the previous work [YPZ12; CJ17b; Zen+19]. We further divide the insertion and deletion of logging statements into two categories: logging statements that are added or deleted along with the addition or deletion of the files.

We use the `git rev-list` command with `--no-merges` option to retrieve the commits. Note that the `--no-merges` option omits commits with more than one parent. This is necessary in order to avoid duplicate commits. For each commit, we extract two adjacent versions of each file that are changed in the commit. We only consider `.c` files. Then, in order to generate an edit script representing syntactic modifications by inferring changes at the level of the abstract syntax tree, we use GumTree, which is state of the art AST differencing tool [Fal+14]. Figure 5.4 shows our workflow for retrieving changes to logging statements.

5.2.2 Data Analysis

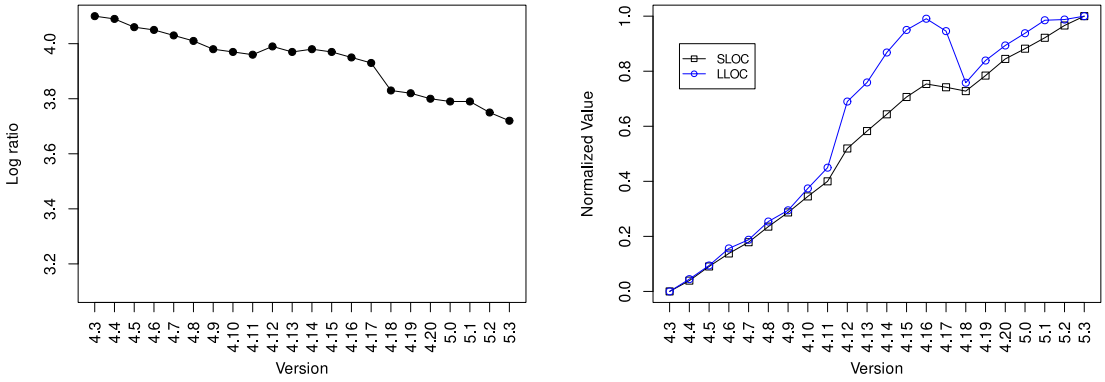
5.2.2.1 How does the use of logging code evolve over in the Linux kernel?

We first analyze the size of logging code, looking at two perspectives. First, we observe how the proportion of logging code evolved, measured by the log ratio metric. Second, we compare the evolution of SLOC and LLOC. The evolution of the log ratio metric for the Linux system from v4.3 to v5.3 can be seen in Figure 5.5a. It is possible to observe that it been decreasing over the years. This analysis is complemented by Figure 5.5b, which indicates how LLOC and SLOC, individually, increased across the different versions. SLOC and LLOC are normalized using min-max normalization to fall in the $[0, 1]$ interval.

We can see in Figure 5.5b that the SLOC and LLOC curves follow the same trend, except v4.11 to v4.18. By going through the Linux kernel changelogs, we found

that these inconsistencies seem to be the result of the addition/removal of drivers or filesystems, which contained a large amount of logging code. The Linux kernel v4.12 added Intel *atomisp* camera drivers (commit `a49d253`) and rtl8723bs sdio wifi driver (commit `554c0a3a`). These two changes increased the number of logging statements by 6,103, which contributed to a sharp increase in the number of lines of logging code. Similarly, we can see a sudden decrease in SLOC and LLOC between v4.16 and v4.18 releases. After inspecting the Linux kernel v4.18 changelogs, we found that SLOC of the Linux kernel v4.18 was smaller than its previous release, and that occurred just three times in the history of the Linux kernel before the release of kernel v4.18. The reason for this can be attributed to removal of the lustre filesystem (commit `be65f9e`) and the atomisp driver (commit `51b8dc51`), which earlier contributed to 10,442 logging statements.

From Figure 5.5a and Figure 5.5b, we can see that despite the fact that the newer versions of Linux kernel contain a higher number of lines of code, the log ratio is lower than the previous versions. In other words, Linux kernel developers do not log as much as before if we compare the number of lines of logging code to the number of lines of the source code. This may be due to the fact that the code has become more mature over the years, reducing the need to track faults and failures or to the increase in the number of debugging and tracing tools, which can be used to diagnose problems as shown by Corbet [Cor16] and Edge [Edg19]. The authors noticed an increase in the use of tracepoints rather than simple `printk()` in recent versions of Linux kernel. The term tracing is used here to show the flow of execution of specific program constructs, for example, traces of routine calls, system calls, etc. There exist several studies that investigate the use of traces in software engineering (e.g., Hamou-Lhadj et al. [HL02; HL04]). An opportunity for future research is to study the relationship between tracing and logging, two powerful dynamic analysis approaches.



(a) Evolution in the value of log ratio (b) Evolution of SLOC and LLOC

Figure 5.5: Evolution in the use of logging code between Linux v4.3 and v5.3

Finding 5: The log ratio (LLOC/SLOC) has been gradually declining from v4.3 to v5.3 to go from 4.10% in v4.3 to 3.72% in v5.3, a net reduction of 9.27%. SLOC and LLOC are closely correlated across most versions of Linux kernel between v4.3 and v5.3.

Similar to previous studies [Li+19a; YPZ12; CJ17b; Zen+19], to understand how often a source code revision involves a modification to logging code, we calculate the ratio of the number of commits that involve modifications to logging code (i.e., addition, deletion, or update to the logging statement) to the total number of commits. Out a total of 285,045 commits, we found that 39,351 (14%) commits involve modifications to logging code. This result compares to that of the study of Yuan et al. [YPZ12] on C/C++ systems who found that 18% of the commits they studied involve log modifications. It is also similar to the results reported by Chen et al. [CJ17b] who found that there are around 16% of such commits when studying Java systems.

Finding 6: We found that 14% of commits made between the Linux kernel versions v4.3 and v5.3 involves modifications to logging code.

5.2.2.2 What type of modifications are made to logging statements?

Given that logging code changes over time, we now look at the types of modifications that are made. Table 5.4 shows the distribution of modifications made to logging code in terms of the number of logging statements added, deleted, and updated between Linux versions v4.3 and v5.3. There are 211,437 logging statement modifications, out of which 24.78% are log updates, 45.99% are log insertions, and log deletions account for 29.23%. The *drivers* subsystem alone represents 86.60% of the total modifications made to logging code, followed by *arch* (5.47%), and *filesystem* (3.74%). This is somewhat expected since the *drivers* subsystem is considerably larger (over 10 million SLOC) than the other subsystems, see Table 5.2.

The percentage of log additions in the Linux kernel is similar to the one reported by Chen et al. [CJ17b] who showed that log additions contribute to 18%~41% of the total log modifications. A similar result has also been observed by Zeng et al. [Zen+19] when studying log practices in Android applications (55.5% of the total modifications). Yuan et al. [YPZ12] did not report the percentage of log additions in their examination of C/C++ systems.

We found that the number of log deletions accounts for 29.23% of the total number of modifications made to logging statements. This result differs significantly from work of Yuan et al. [YPZ12] who reported that the number of log deletions is only 2% of the total number of modifications. Our result is however similar to that of Chen et al. [CJ17b] who reported that log deletion contributes to 26% of the total modifications in Java systems. Such evolution of logging code may adversely impact the bug triaging process as developers rely on the logs contained in the bug reports as noted by Ran [Ran19]. The authors found that it is not possible to rebuild the execution paths for bug reproduction from logs embedded in the bug report in 34% cases. They also argued that the continuous evolution of system logs could have an effect on the accuracy of log processing tools and machine learning models deployed for identifying anomalous activities, as models need to be retrained whenever logging

statements are changed.

To drill down, we categorized log addition and deletions into two categories: added along with the addition of a new file and deleted along with the deletion of an existing file. Table 5.5 shows a detailed breakdown. We found that 55.77% of all log insertions were made along with the addition of new files. Similarly, we found that 55.34% of the deleted logging statements were deleted along with the deletion of existing files.

Table 5.4: Distribution of modifications made to logging statements

Subsystem	**Insertion**	**Deletion**	**Update**
arch	3,987	4,620	2,962
core	1,863	647	1,700
driver	87,125	53,235	42,748
fs	2,776	1,626	3,502
net	1,490	1,677	1,479
Total	97,241 (45.99%)	61,805 (29.23%)	52,391 (24.78%)

Table 5.5: Breakdown of logging statements added or deleted along with the file

Subsystem	**Added with a new file**	**Deleted with an existing file**
arch	1,289	3,038
core	614	84
driver	51,178	30,452
fs	833	409
net	320	218
Total	54,234 (55.77%)	34,201 (55.34%)

Finding 7: Out of the 211,437 logging statements modifications, 24.78% are log updates, 45.99% are log insertions, and 29.23% are log deletions. The majority of changes to logging code (86.60%) are made in the *drivers* subsystem.

5.2.2.3 Which updates are made to logging statements?

We now proceed to a deeper analysis of log changes, analyzing how log statements are updated. We classify log updates into three categories depending on which part of the

logging statement has been modified including the logging function (or macro), the static content representing the log message, the log level, and the dynamic content, i.e., the variables and function calls. Table 5.6 reports the results. Note that one update may consist of one or more changes to the same logging statement. For example, if the log function and the static content of one logging statement have changed, this will be counted as two updates. This explains why the total number of log updates can be higher than the total number of updated logging statements. We next analyze the different categories of updates.

Table 5.6: Distribution of updates made to logging statements

Subsystem	Logging function	Dynamic content	Static content
arch	1,226	1,284	2,014
core	606	721	979
driver	15,685	28,501	23,127
fs	1,344	1,863	2,422
net	435	952	809
Total	19,296 (36.83%)	33,321 (63.60%)	29,351 (56.02%)

Changes made to the logging function: Of the 52,391 updated logging statements, 19,296 (36.83%) include modifications to the logging functions that were used as shown in Table 5.7. We only show important changes in this table, which we define as changes that appear at least 100 times. Showing all the changes will take too much space without necessarily adding much value to the analysis.

The analysis of the changes to the logging functions revealed that 6,512 (33.75%) of these updates are changes between `printk`, `pr_<*>`, and `dev_<*>` macros. For example, in commit `26a0a10`, a developer updated logging statements from `printk` to using device-aware `dev_err()/dev_info()` logging functions to improve the precision of the resulting logs by including device-specific information. This is further cemented by an observation that there has been a steady decrease in the usage of the `printk()` function, with a usage reduction of 29.32% between version 4.3 and 5.3 of the Linux kernel. On the other hand, we find that the use of `pr_*()` and `dev_*()` functions is

gradually increasing, as more and more `printk()` call sites are now being converted to use them [Cor12].

We also found an increasing use of so-called "rate limited" logging functions such as the `<*>[_once/_ratelimited]` [1] family of macros, which can be seen in commit `527aa2a`, where all calls to `pr_info` were converted to `pr_info_ratelimited`. The objective of this type of functions is to prevent overloading the log buffers by controlling the amount of logs generated in a given period of time.

Moreover, we observed that many changes to logging functions are triggered by the need to make them more concise. For example, in commit `466414a`, a developer introduced `btc_alg_dbg` and `btc_iface_dbg` logging macros, and converted all calls to `BTC_PRINTK` to the new logging macros (`btc_<*>_dbg`). The benefit is that software developers do not have to specify `btc_msg_type` in the function argument, resulting in more concise logging statements. Ten months later, in commit `10468c3`, all calls to `btc_<*>_dbg` were again changed to another logging function, named `RT_TRACE`, to be consistent with the use of this function in other drivers.

We thus conclude from the above observations that changes to logging functions are aimed at improving the quality of the logging output by either enhancing precision, conciseness, or consistency. However, after analyzing many commits related to log updates, we could not find any evidence that there was a Linux-wide strategy, which suggests that these updates are a result of pre-established guidelines. It appears that the decision on how to log is left to the discretion of the developers.

> **Finding 8:** The changes to logging functions are triggered by the need to improve the quality of the logging output by either enhancing precision, conciseness, or consistency.

The second reason for logging function modification is changing the severity of logging statements by specifying log levels. The `printk` function allows one of the eight log levels defined in `/include/linux/kern_levels.h`. For example,

[1] https://github.com/torvalds/linux/blob/v5.3/include/linux/printk.h#L418

Table 5.7: Frequency of changes made to logging functions (frequency > 100)

Old logging function	New logging function	Frequency
printk	pr_err	950
printk	pr_info	881
BTC_PRINT	btc_alg_dbg	663
btc_alg_dbg	RT_TRACE	621
printk	pr_cont	606
printk	pr_warn	591
PDEBUG	gspca_dbg	477
pr_err	dev_err	440
pr_info	ioc_info	418
PDBG	pr_debug	392
dev_err	DRM_DEV_ERROR	387
pr_debug	dev_dbg	369
test_msg	test_err	345
brcmf_err	bphy_err	315
RT_TRACE	pr_err	275
pr_warning	pr_warn	262
pr_err	ioc_err	231
printk	pr_debug	226
BT_ERR	bt_dev_err	205
DTRACE	dml_print	173
PRINT_ER	netdev_err	169
dev_dbg	musb_dbg	163
pr_info	dev_info	161
SSI_LOG_DEBUG	dev_dbg	156
BUGMSG	arc_printk	153
SSI_LOG_ERR	dev_err	152
dev_info	dev_dbg	151
dev_err	dev_dbg	146
DRM_ERROR	DRM_DEBUG	145
dev_info	pci_info	139
PERR	gspca_err	129
btc_iface_dbg	RT_TRACE	119
BTC_PRINT	btc_iface_dbg	119
pr_info	pr_debug	119
gvt_err	gvt_vgpu_err	118
DRM_ERROR	DRM_DEV_ERROR	104
pr_warn	dev_warn	102
pr_info	pr_info_ratelimited	102
pr_err	pr_debug	100

```
printk(KERN_ERR "GCT Node MAGIC incorrect - GCT invalid\n");
```

However, the new set of logging API introduced in Linux kernel 1.3.98[2] embedded log levels into the function names, such as `pr_debug` and `pr_info`.

Table 5.8 provides the distribution of changes made to the severity of logging statements. If the developer updates a logging statement to use a new logging function, while keeping the same log level, we do not consider this change as a log level change. As stated in previous work [YPZ12], developers often fail at determining how critical an error is in the first attempt. This observation is confirmed by the finding from our study that, out of total 4,127 log level changes, approximately 1/3 of the total log level changes were between ERR and DEBUG log levels. Specifically, 720 (17.45%) logging statements lowered the severity of the log message from ERR to DEBUG, and 616 (14.93%) from ERR from DEBUG. We found a total of 1,522 (36.88%) instances where developers increased the severity of a logging statement to increase their visibility. In addition, logging debugging messages at the ERROR level would result in log flooding, making it difficult to diagnose the real problems. We found that a total of 2,605 (63.12%) instances where developers reduced the severity of a logging statement in order to prevent log flooding. In fact, of these 4,127 log level modifications, 3,832 (92.85%) changes are between ERR, WARNING, INFO, and DEBUG log levels.

> **Finding 9:** We found that 92.85% of the changes of log levels are between ERR, WARNING, INFO, and DEBUG log levels, suggesting that it is difficult for Linux developers to decide on which log level to use.

Changes made to the dynamic content: Of the 52,391 updated logging statements, 33,321 (63.60%) include modifications to the dynamic content (i.e., variables and function calls). Yuan et al. [YPZ12] found that developers often add variables into existing logging statements as afterthoughts, which can aid in the failure

[2] https://repo.or.cz/davej-history.git?a=commit;h=aa66269c

Table 5.8: Breakdown of the log level changes made between Linux kernel v4.3 and Linux kernel v5.3.

Old \ New	EMERG	ALERT	CRIT	ERR	WARN	NOTICE	INFO	DEBUG
EMERG	**0**	1	10	51	30	0	1	0
ALERT	0	**0**	6	8	10	0	0	0
CRIT	1	0	**0**	8	3	0	0	0
ERR	1	0	5	**0**	242	19	195	720
WARN	22	2	14	343	**0**	14	336	346
NOTICE	0	0	0	10	5	**0**	21	22
INFO	1	0	1	188	62	23	**0**	562
DEBUG	0	0	0	616	77	6	145	**0**

diagnosis process. However, changes to dynamic content represented only 27% of all log modifications in their study. [YPZ12].

Our finding is; however, in line with the study of [Li+19a] on 12 C/C++ open-source projects, where the authors found that 69.1% revisions made modifications to the dynamic content of logging statements. This high number regarding changes to dynamic content has also been observed by Chen et al. [CJ17b] and Zeng et al. [Zen+19] where they studied server/desktop application and android application written in Java, respectively.

One possible explanation for this high number of changes of the logging statement dynamic content in Linux kernel can be seen in commit `6be9005`, where the developer switched to `DRM_DEV_DEBUG_<*>` instead of `DRM_DEBUG`. `DRM_DEV_DEBUG_<*>` are device-aware logging macros and they require `struct *device` as an argument to include device name in the log output. In order to help developers decide which variables should be included in the logging statement, a recent study by Liu et al. [Liu+19] proposed a deep learning-based approach which achieved an average MAP score of 0.84 on nine open-source Java projects. Such approaches tailored to the Linux kernel domain could be helpful to alleviate this problem of what information should be included in the logging statement.

Changes made to the static content: 56.02% of the 52,391 changed logging

statements include modifications to the static content (i.e., the log message). A similar result has also been observed by Chen et al. [CJ17b] and Yuan et al. [YPZ12] who reported a range of 14%~65% and 18%~56%, respectively. Prior studies list *fixing inconsistency*, *clarification*, and *spelling/grammar mistakes* as the major causes of this type of modifications [CJ17b; YPZ12; CJ19]. Deep learning based approaches similar to Panthaplackel et al. [Pan+20] should be explored to automatically update the static text in the logging statements based on changes made to the surrounding source code.

> **Finding 10:** We found that 63.60% of the updated logging statements made modifications to the dynamic content, while 56.02% made changes to the static content.

5.3 RQ3: What are the characteristics of changes made to logging code as afterthoughts?

A change to logging code is considered as an afterthought if the change explicitly addresses a bug caused by or is used to enhance the logging code [YPZ12; CJ17b]. Afterthought changes are different from log updates seen in RQ2. In RQ2, a change to a logging statement can be triggered by changes to other parts of the code. For example, if the developer changes the name of a variable that is logged, then change to the corresponding logging statement is needed. Afterthought changes, on the other hand, take the form of commits that explicitly target logging statements to fix bugs caused by these statements or for enhancement purposes. It is essential to study afterthought changes because they add to the overall maintenance effort. Having a large number of afterthought log changes may defeat the very purpose of logging, which is to reduce the maintenance effort by facilitating debugging and other failure diagnosis tasks.

An example of a commit addressing a problem associated with logging code can

be seen in Listing 5.1, where an error message is displayed by the thinkpad_acpi driver when brightness interfaces are not supported, encouraging the user to contact IBM for this problem. However, according to the developer who handled this commit, back-light interfaces on newer devices are supported by the i915 driver. The developer decided to change the log level from *"error"* to *"info"* to reduce the visibility of the log message. Another example can be seen in commit `e404f94`. In this revision, the developer decided to enhance the existing logging statements by including `qp_num(qp)` in debug messages in order to improve debugging tasks.

```
--- a/drivers/platform/x86/thinkpad_acpi.c
+++ b/drivers/platform/x86/thinkpad_acpi.c
@@ -6459,8 +6459,7 @@ static void __init
   ↪ tpacpi_detect_brightness_capabilities(void)
          pr_info("detected a 8-level brightness capable ThinkPad\n");
          break;
   default:
-         pr_err("Unsupported brightness interface, "
-             "please contact %s\n", TPACPI_MAIL);
+         pr_info("Unsupported brightness interface\n");
          tp_features.bright_unkfw = 1;
          bright_maxlvl = b - 1;
   }
```

Listing 5.1: Commit `d618651` - thinkpad_acpi: Don't yell on unsupported brightness interfaces

To understand the nature of afterthought changes made to logging code, we conduct a qualitative analysis in which we manually examine the corresponding fixes provided by the Linux kernel developers. The ultimate goal is to gain deep insight into the reasons behind these changes, which we hope can help developers and researchers design new approaches and tools to prevent these problems in the first place. We detail

the data extraction process in Section 5.3.1 and present our results in Section 5.3.2.

5.3.1 Data Gathering and Extraction

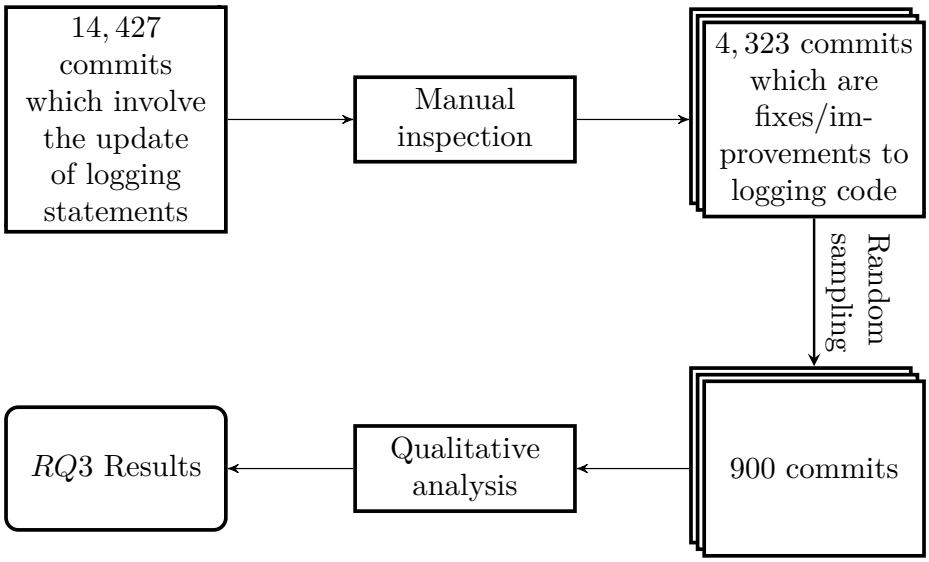


Figure 5.6: RQ3 data collection process

Considering the 14,427 commits used for answering RQ2, not all aim to provide fixes or improvements to logging code, which is the type of change in logging code that we are interested in to answer RQ3. To identify those that are afterthought log changes, previous studies [Has+18; Maz+20], rely on a keyword-based approach by searching for commits that contain in their message variations of the word "log". This strategy may miss many commits in the Linux kernel such as the one described above.

In this study, the author of the thesis manually examined all 14,427 commits by reading the commit titles and messages if necessary. He identified that 4,323 commits out of 14,427 commits that explicitly discuss changes to logging statements. This task took several weeks to complete and required multiple iterations. He then selected randomly 900 commits out 4,323 for the qualitative analysis. This subset is statistically representative with a confidence level of 99% and a margin of error of ±4% [Bos12].

5.3.2 Data Analysis

To analyze the resulting 900 commits, for each commit, the author reviewed the commit message, the commit diff, related artefacts such as bug reports, and discussions on the Linux kernel mailing lists, if available. He classified the reasons behind these afterthought changes into 13 categories, which were reviewed and validated by two more researchers. These categories are shown in Table 5.9 and are discussed in more detail in the subsequent sections. We also provide recommendations to prevent the type of problems depicted in these categories.

Table 5.9: Characterization of fixes to the problematic logging code

Reason for the change	#Commits
Improving debuggability	92
Reword ambiguous/misleading logging message	62
NULL pointer dereference	13
Spelling/Grammar mistakes	171
Fix incorrect loglevel	156
Formatting issues	71
Modernize logging code	163
Fix early logging	9
Copy-Paste mistakes	6
Logging wrong information	28
Revealing kernel pointers	16
Remove redundant information	31
Fix format specifier	125
Total	900

5.3.2.1 Improving debuggability

We found that 10.22% of the studied commits are about improving debuggability by adding information to logging code with the aim of reducing the time needed to diagnose program failures. This is in line with the finding of Yuan et al. [YPZ12], where the authors stated that developers often add information to the existing logging statements to narrow down the root causes of the underlying problems. For example,

in commit `d0de579`, the developer mentioned: *"Identify Namespace failures are logged as a warning but there is not an indication of the cause for the failure. Update the log message to include the error status"*. Another example is commit `077c066`, which added a local variable `idx`, representing a section index, to the logging statement as shown in Figure 5.7. The reason for this change is given by the developer as: *"While debugging a bpf ELF loading issue, I needed to correlate the ELF section number with the failed relocation section reference. Thus, add section numbers/index to the pr_debug."*.

We also found many cases where developers added logging statements to record additional runtime information. One example of this can be seen in commit `9ef8690`, where the developer inserted few additional logging points in order to make errors more visible: *"The NCSI driver is mostly silent which becomes a headache when trying to determine what has occurred on the NCSI connection. This adds additional logging in a few key areas such as ..."*.

> **Recommendation:** To prevent these changes, we recommend that developers provide beforehand detailed information on where the failures are located and any other information relevant to failure that can facilitate debugging. Including error code in the logging statement, if available, is always a good idea.

tools/libbpf: improve the pr_debug statements to contain section numbers | linux@077c066

```
pr_warning("failed to alloc name for prog under section %s\n", section_name);
```

```
pr_warning("failed to alloc name for prog under section(%d) %s\n", idx,
    ↪ section_name);
```

Figure 5.7: Example of adding information to a log message to improve debuggability

5.3.2.2 Reword ambiguous/misleading logging messages

Ambiguity in an error message can delay the process of diagnosis as it does not allow end-users to easily uncover the part of the programs that failed. One example of this is shown in Figure 5.8. Here, the developer chose to use the full register name in the error message because a short version of the register name may be ambiguous when diagnosing a fault. A similar example can be found in commit `2e5d04dad` where the *iwlagn* driver uses exactly the same error message in three different functions. Therefore, the developer chose to add the name of the function to the error string to disambiguate from where the error originated.

ASoC: tas6424: Print full register name in error message | linux@919869214

```
dev_err(dev, "failed to read FAULT1 register: %d\n", ret);
```

```
dev_err(dev, "failed to read GLOB_FAULT1 register: %d\n", ret);
```

Figure 5.8: Example of an ambiguity in a log error message

Another common problem is the ambiguity of a log message, which can be very misleading during the analysis phase. Figure 5.9 shows an example where the earlier log message did not consider the fact that while we were unable to use DMA, we could still complete the transfer by PIO mode. The log message `"failed to start DMA"` is being printed with the error log level, which might mislead users to think that a fatal error occurred.

Developers often reword the logging message to make the logs more informative and facilitate analysis. An example from this category is shown in Figure 5.10, where developers decided to reword NULL pointer dereference message. The logging message was modified to drop *"unable to handle"* from the message. As it might imply that in some cases the kernel actually handles NULL pointer dereference, which is not valid. A similar example is found in commit `135e535`, where the developer

mmc: dw_mmc: fix misleading error print if failing to do DMA transfer | linux@d12d0cb

```
/* We can't do DMA */
dev_err(host->dev, "%s: failed to start DMA.\n", __func__);
```

```
/* We can't do DMA, try PIO for this one */
dev_dbg(host->dev, "%s: fall back to PIO mode for current transfer\n",
    ↪ __func__);
```

Figure 5.9: Example of an error message which may mislead end-users

clarified an error message to avoid user confusion. He reported that: *"Some user who install SIGBUS handler that does longjmp out therefore keeping the process alive is confused by the error message* `[188988.765862] Memory failure: 0x1840200 : Killing cellsrv:33395 due to hardware memory corruption` *Slightly modify the error message to improve clarity."* In conclusion, poorly worded log messages may lead to user confusion and fixing these logs takes up maintenance time and effort.

Another issue that may have contributed to ambiguity is logging statements with the same static text in a single file. This practice of duplication in logging code was rightfully reported by Li et al. [Li+19b] as a logging *code smell*. One example of this practice is depicted in commit `a15e824` and commit `90cc7f1`, where the developer added additional information so that log messages can be uniquely identified using search techniques.

Recommendation: To avoid this type of log-related changes, we recommend putting in place a review process, just like for reviewing other parts of the code. This task can be done before accepting the commits. Including the review of log statements during the code review phases is also another possibility.

x86/fault: Reword initial BUG message for unhandled page faults | linux@f28b11a

```
pr_alert("BUG: unable to handle kernel %s at %px\n",
    address < PAGE_SIZE ? "NULL pointer dereference" : "paging request", (void
        ↪ *)address);
```

```
if (address < PAGE_SIZE && !user_mode(regs))
    pr_alert("BUG: kernel NULL pointer dereference, address = %px\n", (void *)
        ↪ address);
else
    pr_alert("BUG: unable to handle page fault for address = %px\n", (void *)
        ↪ address);
```

Figure 5.10: Example of clarifying an error message to avoid user confusion

5.3.2.3 NULL pointer dereference

We found 1.44% cases where a developer attempted to dereference a pointer which may have a NULL value or a variable which may not be initialized. A NULL pointer dereferencing causes a runtime crash. Figure 5.11 shows an example. This fix occurred in commit `95d2a32` and the message reads as follow: *"A null pointer dereference will occur when skb is null and* `skb->dev->name` *is printed. Replace the* `skb->dev->name` *with plain text ks_wlan to fix this."*. To prevent this type of problems, developers should incorporate tools such as Coccinelle [3] in their workflow for detecting dereferences of NULL pointers.

> **Recommendation:** A potential NULL pointer dereferencing can be avoided by adding automatic checks to logged variables, and function return values that are used as logging statements arguments.

5.3.2.4 Spelling/Grammar mistakes

19% of the 900 studied log updates are caused by spelling or grammar mistakes. Figure 5.12 shows an example where the word "synchronously" was misspelt as

[3] http://coccinelle.lip6.fr/rules/#null

staging: ks7010: don't print `skb->dev->name` if skb is null | linux@95d2a32

```
printk(KERN_WARNING "%s: Memory squeeze, dropping packet.\n",
      skb->dev->name);
```

```
printk(KERN_WARNING "ks_wlan: Memory squeeze, dropping packet.\n");
```

Figure 5.11: Example of NULL pointer dereference in a logging statement

"synchronuously" in the static text. Another example can be seen in commit `748ac56`. To correct a grammatical mistake, the static text was changed from `"Failed to registered ssb SPROM handler"` to `"Failed to register ssb SPROM handler"`. We also noticed that there is a lack of any kind of standardization over the use of capitalization, grammatical style, punctuation, etc.

> **Recommendation:** It is recommended to use a spell checker, as provided by many IDEs. This is an initial step to prevent the need for this kind of corrections. Additional tools for grammar checking could be embedded in current IDEs. Guidelines for a common writing style should be developed and promoted. Tools such as `kernelscan` should be incorporated into build pipelines. [4]

usbip: vhci: fix spelling mistake: "synchronuously" → "synchronously" | linux@cb48326

```
dev_dbg(&urb->dev->dev, "urb seq# %u was unlinked %ssynchronuously\n", seqnum
    ↪ , status == -ENOENT ? "" : "a");
```

```
dev_dbg(&urb->dev->dev, "urb seq# %u was unlinked %ssynchronously\n", seqnum,
    ↪  status == -ENOENT ? "" : "a");
```

Figure 5.12: Example of a simple spelling mistake in a debugging message

[4] https://github.com/ColinIanKing/kernelscan

5.3.2.5 Fix incorrect log levels

We found 17.33% cases where developers failed to make a distinction between fatal errors and errors that are recoverable, which have led to changes to log levels. Using a non-error log level for logging error conditions would make it hard to diagnose such errors as the corresponding error messages would go unnoticed. Likewise, logging debugging messages as errors will also result in a flood of log messages making it hard to concentrate on the real problems.

i2c: imx: notify about real errors on dma `i2c_imx_dma_request` | linux@5b3a23a

```
dev_dbg(dev, "can't configure rx channel\n");
```

```
dev_err(dev, "can't configure rx channel (%d)\n", ret);
```

Figure 5.13: Example of increasing the severity of a logging statement to enhance visibility

One such example is shown in Figure 5.13, where the developer mentioned that *"... In contrast real problems that were only emitted at debug level before should be described at a higher level to be better visible and so understandable."*. Thus, the developer decided to change the log level from "debug" to "error".

Developers need to carry out logging activities taking into account performance constraints [Din+15; Sig+10]. One of the most frequent issues is log spamming, which often leads to the degradation of system performance. One example of this category can be seen in commit `7f20d83`, where the developer downgraded the log message to "debug" level to suppress frequent *"VTU miss violations"* messages, as *"VTU miss violations"* are rather common.

Recommendation: Even though the existing description of log levels in the Linux kernel provides guidance on how they should be used, a suggestion is to detail it even more. For instance, the error level should only be used in situations in which components may fail and compromise system operation. There is clearly a need for more detailed guidelines on how to use log levels.

5.3.2.6 Fix format specifiers

We found 13.89% of the 900 studied revisions are the result of using improper format specifiers in logging statements. [5] A typical issue related to `printk` format specifier is shown in Figure 5.14. Here, the developer decided to use the `%pS printk` format specifier for printing symbols from direct addresses. Moreover, the explanation for this change was given as *"This is important for the ia64, ppc64 and parisc64 architectures, while on other architectures there is no difference between %pS and %pF. Fix it for consistency across the kernel."*. It appears that the majority of this type of changes are motivated by the intention to fix build warnings, which developers should have addressed at the commit-time. We found that developers often get around using proper format specifier by resorting to unnecessary casts. [6] This practice was rightfully reported by Chen et al. [CJ17a] as a logging *anti-pattern*.

ti_sci: Use %pS printk format for direct addresses | linux@595f3a9

```
dev_err(dev, "Mbox timedout in resp(caller: %pF)\n", (void *)_RET_IP_);
```

```
dev_err(dev, "Mbox timedout in resp(caller: %pS)\n", (void *)_RET_IP_);
```

Figure 5.14: Example of change to format specifier for consistency with other parts of the Linux kernel

[5] https://www.kernel.org/doc/Documentation/printk-formats.txt
[6] https://github.com/torvalds/linux/commit/3fcb3c836ef413d3fc848288b308eb655e08d853

Recommendation: There exist static analysis tools such as `smatch` [7] and `sparse` [8] that are useful at detecting incorrect format specifiers. Developers should consider using these tools to detect this type of problems at commit-time rather than fixing the corresponding code as afterthought.

5.3.2.7 Modernize logging code

We found 163 commits (18.11%) that were the result of ongoing modernization of logging code. Usually, such improvements do not address severe problems associated with the logging code; instead, they arise from attempts to improve the consistency of logging code across various parts of the Linux kernel. One such change is the use of device-managed logging macros in order to simplify error handling, reduce source code size, improve readability, and/or reduce the risks of bugs. Another improvement is the use of `__func__` and `pr_fmt` rather than hard-coding function names and module names in error messages. For example, in commit `a8ab042`, the developer mentioned: *"Instead of having the function name hard-coded (it might change and we forgot to update them in the debug output) we can use* `__func__` *instead and also shorter the line so we do not need to break it."*. As Yuan et al. [YPZ12] pointed out, inconsistency in the function names referred in the log message is one of the main reasons for behind changes made to logging code.

Recommendation: Enforcement of simple practices such as no hard-coded module names or function names in the error message at the commit-time could go a long way.

5.3.2.8 Fix early logging

Out of 900 commits, we found 9 cases in which log messages refer to devices before they are registered. These logs contain messages referring to `"(unnamed net device)`

[7] https://repo.or.cz/w/smatch.git
[8] https://sparse.wiki.kernel.org/

`(uninitialized)"`, which lead to logs that may confuse end-users. Figure 5.15 shows a fix to one of these problems.

> **Recommendation:** Developers should only use the device or network-specific logging macros after checking that the devices were correctly initialized and registered. Besides, static analysis tools can be used to check that all variables are initialized before they are used for logging.

staging: fsl-dpaa2/eth: Don't use netdev_err too early | linux@0f4c295

```
netdev_err(net_dev, "Failed to configure hashing\n");
```

```
dev_err(dev, "Failed to configure hashing\n");
```

Figure 5.15: Example of a simple early logging mistake

5.3.2.9 Copy-Paste mistakes

We noticed six instances where developers were trying to save time by copying and pasting a few snippets of code; however, they forgot to make the necessary modifications. One such example is shown in Figure 5.16. In the `gpbridge_init` method, the developer entered an incorrect logging message, caused by a simple copy/paste mistake. In this case, it seems that the developer copied line `if (gb_usb_protocol_init())` and corrected it. However, the debugging statements were not updated accordingly.

> **Recommendation:** Developers should avoid copying and pasting logging statements. If they do not match the targeted subject, they may confuse and mislead developers when debugging and analyzing logged messages.

greybus: gpb: Fix print mistakes | linux@b908dec

```
if (gb_usb_protocol_init()) {
       pr_err("error initializing usb protocol\n");
       goto error_usb;
}
if (gb_i2c_protocol_init()) {
       pr_err("error initializing usb protocol\n");
       goto error_i2c;
}
```

```
if (gb_usb_protocol_init()) {
       pr_err("error initializing usb protocol\n");
       goto error_usb;
}
if (gb_i2c_protocol_init()) {
       pr_err("error initializing i2c protocol\n");
       goto error_i2c;
}
```

Figure 5.16: Example of incorrect logging message due to a simple copy/paste oversight

5.3.2.10 Logging wrong information

We found 28 cases where developers specified the wrong variables as arguments of the logging function calls. The root causes of these issues are simple copy/paste mistakes or typographical errors [9]. One example of this shown in Figure 5.17. In this case the developer first checks the condition `btrfs_dir_name_len(leaf, dir_item) > namelen` and if it is *true*, report the `name_len` of `dir_item` in the error message. By mistake, however, the developer ended up reporting `data_len` instead of `name_len`. Such errors are difficult to detect using static analysis tools, since both `btrfs_dir_name_len` and `btrfs_dir_data_len` have the same return type.

> **Recommendation:** Developers should review the logging statement just like any other part of the code.

[9] https://cwe.mitre.org/data/definitions/688.html

btrfs: tree-log.c: Wrong printk information about namelen
| linux@286b92f

```
btrfs_crit(fs_info, "invalid dir item name len: %u", (unsigned)
    ↪ btrfs_dir_data_len(leaf, dir_item));
```

```
btrfs_crit(fs_info, "invalid dir item name len: %u", (unsigned)
    ↪ btrfs_dir_name_len(leaf, dir_item));
```

Figure 5.17: Example of reporting wrong information in the logging statement

5.3.2.11 Revealing kernel pointers

Missing key/vital information from log messages can delay the diagnosis process. However, revealing sensitive information such as cryptographic keys or kernel addresses can lead to information leaks [10]. We found 16 commits, which mentioned these cases. On example can be seen in commit `9cdf0ed` where the commit message reads as *"Printing raw kernel pointers might reveal information which sometimes we try to hide (e.g. with Kernel Address Space Layout Randomization). Use the "%pK" format so these pointers will be hidden for unprivileged users."*. Other examples of such cases can be seen at CVE-2018-5995 [11] and CVE-2018-7273 [12], where developers print kernel addresses into logs which can allow an attacker to extract sensitive information. The problem of accidental data leakage through the misuse of logs has recently been examined by Zhou et al. [Zho+20]. The authors showed that logs could reveal sensitive information in Android apps. Similar studies should be conducted for larger systems such as the Linux kernel to understand the extent of this serious problem. Research tools such as KALD [Bel+19] should be included into Linux kernel development pipeline.

[10] https://cwe.mitre.org/data/definitions/200.html
[11] https://nvd.nist.gov/vuln/detail/CVE-2018-5995
[12] https://nvd.nist.gov/vuln/detail/CVE-2018-7273

Recommendation: Tests should be put in place to verify that logs do not accidentally cause security and data privacy breaches. This effort should adhere to the broader task of ensuring the security of the Linux kernel.

drm/exynos: Print kernel pointers in a restricted form | linux@9cdf0ed

```
dev_dbg(dev, "< xfer %p: tx len %u, done %u, rx len %u, done %u\n", xfer,
    ↪ length, xfer->tx_done, xfer->rx_len, xfer->rx_done);
```

```
dev_dbg(dev, "< xfer %pK: tx len %u, done %u, rx len %u, done %u\n", xfer,
    ↪ length, xfer->tx_done, xfer->rx_len, xfer->rx_done);
```

Figure 5.18: Example of revealing raw kernel pointers

5.3.2.12 Remove redundant information

We found 31 cases in which developers report information that is not needed or is redundant. One illustration of this case found in commit `2bcfdc2` is shown in Figure 5.19 where the developer removed *uuid* from the debug messages in `bus-fixup.c` as this was already part of the device name. Another common pattern that we observed is the removal of `__func__` from `dev_dbg()` calls. The reason for this change is given in commit `b814735` as *"Dynamic debug can be instructed to add the function name to the debug output using the +f switch, so there is no need for the nfit module to do it again. If a user decides to add the +f switch for nfit's dynamic debug this results in double prints of the function name ..... Thus remove the stray* `__func__` *printing."*. In addition, removing `__func__` from `dev_*()` callsites helps reduce the Linux kernel size as pointed out by Wolfram Sang, the current maintainer of the Linux I2C subsystem. [13]

[13] https://git.kernel.org/pub/scm/linux/kernel/git/wsa/linux.git/commit/?h=strings/rtc-no-func&id=762c5af234c5b816b7da3687a3e703cf8cdc2214

Recommendation: Developers should strive for writing concise logging statement to prevent redundant information. Automatic tools can be developed to detect redundancies. However, some redundancies may require domain knowledge. These can be detected by developers when reviewing the code.

mei: bus: remove redundant uuid string in debug messages | linux@2bcfdc2

```
dev_dbg(&cldev->dev, "running hook %s on %pUl\n", __func__, mei_me_cl_uuid(
    ↪ cldev->me_cl));
```

```
dev_dbg(&cldev->dev, "running hook %s\n", __func__);
```

Figure 5.19: Example of reporting redundant information

5.3.2.13 Formatting issues

We found 71 cases where developers fix the formatting of log messages. Poorly formatted messages make it difficult to search for matching text in the log file. We can see this issue in commit `a790634` where certain messages were appearing on separate lines resulting in a strange output. This was fixed using line continuations where necessary. Another issue that falls under this group is that developers frequently break log message strings over several lines to meet the checkpatch's 80 characters per line restriction. According to commit `4bd69e7b`, this is no longer considered a good practice, because it makes it more difficult to `grep` for strings at the source code.

Recommendation: Developers should avoid breaking log message lines to facilitate post-mortem analysis of the logs.

Chapter 6

Discussions and Threats to Validity

In Section 6.1, we discuss our findings and implications. In Section 6.2, we discuss threats to the internal and external validity of our work.

6.1 Discussions

Logging is pervasive in the Linux kernel, as 3.73% of the Linux kernel source is logging code. However, when we evaluate the pervasiveness of logging as file and method level, we find that distribution of logging code is very skewed, suggesting that certain aspects of the Linux kernel are more likely to have logging code than others. It would be interesting to assess the correlation between log ratio of a method and cyclomatic complexity of a method in future studies. We also find that many old drivers and file-systems have custom logging macros [1] defined to trace function entry/exit. Even though tools like `ftrace` can be used to trace Linux kernel function calls, use of logging statements for function tracing is still prevalent.

In the Linux kernel, logging code is actively maintained. Although only 3.73% of the Linux kernel source is logging code, we find that 14% of commits related to Linux kernel versions 4.3 to 5.3 involves modifications to logging code resulting

[1] https://github.com/torvalds/linux/blob/v5.3/fs/afs/internal.h#L1449

in a total of 211,437 logging statements modifications, which represents 66.19% of the total number of logging statements present in Linux kernel v5.3. Moreover, we find that logging code deletion accounts for 29.23% of total modifications to logging code, in contrary to findings reported by Yuan et al. [YPZ12]. Although the majority of logging code deletion happens as the file is deleted, there exists non-negligible amount (44.66%) of deletion of logging code occurring as afterthoughts. A possible explanation for this might be observed in commit `a8d5dad` where developer deleted two logging statements which were put there to report memory allocation failures. There exist a rule in `checkpatch.pl` to check for possible unnecessary *out of memory* message; however; it seems that developers do not care to fix this issue at the commit-time, provided that a large number of commits removing *out of memory* error message as afterthoughts. Another scenario observed for logging code deletion is removing logging statements which are redundant. For instance, in commit `c99a23e55` where developer removed an error message present inside error handling code when `i2c_mux_add_adapter` fails, because `i2c_mux_add_adapter` will print an error message itself on failure.

We found that developers often face difficulties specifying the right log level in the first try, as it is difficult to statically test whether a given log level is correct or not. If we look at commit `3b364c659`, developer downgraded the logging message to `info` log level from `warn`, and reason for this change was mentioned as: *"On an embedded system it is quite possible for the bootloader to avoid configuring PCIe devices if they are not needed."*. It is surprising that it took the developers two years and two months to notice this problem.

In Section 5.1.2, we find that 55.66% of the logging statements are used inside the *if* block, most of which is used for logging errors after checking the return value of function calls. As this decision is left to developers, we noticed many inconsistencies in the text of an error message, logging function used, and information included in an error message. Table 6.1 shows one such example. A call to

`thermal_zone_device_register()` returns a pointer to the newly created struct `thermal_zone_device`, and in case of error returns an `ERR_PTR`. [2] All these three drivers performs a registration of a new thermal zone device and check the return value using `IS_ERR()` macros, and in case of an error logs an error using `dev_err()`. Even though all three drivers are performing the same action, we can see that there is no similarity between error messages.

We also notice inconsistencies in the information included when logging an error message. For example, when reporting an error when the call to `devm_request_irq` fails, at many places caller of `devm_request_irq` does not even include the irq requested and an error code returned. [3] A possible solution to avoid such inconsistencies would be centralizing error reporting rather than leaving the decision to the developers. One such change made to get more consistent error reporting can be seen in commit `7723f4c`, the reason for this change was mentioned as *"A grep of the kernel shows that many drivers print an error message if they fail to get the irq they're looking for. Furthermore, those drivers all decide to print the device name, or not, and the irq they were requesting, or not, etc. Let's consolidate all these error messages into the API itself, allowing us to get rid of the error messages in each driver."*. Such centralization of error reporting helps in reducing the size of the Linux kernel. This also reduces the number of commits made as afterthoughts to add additional information or fixing typographical mistakes.

6.2 Threats to Validity

6.2.1 Internal Validity

Threats to internal validity are associated with factors that may impact our results. In this study, a source of bias is an automated data collection process. To identify logging

[2] https://github.com/torvalds/linux/blob/v5.3/drivers/thermal/thermal_core.c#L1211

[3] https://github.com/torvalds/linux/blob/v5.3/drivers/usb/dwc2/gadget.c#L4846

Table 6.1: Lack of consistency in the text of the error messages

```
spear_thermal = thermal_zone_device_register(...);
if (IS_ERR(spear_thermal)) {
    dev_err(&pdev->dev, "thermal zone device is NULL\n");
    [...]
}
```

drivers/thermal/spear_thermal.c

```
priv->zone = thermal_zone_device_register(...);
if (IS_ERR(priv->zone)) {
    dev_err(dev, "can't register thermal zone\n");
    [...]
}
```

drivers/thermal/rcar_thermal.c

```
sensor->thermal_dev = thermal_zone_device_register(...);
if (IS_ERR(sensor->thermal_dev)) {
    dev_err(dev, "failed to register thermal zone device\n");
    [...]
}
```

drivers/thermal/st/st_thermal.c

statements, we rely on the semantic patterns specified by [SLM15]. However, this approach may not identify statements that lack variability in their use. To mitigate this issue, we manually examined macros containing calls to basic functions such as `printk()`, `pr_*()`, and `dev_*()` in order to identify missed logging functions. The set of functions collected by combining both processes was thus manually reviewed with the aim of eliminating obvious false positives. While evaluating the pervasiveness of logging statements at program constructs level, we were unable to assign few logging statements to any constructs due to the limitation of the static analysis approaches.

In order to automatically identify and classify changes made to logging code, we use a pipeline comprised of GumTree [Fal+14] and a script specifically developed for this study. To mitigate possible misclassifications, we manually inspected a sample of 100 randomly selected modifications, which showed an accuracy of 98% on classified modifications.

Another threat to this study is the manual classification of log-related commits. We manually examined all commits by their title and message whenever necessary. However, we can not eliminate the possibility that errors may have occurred during this manual analysis. We thus do not claim that our dataset is complete. This threat is mitigated by the fact that our goal was to collectively study the nature of the problematic logging code rather than collecting every possible revision made to fix or improve logging code.

Because we perform a qualitative analysis in order to categorize changes made to logging code and the nature of problematic logging code, researcher bias also becomes a threat to the internal validity of our study. In this kind of analysis, results are associated with researcher interpretation of the data. Therefore, to mitigate it, every classification that raised questions were discussed by the researchers until they reached an agreement.

6.2.2 External Validity

Threats to external validity are related to which extend our results can be generalized. In fact, in this work, we focused solely on the Linux kernel project, which is known for having its own development culture. Nevertheless, it is a large scale, the open-source project maintained by developers from different companies, which constitutes a representative sample of C/C++ projects.

Chapter 7

Conclusion and Future Work

7.1 Summary of the Findings

In this thesis, we presented an empirical study on the practice of logging in the Linux kernel. We found that there is one line of logging code in every 27 lines of the source code in the Linux kernel v5.3, which represents 3.73% of the total source code. We also found that logging density varies across the Linux kernel subsystems and components with no apparent reason. This could be due to the fact that each component is maintained by different development teams and that there are no common logging guidelines for Linux developers.

We also found that 72.36% of the total number of files in the Linux kernel have at least one logging statement, while only 26.12% of functions are logged. The program constructs that are logged the most are if-blocks (including else-if blocks) with 57% of the total number of logging statements. These statements are mainly used for logging information when the function call fails. This is equivalent to logging the try/catch block in other programming languages such as C++ and Java.

After studying 22 releases of the Linux kernel (from versions v4.3 to v5.3), we found that the use of logging code has been declining. We attributed this to the increase in debugging and tracing tools. We also found that out of 211,437 logging

statements modifications that occurred between versions v4.3 and v5.3, 24.78% are log updates, 45.99% are log insertions, and 29.23% are log deletions.

By manually investigating 900 commits that aim to fix or improve logging code, we found that the majority of changes are to fix spelling/grammar mistakes, fix incorrect log levels, and upgrade logging code to use new logging macros to improve the precision and consistency of log output.

Many of these changes could be avoided if Linux kernel developers use static analysis tools to detect potential null references in logging statements, spell/grammar checkers, basic writing styles for log messages. We also recommend that the Linux kernel developers organize systematic review sessions to review the quality of logging statements. Finally, we strongly recommend the development of guidelines to standardize the logging practice across the Linux kernel development teams.

7.2 Future Directions

An important future direction is to conduct a qualitative study to understand the rationale behind logging. This can be done by selecting one or two Linux kernel components and examine the various purposes for which logging is used. This study should not be limited to the analysis of the source code of these components but must include a user study, which involves the development teams. We believe that insight obtained from developers will improve significantly our understanding of how logging is used in practice.

In addition, we need to work towards developing guidelines and standards for logging. To do so, we should start by putting in place a catalogue of good and bad logging practices. This work can lead to the development of tools that can be embedded in software development toolkits to enforce good logging practices. We also believe that this work can lead to tools and processes for the detection of logging smells (similar to code smells), logging patterns, and logging anti-patterns.

Furthermore, while doing this study, we observed that there are not many studies that combine logging with tracing. By tracing, we mean tracing the program control flow (e.g., tracing function calls [HL02]). In our research lab, we have developed many trace analysis and modeling techniques that we believe can be readily used for large streams of log data including techniques for trace abstraction and summarization [HL06; Pir+13], trace modeling and metamodeling [Hoj+20; HL12], and trace analysis tools [HL04]. We, therefore, recommend (1) extending these techniques to support the analysis of log data, and (2) start working toward holistic approaches that combine logging and tracing in one robust dynamic analysis framework.

7.3 Closing Remarks

Despite the many benefits of log data, the practice of logging is still ad-hoc and without recognized guidelines. This study examines in depth the practice of logging in the Linux kernel. By doing so, we hope that this study contributes to the corpus of knowledge on the practice of logging in large systems, while highlighting the challenges that developers face when performing logging activities in software engineering. The long-term goal is to help develop standards and common guidelines for logging, as well as better processes, techniques, and tools.

Bibliography

[Bag+18] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E Hassan, Juergen Dingel, and James R Cordy. "Analyzing a decade of Linux system calls". In: *Empirical Software Engineering* 23.3 (2018), pp. 1519–1551.

[Bel+19] Brian Belleville, Wenbo Shen, Stijn Volckaert, Ahmed M Azab, and Michael Franz. "KALD: Detecting Direct Pointer Disclosure Vulnerabilities". In: *IEEE Transactions on Dependable and Secure Computing* (2019).

[Ber+17] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. "Experience report: Log mining using natural language processing and application to anomaly detection". In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2017, pp. 351–360.

[Boo20] Bootlin. *Embedded Linux kernel and driver development training*. CreateSpace Independent Publishing Platform, 2020. URL: https://github.com/bootlin/training-materials.

[Bos12] Sarah Boslaugh. *Statistics in a nutshell: A desktop quick reference*. " O'Reilly Media, Inc.", 2012.

[CJ17a] Boyuan Chen and Zhen Ming Jiang. “Characterizing and detecting anti-patterns in the logging code”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 71–81.

[CJ17b] Boyuan Chen and Zhen Ming Jack Jiang. “Characterizing logging practices in Java-based open source software projects–a replication study in Apache Software Foundation”. In: *Empirical Software Engineering* 22.1 (2017), pp. 330–374.

[CJ19] Boyuan Chen and Zhen Ming Jack Jiang. “Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems”. In: *Empirical Software Engineering* 24.4 (2019), pp. 2285–2322.

[Cor12] Jonathan Corbet. “The perils of pr_info()”. In: *LWN. net* (2012).

[Cor16] Jonathan Corbet. “Tracepoint challenges”. In: *LWN. net* (2016).

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware.* ” O’Reilly Media, Inc.”, 2005.

[Din+15] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. “Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis”. In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’15. Santa Clara, CA: USENIX Association, 2015, pp. 139–150. ISBN: 9781931971225.

[Edg19] Jake Edge. “Unifying kernel tracing”. In: *LWN. net* (2019).

[El-+20] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. “A systematic literature review on automated log abstraction techniques”. In: *Information and Software Technology* 122 (2020), p. 106276.

[Fal+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. "Fine-grained and accurate source code differencing". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, pp. 313–324. DOI: 10.1145/2642937.2642982. URL: http://doi.acm.org/10.1145/2642937.2642982.

[Fu+14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. "Where do developers log? an empirical study on logging practices in industry". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 24–33.

[Has+18] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. "Studying and detecting log-related issues". In: *Empirical Software Engineering* 23.6 (2018), pp. 3248–3280.

[He+18] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. "Characterizing the Natural Language Descriptions in Software Logging Statements". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 178–189. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238193. URL: http://doi.acm.org/10.1145/3238147.3238193.

[HL02] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. "Compression techniques to simplify the analysis of large execution traces". In: *Proceedings 10th International Workshop on Program Comprehension*. IEEE. 2002, pp. 159–168.

[HL04] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. "A survey of trace exploration tools and techniques". In: *Proceedings of the 2004 conference*

of the Centre for Advanced Studies on Collaborative research. 2004, pp. 42–55.

[HL06] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system". In: *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE. 2006, pp. 181–190.

[HL12] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. "A metamodel for the compact but lossless exchange of execution traces". In: *Software & Systems Modeling* 11.1 (2012), pp. 77–98.

[Hoj+20] Fazilat Hojaji, Bahman Zamani, Abdelwahab Hamou-Lhadj, Tanja Mayerhofer, and Erwan Bousse. "Lossless compaction of model execution traces". In: *Software and Systems Modeling* 19.1 (2020), pp. 199–230.

[IF10] Ayelet Israeli and Dror G Feitelson. "The Linux kernel as a case study in software evolution". In: *Journal of Systems and Software* 83.3 (2010), pp. 485–501.

[IKH18] Md Shariful Islam, Wael Khreich, and Abdelwahab Hamou-Lhadj. "Anomaly detection techniques based on kappa-pruned ensembles". In: *IEEE Transactions on Reliability* 67.1 (2018), pp. 212–229.

[KG11] Kamal Kc and Xiaohui Gu. "ELT: Efficient log-based troubleshooting system for cloud computing infrastructures". In: *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. IEEE. 2011, pp. 11–20.

[Kha+18] Subhendu Khatuya, Niloy Ganguly, Jayanta Basak, Madhumita Bharde, and Bivas Mitra. "Adele: Anomaly detection from event log empiricism". In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 2114–2122.

[Li+18] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. "Studying software logging using topic models". In: *Empirical Software Engineering* 23.5 (2018), pp. 2655–2694. DOI: 10.1007/s10664-018-9595-8. URL: https://doi.org/10.1007/s10664-018-9595-8.

[Li+19a] Shanshan Li, Xu Niu, Zhouyang Jia, Xiangke Liao, Ji Wang, and Tao Li. "Guiding log revisions by learning from software evolution history". In: *Empirical Software Engineering* (2019), pp. 1–39.

[Li+19b] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. "DLFinder: Characterizing and detecting duplicate logging code smells". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 152–163.

[Li+20] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. "A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives". In: *IEEE Transactions on Software Engineering* (2020).

[Liu+19] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. "Which Variables Should I Log?" In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[Lot+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "Evolution of the linux kernel variability model". In: *International Conference on Software Product Lines*. Springer. 2010, pp. 136–150.

[LSH17] Heng Li, Weiyi Shang, and Ahmed E. Hassan. "Which log level should developers choose for a new logging statement?" In: *Empirical Software Engineering* 22.4 (Aug. 2017), pp. 1684–1716. ISSN: 1573-7616. DOI: 10.1007/s10664-016-9456-2. URL: https://doi.org/10.1007/s10664-016-9456-2.

[LSS15] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. “Two level empirical study of logging statements in open source Java projects”. In: *International Journal of Open Source Software and Processes (IJOSSP)* 6.1 (2015), pp. 49–73.

[Lu+14] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. “A Study of Linux File System Evolution”. en. In: *ACM Transactions on Storage* 10.1 (Jan. 2014), pp. 1–32. ISSN: 1553-3077, 1553-3093. DOI: 10.1145/2560012. URL: https://dl.acm.org/doi/10.1145/2560012 (visited on 04/06/2020).

[Maz+20] Alejandro Mazuera-Rozo, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. “Investigating types and survivability of performance bugs in mobile apps”. In: *Empirical Software Engineering* (2020), pp. 1–43.

[Mir+16] Andriy Miranskyy, Abdelwahab Hamou-Lhadj, Enzo Cialini, and Alf Larsson. “Operational-log analysis for big data systems: Challenges and solutions”. In: *IEEE Software* 33.2 (2016), pp. 52–59.

[OAS08] Adam J Oliner, Alex Aiken, and Jon Stearley. “Alert detection in system logs”. In: *2008 Eighth IEEE International Conference on Data Mining*. IEEE. 2008, pp. 959–964.

[Pan+20] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. *Learning to Update Natural Language Comments Based on Code Changes*. 2020. arXiv: 2004.12169 [cs.CL].

[PCW12] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. “Towards a catalog of variability evolution patterns: the Linux kernel case”. en. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development - FOSD ’12*. Dresden, Germany: ACM Press, 2012, pp. 62–69. ISBN: 9781450313094. DOI: 10.1145/2377816.2377825. URL:

http://dl.acm.org/citation.cfm?doid=2377816.2377825 (visited on 04/06/2020).

[PCZ18] Aidi Pi, Wei Chen, and Xiaobo Zhou. "Profiling Distributed Systems in Lightweight Virtualized Environments with Logs and Resource Metrics". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '18. Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 9–10. ISBN: 9781450358996. DOI: 10.1145/3220192.3220197. URL: https://doi.org/10.1145/3220192.3220197.

[Pec+15] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. "Industry practices and event logging: Assessment of a critical software development process". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 169–178.

[Pir+13] Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, Luay Alawneh, and Arya Shafiee. "Stratified sampling of execution traces: Execution phases serving as strata". In: *Science of Computer Programming* 78.8 (2013), pp. 1099–1118.

[Ran19] Chen An Ran. "Studying and Leveraging User-Provided Logs in Bug Reports for Debugging Assistance". 2019. URL: https://spectrum.library.concordia.ca/985950/.

[Sha+14] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. "An exploratory study of the evolution of communicated information about the execution of large software systems". In: *Journal of Software: Evolution and Process* 26.1 (2014), pp. 3–26. DOI: 10.1002/smr.1579. eprint: https://

onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1579. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1579.

[Sig+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[SLM15] Peter Senna Tschudin, Julia Lawall, and Gilles Muller. "3L: Learning Linux Logging". In: *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2015)*. Lille, France, Dec. 2015. URL: https://hal.inria.fr/hal-01239980.

[SNH15] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. "Studying the relationship between logging characteristics and the code quality of platform software". In: *Empirical Software Engineering* 20.1 (2015), pp. 1–27.

[TRL04] Jeff Tian, Sunita Rudraraju, and Zhao Li. "Evaluating web software reliability based on workload and failure data extracted from server logs". In: *IEEE Transactions on Software Engineering* 30.11 (2004), pp. 754–769.

[Yen+13] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. 2013, pp. 199–208.

[YPZ12] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: *Proceedings of the 34th*

International Conference on Software Engineering. IEEE Press. 2012, pp. 102–112.

[Yua+12] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. "Improving Software Diagnosability via Log Enhancement". In: *ACM Trans. Comput. Syst.* 30.1 (Feb. 2012), 4:1–4:28. ISSN: 0734-2071. DOI: 10.1145/2110356.2110360. URL: http://doi.acm.org/10.1145/2110356.2110360.

[Zen+19] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid". In: *Empirical Software Engineering* 24.6 (2019), pp. 3394–3434.

[Zha+17] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 565–581.

[Zho+20] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. "MobiLogLeak: A Preliminary Study on Data Leakage Caused by Poor Logging Practices". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 577–581.

[Zhu+15] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. "Learning to Log: Helping Developers Make Informed Logging Decisions". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 415–425. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818807.