EasyLog: An Efficient Kernel Logging Service for Machine Learning

Xundi Yang, Kefan Qiu, and Quanxin Zhang(✉)

Beijing Institute of Technology University, Beijing 100081, China
{kfqiu,zhangqx}@bit.edu.cn

Abstract. Recently, logs serves as a crucial tool to monitor system's real-time state for experiments and generate data for machine learning. However, the existing Linux logging system faces challenges such as excessive log output and a high rate of important log message loss. To tackle these issues, we propose the EasyLog solution, which effectively mitigates these problems. EasyLog draws inspiration from the design principles of log-related functions like `pr_xx`, `dev_xx`, and the `Devkmsg` service. EasyLog extracts and records logs with special identifier suffixes by introducing a ring buffer. In terms of interface utilization, EasyLog offers the `easy_xx` interface for kernel developers and the reading interface for user-space applications.

Keywords: Machine learning systems · Linux logging service · Log message loss · Ring buffer · Interface utilization · Server kernel development

1 Introduction

In recent years, there has been a growing deployment of machine learning systems on Linux Operating Systems. Within the Linux environment, numerous runtime problems and potential threats necessitate resolution through security monitoring and analysis, which includes the examination of logs to identify causes [1]. Log information serves as an immediate reflection of a system's operational status. Developers utilize logs for diagnosing system malfunctions, recording experimental results, and generating data for training security analysis models.

In this paper, we focus on the logging service in the Linux server kernel and aim to assist programmers in designing kernel drivers and generating data for machine learning. Generally, developers strive to capture system state dumps, execute tracing, and communicate events through log data. Within the 13,390,104 lines of source code in the Linux kernel, there are 498,897 lines (approximately 3.79%) dedicated to logging code [2]. Logging functions in the Linux kernel write messages into the log buffer. Commonly used logging functions include the `printk` function, which is similar to the `printf` function. The distinction lies in `printk`'s specification of the log level for event recording. Subsequent Linux kernel versions have introduced variations of the `printk` function such as `pr_info`, and `dev_warn`, which incorporate log levels into their nomenclature.

M. Andreoni (Ed.): ACNS 2024 Workshops, LNCS 14586, pp. 16–28, 2024.
https://doi.org/10.1007/978-3-031-61486-6_2

However, the existing kernel logging services are unreliable due to several factors. Firstly, the Linux kernel generates a substantial volume of continuous log output from numerous programs. When Shiqing et al. [3] tested the overhead of the Linux audit system, they found that servers generate approximately 130GB of log data per day, while client machines generate about 5GB of log data per day. Nevertheless, developers often focus on specific Linux kernel modules. For instance, Tan Y et al. [4] employ authorization lists recorded in the logs as the basis for comparison. In the RootAgency [5], the logs document the time consumption from the test app initiating the request until the end user receives the reply for the root privilege request. Neither of them pays attention to logs related to unrelated modules. The excessive volume of irrelevant logs has caused interference in their experiments. Furthermore, the kernel logging service also faces the issue of log loss. The kernel log buffer is a ring buffer, which operates on a first-in, first-out (FIFO) basis. The default size of the Linux kernel log buffer is 128KB. As data accumulates beyond the capacity of the ring buffer, the oldest data is overwritten to accommodate new information.

In this paper, we introduce a kernel logging service solution, EasyLog, which effectively mitigates the above issues. In terms of interface utilization, EasyLog provides write interfaces such as `easy_xx` for log writing at the kernel layer and system call interfaces such as `open`, `read`, and `close` for log reading at the application layer. Besides, regarding log simplification, the design of the `easy_xx` functions is influenced by kernel functions like `dev_info`, appending specific identifiers to the end of each log. When our write interfaces append to record logs, EasyLog extracts logs using the identifier and stores them in our new ring log buffer, which can be expanded to 2 MB. This approach significantly reduces the volume of logs. It also elongates the time required to fill the circular buffer and decreases the likelihood of log loss. Furthermore, developers can utilize the `easy_xx` functions in their experimental kernel, subsequently compile and execute the code, and directly get logs from the new log buffer. To sum up, EasyLog reduces log volume, enhances effective log density, diminishes the probability of log loss, and facilitates the development of new modules for programmers.

The rest of the paper is structured as follows. Section 2 describes recent application of logs, the principles underlying the `printk` mechanism, and recent advancements in kernel-level logging service. In Sect. 3, the comprehensive architectural design is presented. Section 4 describes the details of the implementation and the interface design of EasyLog. In Sect. 5, we validate the effectiveness of EasyLog in mitigating the loss rate of important logs. Section 6 elucidates the utilization of the EasyLog service to assist in the development of new modules for the Linux kernel. Section 7 concludes.

2 Background

2.1 The Applications of Logs

When the kernel crashes, developers can solve bugs by analyzing the preserved system log files. In recent years, logs aid developers in diagnosing system errors, training security models, and documenting experimental results. In 2015, the EASEAndroid platform [6], the inaugural audit log analytic system for SEAndroid, employed semi-supervised learning to autonomously enhance the SEAndroid policy. In 2018, Xue B et al. [7] obtain the encryption rate and data processing size of the baseband processor through log information. In 2021, Li Y et al. [1] proposes a host security analysis method based on D-S evidence theory, which involves extracting information from monitoring logs and subsequently training a security analysis model. The model can be applied to host security analysis in different operating systems with minimal or almost no modification.

2.2 The Analysis of Printk

The kernel log module resides in `./kernel/printk/`. Figure 1 illustrates the read-write framework of the kernel log module. As depicted in the figure, its core component is the ring buffer, denoted as the “log buffer”. The `printk` function and the `devkmsg_write` function, acting as producers, store messages in the log buffer. On the other hand, the log service modules on the right side of the figure function as a consumer, reading messages from the log buffer.

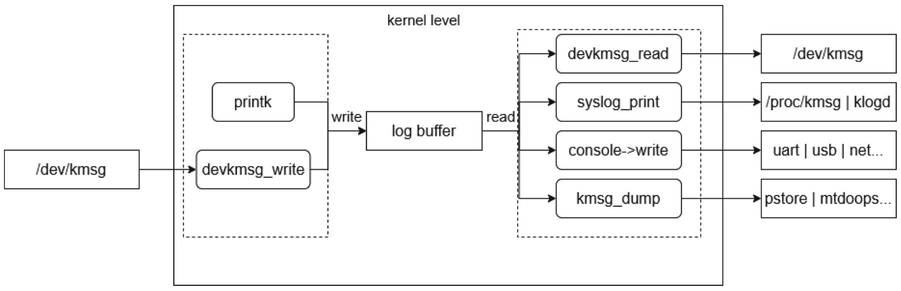


Fig. 1. The Read-Write Framework of the kernel log module

Log Buffer: The size of the kernel log buffer is determined jointly by the configuration parameters `CONFIG_LOG_BUF_SHIFT` and the number of CPU cores in the SMP system. During the kernel boot, information regarding the system's memory layout and the number of CPU cores is unknown before device tree parsing. To support the utilization of the `printk` function, the kernel defines a static global log buffer with a size of `(1 << CONFIG_LOG_BUF_SHIFT)`. After CPU initialization, an additional global log buffer is dynamically allocated with a size of `(1 << CONFIG_LOG_BUF_SHIFT + 1 << LOG_CPU_MAX_BUF_LEN)`, and the log data from the original static buffer is copied into it.

The log buffer is managed through the data structure `printk_ringbuffer`. Figure 2 presents the data structures of `printk_ringbuffer`. As depicted, it comprises three main components: 1) a ring buffer for data storage, managed using head and tail pointers to track the buffer's status. When data needs to be written, the head pointer is updated based on the length of the data being written. If the free space in the ring buffer is insufficient, the oldest data is purged starting from the tail pointer. Additionally, each piece of written data is assigned an ID, which is used to indicate the index in the `prb_desc` array and the `printk_info` array. 2) An array of `prb_desc` structures, with each element maintaining the position information of a log within the ring buffer, along with its status. 3) An array of `printk_info` structures, with each element responsible for managing additional information associated with a log, such as its sequence number(seq), timestamp, length, log level, and more.

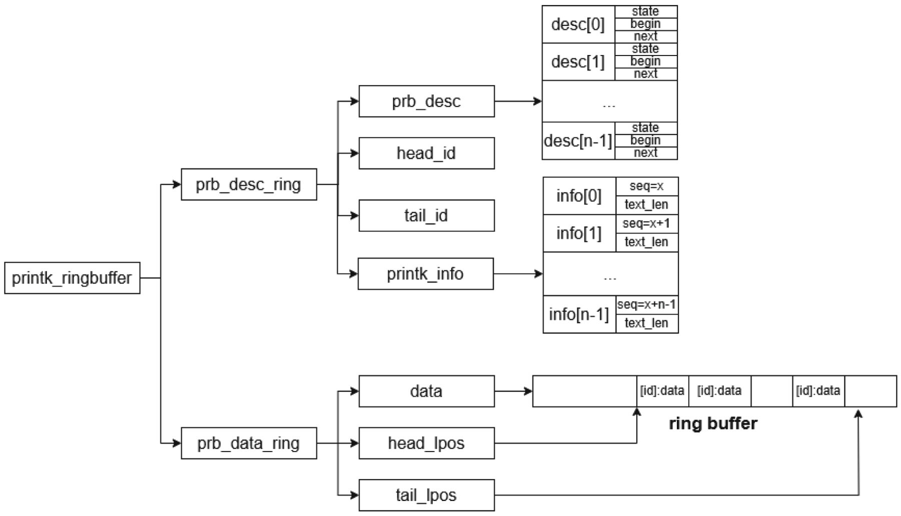


Fig. 2. Principal Members and Data Structures of printk_ringbuffer

Log Stoarge Process: We use the `printk` interface as an example for explaining the log storage process. 1) Allocate an entry and increment the sequence number: The `desc_reserve` function retrieves an available entry from the `prb_desc` structure array. If no free entry is available, it overwrites the oldest log. Upon successful allocation, the descriptor's status is set to `desc_reserved`. Subsequently, the sequence number for that log is set. 2) Allocate space and copy the log: The `data_alloc` function allocates a segment of space from the ring buffer to store the new log data. The `log_data_copy` function copies the data to be written into the allocated space within the ring buffer. 3) Update the status: The `_prb_commit` function updates the status of the new element in the `prb_desc` array to `desc_committed`, and then the `desc_make_final` function updates the status to `desc_finalized`. After this operation, the log is written and ready for reading.

Log Retrieval Process: When reading logs, a sequence number (seq) is provided as a parameter, and the prb_read function is called to retrieve the log corresponding to that sequence number. 1) Retrieve a valid status log: The desc_read_finalized_seq function reads the status of the log corresponding to the provided sequence number. If the status is valid, the subsequent log data and information retrieval operations are executed. After completion, the status of that log is rechecked. If it remains valid and the sequence number has not changed, it signifies that during the reading process, the data was not modified by write operations. Otherwise, the reading process fails. 2) Handling of Reading Failures: In case of reading failures, the prb_first_seq function is called to obtain the first readable log after the provided sequence number, and the retrieval process is restarted.

Devkmsg Log Interface: The devkmsg service provides log read and write operations to user space through the device file node /dev/kmsg. Devkmsg maintains an independent sequence number and log reading is based on this sequence number to determine which log needs to be read. The user space read interface provided by EasyLog in this paper is modeled after the Devkmsg service.

Other Interfaces: 1) syslog interface: This interface exports logs through system calls but does not provide log writing operations. The syslog interface finds utility in various scenarios, including applications like dmesg, klogd, and /proc/kmsg. 2) Console log interface: It primarily offers console initialization, and registration processes, and specifies the preferred console interface through command-line parameters. 3) Kmsg dump interface: This interface is primarily used by pstore. pstore is applied to save system logs to a backend device in the event of a system crash, assisting developers in debugging and analysis.

2.3 Recent Work About Logging

Linux manages storage devices, networks, man-machine interfaces, CPUs, and more through software layers such as device drivers, file systems, and communication protocols. These intricate modules are maintained by hundreds of programmers. As Linux grows in complexity, an increasing number of system analysis tools have been proposed to help developers in analyzing system behavior. The simplest logging tool in Linux is the printk function, as mentioned earlier. In Linux kernel v1.3.983 [2], a set of additional logging functions was introduced to enhance the conciseness of log statement recording. These functions incorporate log levels in their names. Consequently, programmers are no longer required to use printk function with log-level parameters such as KERN_DEBUG and KERN_INFO. Another set of logging functions specifically designed for device drivers, such as dev_dbg and dev_info, automatically embed the device name in their outputs, thereby facilitating the identification of the source of log messages.

Both dev_xx functions and pr_xx functions are variants of the printk function, and the underlying issues with printk remain unresolved. The printk function uses an asynchronous daemon to read and write a ring buffer, making the buffer vulnerable to overwriting and event loss. The Linux Trace Toolkit (LTT) [8] logs around 45 predefined events, including interrupts, system calls, and network packet arrivals. The tool is advantageous due to its relatively low overhead and the presence of a visualization tool to aid in analyzing logged data. However, it lacks flexibility and scalability. Relayfs [9] is proposed, which divides logs into different subsystem/client channels, effectively addressing the fundamental overhead caused by locking during logging. KLogger [10] is presented as a software tool for logging operating system kernel events. Developers can insert new log events into the kernel using this tool. Furthermore, an alternative approach to logging all events is sampling. OProfile [11] adopts a sampling approach, serving as the underlying infrastructure for HP's Prospect tool. OProfile uses Intel's hardware performance counters to generate traps for every N occurrence of specific hardware events. However, since OProfile is based on periodic sampling, it may miss events with finer granularity than the sampling rate.

3 Architecture

Figure 3 illustrates the architecture of EasyLog. 1) The server kernel subsystem utilizes the `easy_xx` functions to record log entries. 2) EasyLog service filters logs with specific suffixes and directs them into a newly created ring buffer. 3) User-space applications access the new log buffer through the character device node `/dev/easylog`.

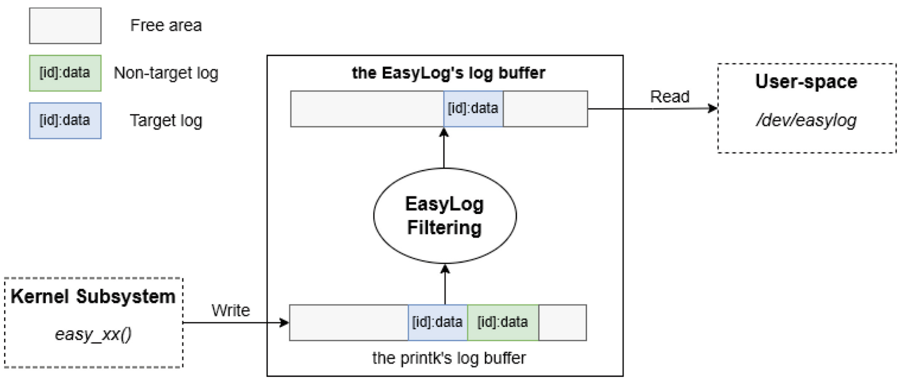


Fig. 3. The architecture of EasyLog

4 Implementation

4.1 Channel and Data Management Schemes

The EasyLog module maintains a ring buffer as its log buffer. Its structure, initialization, and read-write processes resemble the `printk`'s log buffer.

Initialization: During the early stages of kernel startup, the original static log buffer is still employed. Subsequently, in the `start_kernel` function, an initialization function is called to dynamically allocate a global log buffer of size `(1>>CONFIG_EASYLOG_BUF_SHIFT)`.

Log Storage: The kernel subsystem uses the `easy_xx` functions to record log entries, with each log appended with a special identifier suffix.

Log Filtering: The `log_store` function is responsible for appending logs generated by other modules to the original log buffer. Within the `log_store` function, the EasyLog service performs a hook-like operation. The service filters logs generated by other modules, extracts logs with suffixes, removes the suffixes, and subsequently stores them in our log buffer following the writing procedure described in Sect. 2.1. This process ensures that the log buffer's read and write operations do not conflict through spin locks and local interrupt disabling. During this filtering step, both our log buffer and the `printk`'s log buffer need to be locked for protection.

Log Retrieval: EasyLog maps our ring buffer to a pseudo-file, namely the character device file `/dev/easylog`, which can be accessed by userspace for reading or process memory mapping.

4.2 Interface

This section describes the basic utilization of EasyLog in both kernel subsystems and user space. Within the kernel subsystem, a set of kernel-space APIs can be employed to write logs with suffixes, facilitating the extraction of logs by the EasyLog service. In the user space, user programs can access logs from EasyLog by reading from the character device node `/dev/easylog`.

Drawing inspiration from the `pr_xx` functions, the kernel subsystems write a special log using the `easy_xx` function, where 'xx' denotes the corresponding log level. Each kernel log comprises three essential components [2]: the kernel event level, a static message detailing the event, and variable values associated with the log event. Detailed API definitions are provided below.

Code 1: The definition of easy_xx functions

```
__printf(2, 3) void easy_suffix_printk(const char *, const char *,
    ...);

#define easy_emerg(fmt, ...) \
   easy_suffix_printk(KERN_EMERG, fmt, ##__VA_ARGS__)
#define easy_crit(fmt, ...) \
   easy_suffix_printk(KERN_CRIT, fmt, ##__VA_ARGS__)
#define easy_alert(fmt, ...) \
   easy_suffix_printk(KERN_ALERT, fmt, ##__VA_ARGS__)
```

```
#define easy_err(fmt, ...) \
    easy_suffix_printk(KERN_ERR, fmt, ##__VA_ARGS__)
#define easy_warn(fmt, ...) \
    easy_suffix_printk(KERN_WARNING, fmt, ##__VA_ARGS__)
#define easy_notice(fmt, ...) \
    easy_suffix_printk(KERN_NOTICE, fmt, ##__VA_ARGS__)
#define easy_info(fmt, ...) \
    easy_suffix_printk(KERN_INFO, fmt, ##__VA_ARGS__)
```

In this context, the `easy_suffix_printk` function, inspired by the `dev_xx` functions, appends the "–easylog–" identifier to the end of each log. Subsequently, it invokes the `vprintk_emit` function, passing the kernel event level and the newly generated log with the suffix.

The reading operation of user programs in EasyLog draws inspiration from the reading operation of the `devkmsg` service. `/dev/easylog` is a readable character device file, permitting multiple user processes to access log records. Each process can obtain a complete set of log entries from EasyLog.

The collection of user-space file operations provided by EasyLog is as follows: 1) The `open` function, responsible for opening the character device file node `/dev/easylog` and creating a `deveasylog_user` structure object (as detailed later) named `user`, corresponding to the `deceasylog_open` function in the kernel. 2) The `read` function, which reads log entries from the log buffer, corresponds to the `deceasylog_read` function in the kernel. 3) The `release` function, responsible for releasing all resources acquired by the `open` function, corresponding to the `deceasylog_release` function in the kernel.

Code 2: The collection of user-space file operations provided by EasyLog

```
const struct file_operations easylog_fops = {
        .open = deveasylog_open,
        .read = deveasylog_read,
        .release = deveasylog_release,
};
```

Each process that opens `/dev/EasyLog` is associated with an independent `deveasylog_user` structure object, as described below. The `deveasylog_user` structure maintains a unique sequence number for each reading process, denoted as `seq`, which represents the sequence number of the log currently being read by the process. The mutex lock, denoted as `lock`, ensures that only one thread within each process can perform write operations on the `text_buf`. The `rs` variable is used for rate limiting. Following the Log Retrieval Process in Sect. 2.1, logs are read from EasyLog's log buffer, recorded in the `text_buf`, and subsequently returned to user space by invoking the `copy_to_user` function.

Code 3: The structure of deveasylog_user

```
struct deveasylog_user {
        u64 seq;
        struct ratelimit_state rs;
        struct mutex lock;
        char buf[CONSOLE_EXT_LOG_MAX];
        struct printk_info info;
        char text_buf[CONSOLE_EXT_LOG_MAX];
        struct printk_record record;
};
```

5 Experiment

This section of experiments aims to demonstrate the significant reduction in the loss rate of important logs achieved by EasyLog. Important logs refer to the logs associated with the development modules during server kernel development. Within the developing server kernel subsystems, developers invoke the `easy_xx` functions and append specific identifiers to the end of each log. This experiment considers such logs as important logs, defined as label logs. EasyLog extracts the logs with specific identifiers and stores them in a new ring log buffer.

In our experiments, it was essential to simulate real-world log generation scenarios as closely as possible, ensuring that normal logs and label logs terminated their output as synchronously as feasible. The original `printk` and EasyLog log buffer were set to 256 KB. We designed two kernel modules: one module invokes `pr_info` to write N normal log entries, while the other module invokes `easy_info` to write N*0.1 label log entries, ensuring a ratio of 100 normal logs to 10 label logs. The values of N range from $2048, 4096, 6144...20480$. Additionally, normal logs were generated at intervals of 0.1 s, whereas the generation intervals for label logs were a random number between $[0.5, 1.5]$ (with a mean of 1). These two modules were executed concurrently, writing normal logs and label logs in parallel. We consider the logs recorded in `/dev/kmsg` as the logs logged by the original logging system, while the logs in `/dev/easylog` are the logs recorded by EasyLog. The loss rate of label logs was tested separately. The formula for calculating the loss rate of label logs is provided in Eq. 1. Table 1 presents the experimental results.

$$Label\ Logs\ Loss\ Rate = \frac{Label\ Logs\ Loss\ Count}{Label\ Logs\ Count} \times 100\% \tag{1}$$

Table 1. The loss rate of label logs

NO	Label Logs Count[a]	Normal Logs Count[b]	Total Logs Count	Label Logs Loss Rate	
				/dev/kmsg[c]	/dev/easylog[d]
1	205	2,048	2,253	0	0
2	410	4,096	4,506	0	0
3	614	6,144	6,758	0	0
4	819	8,192	9,011	8.91%	0
5	1,024	10,240	11,264	26.86%	0
6	1,229	12,288	13,517	39.71%	0
7	1,434	14,336	15,770	48.12%	0
8	1,638	16,384	18,022	54.46%	0
9	1,843	18,432	20,275	59.31%	0
10	2,048	20,480	22,528	62.79%	0

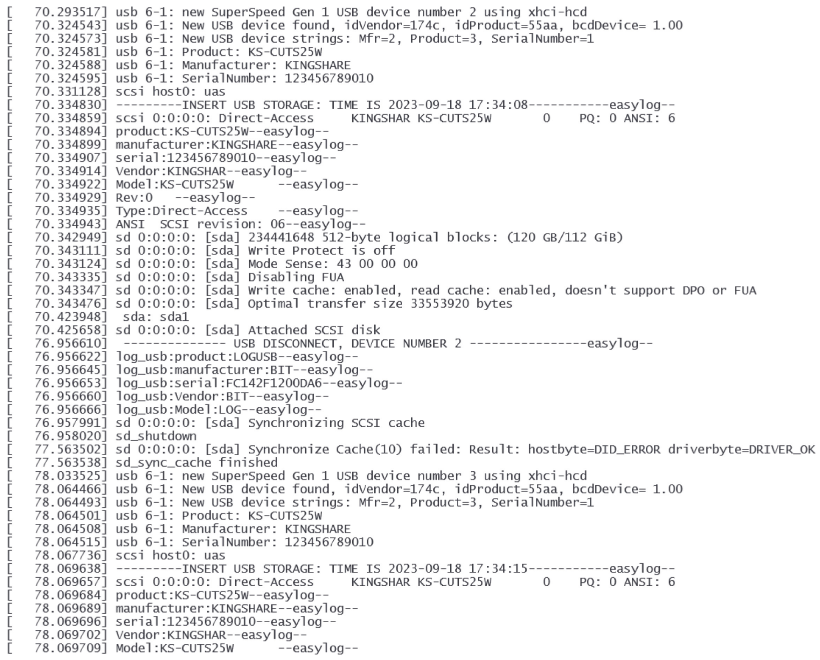
[a] The generation intervals for label logs were a random number between $[0.5, 1.5]$.
[b] Normal logs were generated at intervals of 0.1 s.
[c] The logs in `/dev/kmsg` are the logs logged by the original logging system.
[d] The logs in `/dev/easylog` are the logs recorded by EasyLog.

The Table 1 shows that: 1) When the log volume is relatively small, there is no significant difference in the loss rate of label logs and it remains at 0. 2) As the log volume gradually increases, the loss rate of label logs in the `/dev/kmsg` increases progressively. 3) Due to the constraint of ensuring that normal and label logs terminate their outputs concurrently, the growth in the loss rate of label logs in the `/dev/kmsg` slows down as the log volume increases, and it will not reach 100%. 4) Because the total size of label logs is less than 256 KB, the loss rate of label logs consistently remains at 0. In summary, EasyLog reduces the volume of logs that need to be recorded by extracting important logs, allowing the system sufficient time to store the logs from the buffer into files, thus effectively reducing the probability of log loss.

6 Application

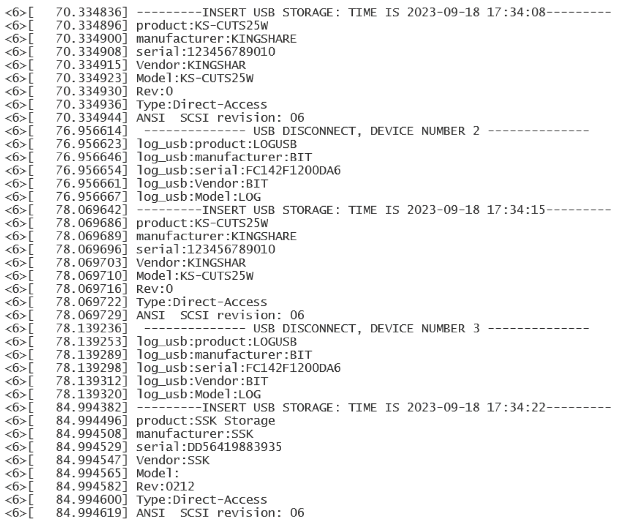
EasyLog aims to assist programmers in designing kernel drivers and generating data for machine learning. In this section, we demonstrate how EasyLog aids developers in kernel driver development. Some hardware devices, such as GPUs, have old versions phased out and new versions released, necessitating corresponding driver updates. Besides, the driver subsystems are significantly larger than other subsystems. From Linux versions v4.3 to v5.3, there were a total of 211,437 modifications to log statements, with the driver subsystem accounting for 86.60% of the overall log code changes [2]. Therefore, optimizing the logging system is of paramount importance for Linux server kernel driver development.

In this chapter, we take USB storage device-related drivers as an example. We output logs when USB flash drives are inserted and removed. The log content includes relevant information about the USB flash drive, such as product, vendor, manufacturer, serial number, as well as the time of insertion and removal. The



```
[   70.293517] usb 6-1: new SuperSpeed Gen 1 USB device number 2 using xhci-hcd
[   70.324543] usb 6-1: New USB device found, idVendor=174c, idProduct=55aa, bcdDevice= 1.00
[   70.324573] usb 6-1: New USB device strings: Mfr=2, Product=3, SerialNumber=1
[   70.324581] usb 6-1: Product: KS-CUTS25W
[   70.324588] usb 6-1: Manufacturer: KINGSHARE
[   70.324595] usb 6-1: SerialNumber: 123456789010
[   70.331128] scsi host0: uas
[   70.334830] ---------INSERT USB STORAGE: TIME IS 2023-09-18 17:34:08-----------easylog--
[   70.334859] scsi 0:0:0:0: Direct-Access     KINGSHAR KS-CUTS25W       0    PQ: 0 ANSI: 6
[   70.334894] product:KS-CUTS25W--easylog--
[   70.334899] manufacturer:KINGSHARE--easylog--
[   70.334907] serial:123456789010--easylog--
[   70.334914] Vendor:KINGSHAR--easylog--
[   70.334922] Model:KS-CUTS25W       --easylog--
[   70.334929] Rev:0   --easylog--
[   70.334935] Type:Direct-Access    --easylog--
[   70.334943] ANSI  SCSI revision: 06--easylog--
[   70.342949] sd 0:0:0:0: [sda] 234441648 512-byte logical blocks: (120 GB/112 GiB)
[   70.343111] sd 0:0:0:0: [sda] Write Protect is off
[   70.343124] sd 0:0:0:0: [sda] Mode Sense: 43 00 00 00
[   70.343335] sd 0:0:0:0: [sda] Disabling FUA
[   70.343347] sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
[   70.343476] sd 0:0:0:0: [sda] Optimal transfer size 33553920 bytes
[   70.423948]  sda: sda1
[   70.425658] sd 0:0:0:0: [sda] Attached SCSI disk
[   76.956610]  -------------- USB DISCONNECT, DEVICE NUMBER 2 ----------------easylog--
[   76.956622] log_usb:product:LOGUSB--easylog--
[   76.956645] log_usb:manufacturer:BIT--easylog--
[   76.956653] log_usb:serial:FC142F1200DA6--easylog--
[   76.956660] log_usb:Vendor:BIT--easylog--
[   76.956666] log_usb:Model:LOG--easylog--
[   76.957991] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[   76.958020] sd_shutdown
[   77.563502] sd 0:0:0:0: [sda] Synchronize Cache(10) failed: Result: hostbyte=DID_ERROR driverbyte=DRIVER_OK
[   77.563538] sd_sync_cache finished
[   78.033525] usb 6-1: new SuperSpeed Gen 1 USB device number 3 using xhci-hcd
[   78.064466] usb 6-1: New USB device found, idVendor=174c, idProduct=55aa, bcdDevice= 1.00
[   78.064493] usb 6-1: New USB device strings: Mfr=2, Product=3, SerialNumber=1
[   78.064501] usb 6-1: Product: KS-CUTS25W
[   78.064508] usb 6-1: Manufacturer: KINGSHARE
[   78.064515] usb 6-1: SerialNumber: 123456789010
[   78.067736] scsi host0: uas
[   78.069638] ---------INSERT USB STORAGE: TIME IS 2023-09-18 17:34:15-----------easylog--
[   78.069657] scsi 0:0:0:0: Direct-Access     KINGSHAR KS-CUTS25W       0    PQ: 0 ANSI: 6
[   78.069684] product:KS-CUTS25W--easylog--
[   78.069689] manufacturer:KINGSHARE--easylog--
[   78.069696] serial:123456789010--easylog--
[   78.069702] Vendor:KINGSHAR--easylog--
[   78.069709] Model:KS-CUTS25W       --easylog--
```

Fig. 4. The output of "dmesg". The figure indicates the original kernel logs include too much unrelated information.



```
<6>[   70.334836] ---------INSERT USB STORAGE: TIME IS 2023-09-18 17:34:08---------
<6>[   70.334896] product:KS-CUTS25W
<6>[   70.334900] manufacturer:KINGSHARE
<6>[   70.334908] serial:123456789010
<6>[   70.334915] Vendor:KINGSHAR
<6>[   70.334923] Model:KS-CUTS25W
<6>[   70.334930] Rev:0
<6>[   70.334936] Type:Direct-Access
<6>[   70.334944] ANSI  SCSI revision: 06
<6>[   76.956614]  -------------- USB DISCONNECT, DEVICE NUMBER 2 --------------
<6>[   76.956623] log_usb:product:LOGUSB
<6>[   76.956646] log_usb:manufacturer:BIT
<6>[   76.956654] log_usb:serial:FC142F1200DA6
<6>[   76.956661] log_usb:Vendor:BIT
<6>[   76.956667] log_usb:Model:LOG
<6>[   78.069642] ---------INSERT USB STORAGE: TIME IS 2023-09-18 17:34:15---------
<6>[   78.069686] product:KS-CUTS25W
<6>[   78.069689] manufacturer:KINGSHARE
<6>[   78.069696] serial:123456789010
<6>[   78.069703] Vendor:KINGSHAR
<6>[   78.069710] Model:KS-CUTS25W
<6>[   78.069716] Rev:0
<6>[   78.069722] Type:Direct-Access
<6>[   78.069729] ANSI  SCSI revision: 06
<6>[   78.139236]  -------------- USB DISCONNECT, DEVICE NUMBER 3 --------------
<6>[   78.139253] log_usb:product:LOGUSB
<6>[   78.139289] log_usb:manufacturer:BIT
<6>[   78.139298] log_usb:serial:FC142F1200DA6
<6>[   78.139312] log_usb:Vendor:BIT
<6>[   78.139320] log_usb:Model:LOG
<6>[   84.994382] ---------INSERT USB STORAGE: TIME IS 2023-09-18 17:34:22---------
<6>[   84.994496] product:SSK Storage
<6>[   84.994508] manufacturer:SSK
<6>[   84.994529] serial:DD56419883935
<6>[   84.994547] Vendor:SSK
<6>[   84.994565] Model:
<6>[   84.994582] Rev:0212
<6>[   84.994600] Type:Direct-Access
<6>[   84.994619] ANSI  SCSI revision: 06
```

Fig. 5. The output of "cat /dev/easylog". The figure indicates the EasyLog's logs are clearer and more coherent.

driver functions involved in this process include the `scsi_add_lun` function in `scsi/scsi_scan.c` and the `usb_disconnect` function in `usb/core/core.c`. In these respective locations within the functions, the `easy_info` function is called to write the relevant log content. Repeatedly inserting and removing different USB flash drives or hard drives, Fig. 4 and Fig. 5 depict screenshots of the `dmesg` output and the `cat /dev/easylog` output. The Fig. 5 is clearer and more coherent, devoid of interference in the logs, which is advantageous for programmers working on new modules.

7 Conclusion

To address issues such as log loss and excessive log volume in the Linux logging service, this paper proposes the EasyLog service. EasyLog maintains a ring buffer as the log buffer. EasyLog extracts logs with special identifiers from the logs written to the original log buffer and stores them in our new log buffer. Furthermore, EasyLog provides `write` functions such as `easy_xx` for kernel modules to write logs and `read` functions for applications to read logs.

In the fifth section, experimental results demonstrate that when the total log volume is relatively low, both EasyLog and the original Linux logging module have a log loss rate of 0 for critical logs. However, when the proportion of critical logs remains constant but the total log volume increases, the log loss rate of critical logs in the original Linux logging system gradually rises. In contrast, the EasyLog module en sures that these critical logs are not overwritten by unrelated logs by extracting critical logs and storing them in our new ring buffer. As a result, the log loss rate of critical logs in EasyLog is significantly lower than in the original Linux logging system. When the total size of critical logs does not exceed the log buffer capacity, the loss rate of critical logs is 0.

In conclusion, EasyLog proficiently mitigates log loss concerns and facilitates the creation of kernel drivers as well as the generation of data for machine learning applications.

Acknowledgements. This work was supported by the National Key Research and Development Program of China under Grant 2022YFB2701501 the National Natural Science Foundation of China (no. U2336201).

References

1. Li, Y., Yao, S., Zhang, R., et al.: Analyzing host security using D-S evidence theory and multisource information fusion. Int. J. Intell. Syst. **36**(2), 1053–1068 (2021)
2. Patel, K., Faccin, J., Hamou-Lhadj, A., et al.: The sense of logging in the linux kernel. Empir. Softw. Eng. **27**(6), 153 (2022)
3. Ma, S., Zhai, J., Kwon, Y., et al.: Kernel-supported cost-effective audit logging for causality tracking. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 241–254 (2018)

4. Tan, Y., Xue, Y., Liang, C., et al.: A root privilege management scheme with revocable authorization for Android devices[J]. J. Netw. Comput. Appl. **107**, 69–82 (2018)
5. Xue, Y., Tan, Y., Liang, C., et al.: RootAgency: a digital signature-based root privilege management agency for cloud terminal devices. Inf. Sci. **444**, 36–50 (2018)
6. Wang, R., Enck, W., Reeves, D., et al.: EASEAndroid: automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 351–366 (2015)
7. Xue, B., Lu, L., Sikang, H., et al.: An isolated data encryption experiment method by utilizing baseband processors. In: Proceedings of the 2018 2nd International Conference on Management Engineering, Software Engineering and Service Sciences, pp. 176–181 (2018)
8. Yaghmour, K., Dagenais, M.R.: Measuring and characterizing system behavior using kernel-level event logging. In: 2000 USENIX Annual Technical Conference (USENIX ATC 2000) (2000)
9. Zanussi, T., Yaghmour, K., Wisniewski, R., et al.: relayfs: an efficient unified approach for transmitting data from kernel to user space. In: Linux Symposium, vol. 494 (2003)
10. Etsion, Y., Tsafrir, D., Kirkpatrick, S., et al.: Fine grained kernel logging with klogger: experience and insights. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 259–272 (2007)
11. Cohen, W.E.: Tuning programs with OProfile. Wide Open Maga. **1**, 53–62 (2004)