3L: Learning Linux Logging (Extended Abstract for BENEVOL'15)

Peter Senna Tschudin
Inria, LIP6, UPMC,
Sorbonne Universités
Peter.Senna@lip6.fr

Julia Lawall
Inria, LIP6, UPMC,
Sorbonne Universités
Julia.Lawall@lip6.fr

Gilles Muller
Inria, LIP6, UPMC,
Sorbonne Universités
Gilles.Muller@lip6.fr

***Abstract*—Logging is a common and important programming practice, but choosing how to log is challenging, especially in a large, evolving software code base that provides many logging alternatives. Insufficient logging may complicate debugging, while logging incorrectly may result in excessive performance overhead and an overload of trivial logs. The Linux kernel has over 13 million lines of code, over 1100 different logging functions, and the strategies for when and how to log have evolved over time. To help developers log correctly we propose a framework that will learn existing logging practices from the software development history, and that will be capable of identifying new logging strategies, even when the new strategies just start to be adopted.**

I. Introduction

Logging is the programming practice of inserting statements in the source code to collect and report run time information. This run time information is typically useful for debugging, especially of rare events, which makes logging popular and makes it important for software reliability. In order to log effectively a developer must choose an appropriate position in the source code for the logging statement, a severity level e.g. warning or error, the correct logging function, and how much to log. Deficient logging may leave the user with insufficient information about the program execution, making debugging difficult, while logging incorrectly may result in unintended consequences such as performance overhead and may overwhelm the person trying to understand a problem with redundant information. Thus, developers working on large code bases usually face two distinct, but related problems when deciding how to log: there is no complete specification on how to log, and it is challenging to learn how to log properly, as it involves domain knowledge.

The Linux kernel is an open source operating system with over 13 million lines of C code. In the source code of Linux 4.2, released in August 2015, there are eight logging severity levels, 1,124 logging functions, and 268,156 calls to logging functions. Of the 1,124 logging functions, 81 were not present in Linux 3.16, released around one year previously. While there are 4,396 files in the Linux kernel documentation directory, only two of them contain at least one section that explains how to log properly. An alternative source of information on the latest programming practices, including how to log properly, is the development history of the Linux kernel. Developers may use the history for finding recent changes to existing code, and for finding recently added modules. The development history of the Linux kernel has 535,011 commits since 2005, and 54,827 commits in the last 12 months. The speed at which Linux evolves makes it challenging for developers to stay up-to-date with the latest practices.

Recognizing the difficulty of correct logging, Zhu et al. [1] propose a machine learning framework that learns the common logging practices on where to log from existing logging instances in the source code of large C# user space software. The framework guides developers by suggesting where to log, via the integrated development environment (IDE). Zhu et al. identified the important factors for determining where to log and extracted features from the source code to satisfy their findings. However, the problems of determining the correct logging function, determining the correct severity level, and keeping up to date with the conventions of a fast-changing code base, all of which are particularly relevant to the Linux kernel, remain open.

We propose a framework that learns existing logging practices from features extracted from the software development history, with the goal of enabling our methodology to detect programming practice trends, e.g. API changes, before the new practice becomes widely adopted. We will apply our framework to the development history of the Linux kernel for making suggestions of where to insert logging statements, which logging function to use, and which severity level to use.

Our work proposes to extend previous work by detecting programming practice trends, and by making suggestions of the logging function and severity level to use. Our work will also be different from previous work in terms of the source used for extracting features: we propose to extract features from the development history, and weight the features by age, while previous work only considered the latest version of a code base.

In this paper we present only our preliminary work, focusing on the motivation and methodology of our approach. We leave a complete design, implementation and evaluation for future work.

II. Background

Logging in the Linux kernel is challenging at two main levels. The first challenge is to determine where to log. Here, the

developer must take into account the interaction between the overhead of logging and the stringent performance constraints of some parts of an operating system kernel. The second challenge is to determine how to log. Here, the developer has to carefully choose the right logging function and the right severity level, to ensure that the information provided will be helpful but not overwhelming. We examine these issues in more detail in the rest of this section.

Performance constraints (where to log): One of the main requirements on operating system code is that it be unobtrusive. Thus logging, which incurs a performance overhead, can be incompatible with critical execution paths, such as the processing of high speed network traffic. As a concrete example of the difficulty of choosing where to log and the performance consequences of logging in the wrong place, we consider the Linux BCM4706 ethernet driver[1] of which an extract is shown in Figure 1. This driver contains three badly placed logging function calls, of which one is shown in the final `if` statement of Figure 1 (lines 31-32). The `if` condition shown here (line 30) is the same at all three logging calls, and is related to the hardware properties of the device. Thus, if one of the tests is true, then they are all true, and they are all true for every processed packet. We found that the execution time overhead caused by emitting these three logging messages per network packet caused the driver throughput to slow down by 37.93 times, from 4.4MBps to 116KBps. The first author has submitted a fix for this issue, which has been accepted into the Linux kernel.[2] This example shows that logging must be done only when it is absolutely needed, and that it should be limited to code that is not performance critical. Performance constraints, however, are not always explicit in the source code, making correctly localizing logging code hard for developers.

Debugging challenges (how to log): A general challenge at the kernel level is debugging itself, because of the high degree of concurrency between processes and interrupts, because of the lack of the runtime support required to use conventional debuggers, and because of the close interaction with hardware, which itself can have unpredictable behavior. These factors can make error conditions difficult to reproduce, implying that it is essential that any log messages that are produced be comprehensive and unambiguous. These issues in turn have led to evolution in the Linux kernel logging API, and have resulted in a huge number of different logging functions with different properties.

The top of Figure 2 shows the growth in the number of Linux kernel logging functions over the last roughly 10 years, from Linux 2.6.12 released in 2005 to Linux 4.2 released in 2015. During that period, the number of logging functions increased by 3.42 times. This increase mirrors the increase in the number of lines of source code, which has grown by 3.17 times in the same period, as shown in the bottom of Figure 2. As the kernel indeed continues to grow, we may expect that the

[1]drivers/net/ethernet/broadcom/bgmac.c

[2]Linux kernel commit 8edfe3b6.

```
1  static int bgmac_dma_rx_skb_for_slt(struct bgmac *bgmac,
2      struct bgmac_slot_info *slot)
3  {
4    struct device *dma_dev = bgmac->core->dma_dev;
5    dma_addr_t dma_addr;
6    struct bgmac_rx_header *rx;
7    void *buf;
8
9    /* Alloc skb */
10   buf = netdev_alloc_frag(BGMAC_RX_ALLOC_SIZE);
11   if (!buf)
12     return -ENOMEM;
13
14   /* If poison ok, hardware will overwrite */
15   rx = buf + BGMAC_RX_BUF_OFFSET;
16   rx->len = cpu_to_le16(0xdead);
17   rx->flags = cpu_to_le16(0xbeef);
18
19   /* Map skb for the DMA */
20   dma_addr = dma_map_single(dma_dev, buf +
21       BGMAC_RX_BUF_OFFSET,
22       BGMAC_RX_BUF_SIZE, DMA_FROM_DEVICE);
23   if (dma_mapping_error(dma_dev, dma_addr))
24         ...
25
26   /* Update the slot */
27   slot->buf = buf;
28   slot->dma_addr = dma_addr;
29
30   if (slot->dma_addr & 0xC0000000)
31     bgmac_warn(bgmac,
32     "DMA address using 0xC0000000 bit(s)...\n");
33
34   return 0;
35 }
```

Fig. 1. Logging causing performance problems

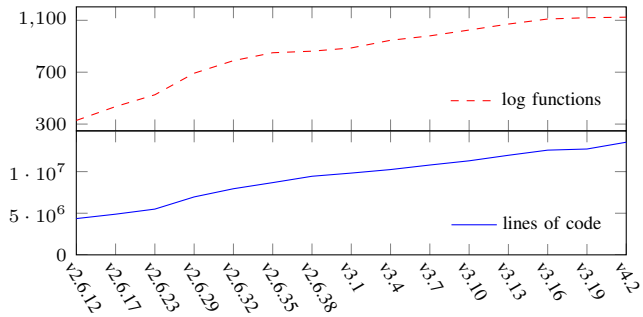

Fig. 2. Number of logging functions and number of lines of code

number of logging functions will also keep growing, further complicating the logging choices facing the developer.

Among the more than 1,000 logging functions provided by the Linux kernel in its most recent release, we can distinguish between a small set of generic logging functions, complemented by a huge collection of logging functions that are dedicated to a particular service, such as a particular device driver or a particular file system. The original kernel logging function, which was already present in Linux 1.0 in 1994, was `printk`, which was accompanied by a family of 8 macros representing different severity levels. These severity levels allow the developer to globally turn on or off log messages with various degrees of criticality, to address performance considerations and to avoid overloading the kernel logs with too much information. Starting in 1996, a new set of API

```
/* XFS logging functions */
static void __xfs_printk(const char *level,
    const struct xfs_mount *mp, struct va_format *vaf)
{
  if (mp && mp->m_fsname) {
    printk("%sXFS (%s): %pV\n", level, mp->m_fsname, vaf);
    return;
  }
  printk("%sXFS: %pV\n", level, vaf);
}

#define define_xfs_printk_level(f, kern_level)          \
void f(const struct xfs_mount *mp, const char *fmt, ...)\
{                                                        \
  struct va_format vaf;                                  \
  va_list args;                                          \
                                                         \
  va_start(args, fmt);                                   \
  vaf.fmt = fmt;                                         \
  vaf.va = &args;                                        \
  __xfs_printk(kern_level, mp, &vaf);                    \
  va_end(args);                                          \
}                                                        \

define_xfs_printk_level(xfs_emerg, KERN_EMERG);
define_xfs_printk_level(xfs_alert, KERN_ALERT);
define_xfs_printk_level(xfs_crit, KERN_CRIT);
define_xfs_printk_level(xfs_err, KERN_ERR);
define_xfs_printk_level(xfs_warn, KERN_WARNING);
define_xfs_printk_level(xfs_notice, KERN_NOTICE);
define_xfs_printk_level(xfs_info, KERN_INFO);
#ifdef DEBUG
define_xfs_printk_level(xfs_debug, KERN_DEBUG);
#endif
```

Fig. 3. Logging functions of the XFS module

functions, with names like `pr_debug` and `pr_info`, started to be introduced with the severity level built in. Today, 93 such functions are available for different classes of kernel services, such as device drivers (*e.g.*, `dev_dbg`) or network device drivers (*e.g.*, `netdev_dbg`). Specific services may in turn use these logging API functions to define service-specific logging functions that encapsulate the logging of various service-specific information. The latter functions are typically both defined and used in a single module. For example, Figure 3 shows the logging functions defined by the XFS file system. These logging functions support a variety of severity levels and make it possible to systematically include environment information such as the module name and the mount point used by the file system, when this information is available.

The difficulty of choosing among the available logging functions is further compounded by their changing usage over time. Figure 4 shows the evolution in the use of logging functions, again over roughly the last 10 years. In the figure, the lower three lines represent the rate of calls, *i.e.*, the number of calls per line of code, to `printk` (dashed and dotted red), to the generic logging API functions ("new api", solid blue), which we recognize as those logging functions beginning with the prefixes `pr_`, `ata_`, `dev`, `fs_`, `hid_`, `net`, or `v4l`, and to all other logging functions ("others", dotted grey). The upper line ("total", dashed black) indicates the rate of any of these calls. We observe that consistently around half (51.84% in Linux 4.2) of the calls are to `printk` or the generic API functions, while the other half (48.16% in Linux 4.2) are to

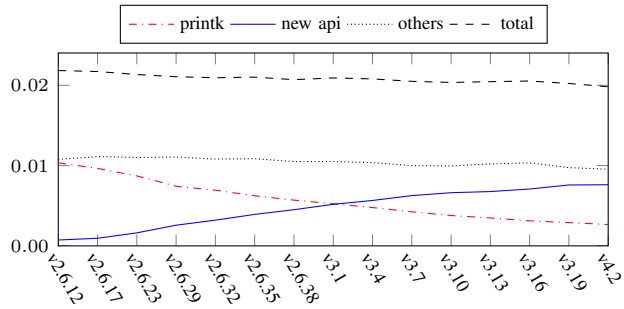

Fig. 4. Number of calls to logging functions per SLOC

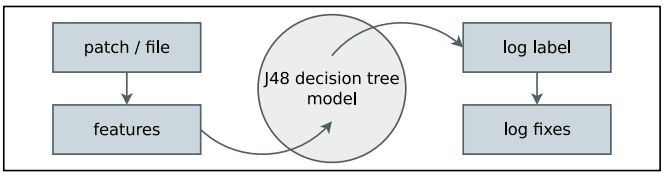

Fig. 5. Our tool

the service-specific functions. However, the rate of use of the generic API functions, which was initially almost nonexistent, has been steadily growing, while that of `printk`, which was initially highly prevalent, has been steadily shrinking. Finally, we can also observe that the overall rate of logging calls is decreasing slowly, reducing by 9.26% in the last 10 years. This may suggest that there are contexts in which logging is no longer needed or no longer desired, further complicating the task of the developer.

III. Methodology

The goal of our tool is to provide relevant information to help developers decide when to log, which severity level to use, and which logging function to use, focusing on error paths. We are designing our tool to be a patch and file verification tool, as might be used in a commit hook of a source code management system such as `git`. An overview of the usage of our tool is shown in Figure 5. Our tool takes as input a patch or file, and then passes this patch or file to the feature extraction pipeline, shown in detail in Figure 6. The obtained features are then used to query a previously trained model that returns a log label. The log label represents the set of logging decisions, including when to log, which logging severity level to use, and which logging function to use. Finally, the inferred log label is compared to the current logging properties of the code, and fixes are proposed for any inconsistencies.

In the rest of this section, we describe the model training process. The result of the training process can be saved for reuse and can be shared to simplify and speed up the use of our tool.

A. Machine Learning Overview

Figure 6 shows our feature and label extraction framework, which represents a typical supervised machine learning

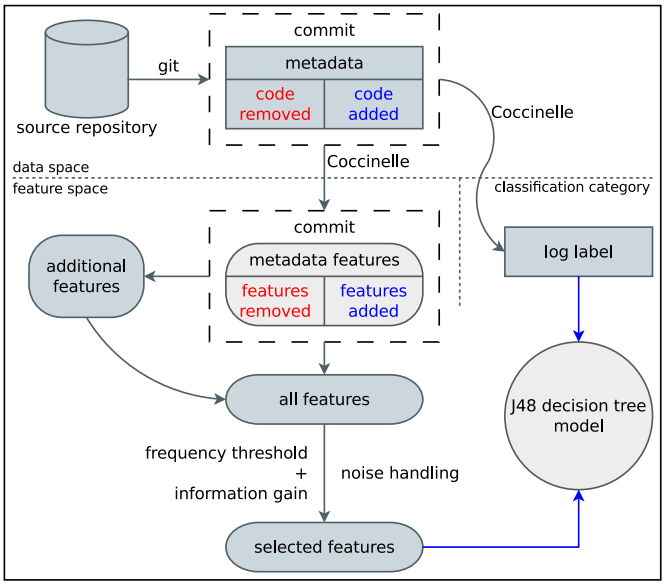

Fig. 6. Feature and label extraction framework

pipeline [2]. The goal of the machine learning process is to construct a model that is able to *label* data elements based on their *features*. Features and labels are representations of the data that must be created to feed the model training process. A feature represents one attribute of the data, such as the error code returned in the error path of a function. A label represents a property of the data, such as the need to log the code snippet. The combination of all features extracted from one instance of the data makes up a training example, and the label of that data instance represents the knowledge the model is learning.

Our training framework, as illustrated in Figure 6, has three parts, the dotted line divides the graph into three data representation spaces: data space, feature space, and classification category. The data space contains the Linux kernel source code, patches and meta data, all provided by the Linux kernel git repository. Git is the source code management system used by the Linux kernel, and makes available all of this information. The feature space shows the processing pipeline of the features extracted from data using the Coccinelle C-code matching and transformation engine [3]. The classification category contains the log label, representing the name of the logging function and the log severity level. The log label is also extracted from the data using Coccinelle. Finally, the circle in the lower right of Figure 6 represents the model being trained with a set of features and a label. We use the J48 decision tree algorithm provided by Weka [4] for training the model.

B. Finding the logging functions

Choosing a label for a given code fragment in the training phase relies critically on knowing whether the code fragment includes logging code. Zhu et al. [1] in their work on user-level C# code have identified logging functions by searching for some relevant keywords in method names. This, however, seems potentially insufficient for the wide variety of logging functions used in the Linux kernel. A possible alternative solution would be to unfold function and macro definitions for calls that contains string parameters, and verify the presence of calls to basic printing functions such as `printk`. We have found, however, that non-logging functions may also contain calls to `printk`-like functions, and thus this approach may result in false positives. We thus decided to rely on properties of call sites rather than names or function and macro definitions for finding logging functions.

Our methodology for finding logging functions combines three semantic patterns. The first is that the function should be called at least once on an `if` branch ending in a `return` statement, representing a typical value-error-check scenario. However, many other functions such as resource release functions and recovery operations are called in the same way, making this criterion not sufficient. The second pattern is that the function should have at least one argument that is a string constant, which we expect will represent the message string. Again, this pattern is not exclusive to logging functions; for example, we have observed that functions like `strcmp` (string comparison) may also be used according to this pattern. The third pattern is that the function should have a variable number of arguments, which is common to `printf` like functions, but is also not exclusive to logging functions. We found that none of these patterns alone produces a list with a low number of false positives, but the combination of the three produces very good results. We used Coccinelle [3] to describe the patterns, and to extract the function names.

C. Features

In this section we will describe our feature extraction and processing pipeline which is used during the model training and by our tool.

Zhu et al. [1] identified important factors for determining where to log in large user space software written in the object-oriented language C#. They identified three categories of source-code features representing the attributes of error paths that are relevant for logging: structural features, textual features, and syntactic features. Structural features refer to the structure of the source code, including the error type, or the exception type, and the methods and functions called. Textual features represent the code as flat text attributes for classification using a bag-of-words model in which the features are the frequency of occurrence of each word. Syntactic features capture eight semantic patterns from the source code, such as the number of called methods and functions.

The Linux kernel is written in the C language, which is not object-oriented, but it makes heavy use of object-oriented concepts such as classes and methods, implemented mostly through the use of structures and pointers. This has made it possible to create a mapping between the features defined by Zhu et al. [1] and the features that are found in the Linux kernel. Many of Zhu et al.'s features could be adapted directly, such as using functions instead of methods, but we added a feature named *entry point*, and we removed a feature named EmptyCatchBlock.

An entry point is a pointer to a function associated with an event. The Linux kernel activates the entry point in response to a known event such as the user requesting a file open operation. Entry points are widely used in the Linux kernel, and all functionalities provided by modules such as device drivers and file systems are registered in the Linux kernel through the declaration of entry points. We added entry points to the structural and textual features because knowing the entry points that lead to a particular fragment of code, potentially interprocedurally, is essential to understanding the code fragment's functionality. In the case of EmptyCatchBlock, another difference is the lack of recognizable try-catch-throw exception handling in the Linux kernel and consequently the lack of empty catch blocks as a particular feature. We use Coccinelle to extract features from the source code.

We apply a set of feature selection and noise handling techniques before training the model to increase the accuracy of the suggestions of our tool. As our approach is similar to the approach used by Zhu et al. [1], we do not include a detailed description of this process due to space limitations.

D. Labels

The label represents the knowledge that the model is learning: where to log, which logging function to use and which logging severity label to use. The label plays a somewhat different role when training the model, where it is part of the training data, and when using the tool, where it is the result produced by the model.

When training the model, the label is chosen based on the presence of a call to a logging function in the data. Our training framework infers the label automatically, based on the set of logging functions identified as described in Section III-B. For the cases in which there is no call to a logging function, the label is `not logged`. For the cases in which a call to a logging function is present, the label encapsulates the function name and the logging severity level. The logging severity level is determined by following the call graph of the logging function until reaching a call to a logging function that explicitly includes a log level, such as `KERN_WARNING`.

When using the tool, as illustrated in Figure 5, only the features are used to query the machine learning model. The model then produces a label that represents the correct logging decision for that set of features. However, a label is also extracted from the data, as done in the training phase, to identify the current state of logging in the code. This extracted label is then compared to the label inferred by the model. If the labels are identical, it means that the logging strategy taken by the patch or file is already correct. If they differ, however, it means that the logging performed by the patch or file can be improved, and we are designing our tool to emit a fix suggestion.

IV. Threats to validity

In the current state of our work, the main threat to validity is an internal threat to validity, in the correctness of our strategy for identifying logging functions. This identification strategy determines how our model is trained, and thus the correctness of its results. To address this threat, we have carefully studied the list of identified logging functions, and cross checked it against the results of other more approximate strategies, such as searching for certain kinds of substrings in function names, to ensure that no obvious candidates are overlooked.

V. Related Work

The work most closely related to ours is that of Zhu et al. [1] on providing logging suggestions for C# code that we have already described extensively. ErrLog [5] is a tool with similar goals that has been applied to various user-level infrastructure software and that tries to recognize error conditions and ensure that all are logged. Other tools that have been developed around logging include an empirical study of logging practices in open-source software [6], LogEnhancer that extends existing log messages with relevant information [7], and SherLog for helping developers effectively use information in log messages to guide debugging [8].

Various machine-learning approaches have been applied to improving the quality of Linux kernel code, including the work of Engler et al. [9] on identifying bugs from deviant code patterns, and an extension of that work by Li and Zhou [10]. Our work, in contrast, specifically targets logging, and uses specific properties of logging code.

VI. Conclusion

In this paper, we have identified the unique features and challenges of correct logging at the operating system kernel level, exemplified by the Linux kernel, and we have proposed a machine-learning based methodology for helping developers make correct logging decisions. In the future, we plan to complete and implement our methodology, and evaluate our approach on the recent history of Linux kernel code.

References

[1] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *ICSE*, 2015, pp. 415–425.

[2] S. Kotsiantis, "Supervised machine learning: A review of classification techniques," *Informatica*, vol. 31, pp. 249—268, 2007.

[3] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys*, 2008, pp. 247–260.

[4] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[5] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *OSDI*, 2012, pp. 293–306.

[6] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE*, 2012, pp. 102–112.

[7] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS*, 2011, pp. 3–14.

[8] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," in *ASPLOS*, 2010, pp. 143–154.

[9] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as inconsistent behavior: A general approach to inferring errors in systems code," in *SOSP*, 2001, pp. 57–72.

[10] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE*, 2005, pp. 306–315.