# Accountable Logging in Operating Systems

Lei Zeng, Yang Xiao*
Department of Computer Science,
The University of Alabama,
Tuscaloosa, AL 35487-0290 USA
Email: yangxiao@ieee.org
*Prof. Yang Xiao is the corresponding author

Hui Chen
Department of Mathematics and
Computer Science,
Virginia State University,
Petersburg, VA 23806 USA

***Abstract*—In this paper, study how to achieve accountable logging for operating system using the flow-net logging and its implementation in current operating system such as Linux. We demonstrate that the flow-net logging technique is capable of preserving event relationship. The performance for the flow-net logging implementation in Linux operation system is evaluated.**

***Index Terms*—Logging, Accountability, Operating System, Flow-net**

## I. Introduction

While computer system gains more complexity, security system breaches keep emerging. Many researches focus on countermeasures that detect security threats and recover the damages. Accountability contributes to these countermeasures. Accountability indicates that an entity with specific actions should be responsible for its actions [1-2]. One event could be traced back for the causes even when it was transpired [2]. Auditing or logging is a typical approach to achieve accountability [3]. Logging consists of accumulation and maintenance of records of activities in systems and networks. Logging includes recording system activities and network activities and maintaining the recorded data at the same time. Normally people refer to the recorded data as logging data, audit logs, audit trails, or logging. Auditing involves conducting reviews and examinations of system activities in order to ascertain the causes of one or more events and the responsibility of a system entity based on the logs.

Syslogd and syslog-ng are the syslog daemons implemented in Linux systems. Not only can they log data from their own machines, but they may also log data from other machines [4]. Syslogd consists of two programs: klogd and syslogd. Klogd manages the logged data from the kernel, and syslogd manages the logged data from application programs, and logged data is written in log files according to the configuration files. Also, there are several applications that have the ability to produce their own logs.

Mainly for debugging purpose, a modern operating system normally has limited generated logged data. Some security mechanisms such as are enforced to track most of the activities in the system. However, these logging records in log files are sorted by time the event generated. Therefore, the relationships between these events are lost.

When an event triggered logging, the event information will be buffered. Then the logging module will write out the contents of the logging buffer periodically. Therefore, the time-stamp of logged event is not necessarily the same as the time the event really happened. It is the time when the event is written out to the file.

When it comes to accountability, the logged events should be traced back in order to determine their causes. The relationships between these events are vital for tracing back security events. The trace back will be difficult if it only depends on the time-stamp to figure out what was really happening in the system.

In this paper, we propose operating system accountable logging using the flow-net methodology (ref. [5]) and we also implement it in current operating system such as Linux. The flow-net methodology not only logs the events, but also logs the relationships between the events. We further evaluate the performance of the logging implementation for operating system using the flow-net.

The organization of this paper is as follows. OS log is presented in Section II. Flow-net methodology implementation in Linux is introduced in Section III. Then performance evaluation is provided in Section IV. Finally we conclude this paper in Section V.

## II. Operating System Log

In this section, we will present background information for the current logging systems, such as SE Linux and Linux system logs, and analyze their generated logging files. Then we will propose our logging methodology to address the current logging system issues.

### *A. SE Linux logging*

As a mechanism adopting Linux Security Modules (LSM) in the Linux kernel, Security-Enhanced Linux (SELinux) supports access control policies, including mandatory access controls (MACs). Although SELinux does not come with Linux, it is provided with modifications applied to kernels of Unix-like operating systems including BSD (Berkeley Software Distribution) and Linux.

In Linux, there is a system log module that maintains all the buffered log events and writes out these buffers. The data generated by SELinux is also manipulated by system log module and therefore is part of the system logs.

If `auditd`, the daemon of the auditing system of Linux, is running background, SELinux denials are saved in the log/audit file. `/var/log/audit/audit.log` is the default log/audit file. If `auditd` is not running, `/var/log/messages` is used to log AVC (Access Vector Cache) denials. Normally /var/log/audit/audit.log is the log/audit file to save SELinux logsif `auditd` is running. An

example of AVC denial is shown as follows and explained in Table 1 [6].

```
avc: denied { read } for pid=3002 comm="httpd" name="index.html"
dev=hda3 ino=32004 scontext=user_u:system_r:httpd_t:s0
tcontext=system_u:object_r:tmp_t:s0 tclass=file
```

Table 1 Analysis of an SELinux logging record [6]

| Message | Description |
|---|---|
| Avc: denied | An operation has been denied |
| {read} | This operation required the read permission |
| Pid=3002 | The process with Pid 3002 executed the operation |
| Comm.= "httpd" | The process was an instance of the httpd program |
| name="index.html" | The target object was named index.html |
| dev=hda3 | The device hosting the target object was a real disk, named hda3 |
| ino=32004 | The object was identified by the inode number 32004 |
| scontext=user_u:system_r:httpd_t:s0 | This is the security context of the process who executed the operation. It contains user, role, type and security level |
| tcontext=system_u:object_r:tmp_t:s0 | It is the is target object's security context. |
| tclass=file | This means that the target object is a file. |

### B. *Linux system logging*

Syslogd and syslog-ng are the syslog daemons implemented in Linux systems. Not only can they log data from their own machines, but they may also log data from other machines [4]. Syslogd consists of two programs: klogd and syslogd. Klogd manages the logged data from the kernel, and syslogd manages the logged data from application programs, and logged data is written in log files according to the configuration files. Also, there are several applications that have the ability to produce their own logs.

The following example is a logging record in /var/log/syslog file.

```
Feb 20 22:34:49 forrestgump-OptiPlex-755 rtkit-daemon[12467]:
Successfully called chroot.
```

The first element of the logging record is timestamp, followed by a user. The application that triggers the event and the result of the event are logged as well. In this example record, process rtkit-daemon with process id 12467 successfully called chroot at 22:34:49 on Feb 20.

In essence, all the log files only contain the events with timestamps and the relationships between these events are not recorded.

## III. FLOW-NET IMPLEMENTATION IN LINUX

### A. *Flow-net methodology*

The flow-net methodology is illustrated in Fig. 1. In Fig. 1, user A logged in, entered a directory, opened File B, read File B, closed File B, and logged out. User D logged in and created File B. The logged information includes three flows: User A, File B, and User D. The flow beginning from User D obtains three events: logging in, creating File B and logging out.

The flow-net records not only contain events information, but also contain relationships between these events. The generated data is really useful for tracing back.

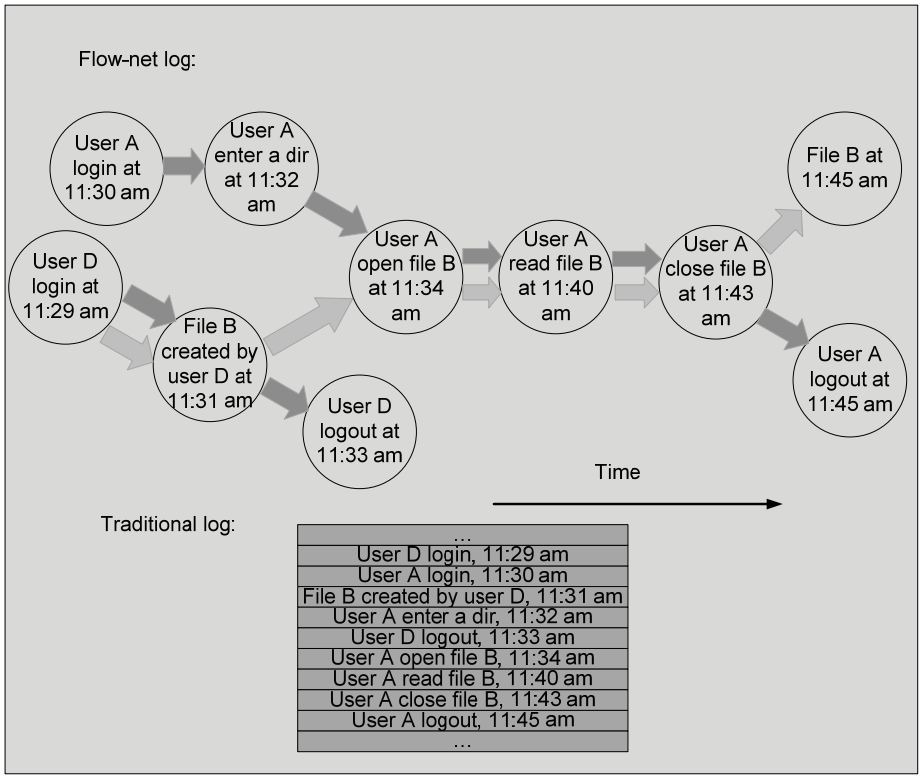


Fig. 1 Flow-net vs. traditional log

For traditional log, nine events in Fig. 1 were logged with their time-stamps in the generated log file. Compared to Flow-net log, the relationship between different events may only be recovered based on the time-stamps and our best knowledge. However, the relationships between different flow-net events are built in the first place when the log generated.

Based on the previous analysis, we might ask: does traditional log lose any information compared with Flow-net log? When we use traditional log events to recover Flow-net log, it might be possible that the relationships between events may not be rebuilt, or may be rebuilt wrongly, based on the time-stamps as well as our knowledge. We can only guess their causes and effects when it comes to accountablilty, which is not accurate for most cases. The accuracy of these guessings are entirely based on auditors' knowledge and experience. Therefore, traditional log cannot address accountability problems.

Besides, if we can recover the relationships correctly, what is the time complexity to build Flow-net log using traditional log. Given the *n* recorded log events, the time complexity to build Flow-net log should be *n!*. At first, we need to take one event at the beginning and determine its relation with the other (*n-1*) events. Subsequently, take out the second event and determine its relation with the remaining (*n-2*) events.At last there is only one event left and rebuilding process is done.

For accountability, the logged events should be traced back in order to determine their causes. The relationships between these events are vital for tracing back security events. The trace back will be difficult if it only depends on the timestamp to figure out what was really happening in the system. Besides, traditional log only log limited security related events with the purpose of debugging, which is not suffice to answer accountability problems. For instance, some non security related events might not be logged for the traditional log.

### *B. Implementation of flow-net*

In order to record the events occurred in the system, the hooks added by SELinux [7] are used to capture the events for simplicity. Since Flow-net structure is built in real time, updating Flow-net structure whenever an event occurs is necessary. Therefore we can modify the event-capturing part of SE Linux and add some logic to build the cross-reference structure. Besides, we should maintain the entries to all entities in Fig. 1. For instance, user A, File B, and use D are the entities in the system. In this case, we need to consider another problem: when we build a new entry for an entity? Note that all files and users are entities. If we maintain the cross-reference for all entities in the system, it will dramatically slow the system and will not have the capability of being understood. We can intuitively think that if an entity is created for the first time, we need to create a new entry for this entity. We don't need to create entries for the files created when the system is installed. When a user login the system, we need to create an entry for the user because the user may invoke many actions in the system and cause lots of logging records generated. But what if users read or write to a file created when the system is installing, which happens very often. For instance, user A makes some modifications to /etc/profile to customize the system. Our solution is to create a new entry for an entity when an event involves this entity. Otherwise, we don't need to create the entry. But, if an entity involves tremendous entities, the performance of the system will be jeopardized. For example, Firefox can create many cache files and other temporary files to speed itself. When these things occur, we need to dramatically increase our cross-reference structure.

We can capture the events such as reading and writing a file in kernel. In order to test whether our scheme works, we only log the read and write events in the system and build a flow-net.

In linux-2.6/fs/sysfs/bin.c:

```
    Static ssize_t write (struct file *file, const char _user *userbuf, size_t
bytes, loff_t *off)
    Add the following codes:
    #include <linux/cred.h>
    #include<asm/current.h>
    #include<linux/sched.h>

    Static char logEventBuf[1024];
    Int uid=current->real_cred->uid;
    Int euid=current->real_cred->euid;
    Struct dentry *dentry=file->f_path.dentry;
    if  (dentry->d_inode->i_iflog==1)
        printk(KERN_INFO "%s want to write to log file
%s\n",current_euid(),dentry->d_name);
    logEventBuf="write";
    ….
```

At this point, we captured the write event and logged it in logEventBuf[] in kernel space. The next step is to forward the data in kernel space to user space and build the Flow-net structure.

In order to build a more complicated flow, we need to capture more events. For example, login event is critical. How can we know the login event happens is also a problem. Because when the machine finishes booting, a getty process is invoked and it will invoke another process login. Therefore, a login event happens whenever a login process is invoked by exec system call. After the init process respawns the getty process, the login event has ended. Besides all file—related operation is very important and worth logged. For example, chmod, chown, and so on.

When we capture an event happening, we first check whether the involved entities are new. If these entities are new, we build a new entry for every new entity and build the link from each involved entry to the captured event. If the entities already have their entries, we traverse their event list and add the newly captured events at the end of the event list. At the same time, we have a user space program like syslog daemon to write the cross reference to a file on the disk.

During our implementation, we use an array to record these link structures as follows.

```
struct time_m {
      int sec;
      int min;
     int hour;
```

```
int mday;
 int mon;
long year;
int wday;
int tyday;};
```

In Flow-net model, any file objects and users have flows, as shown in Fig. 2. There are three flows, user A, user D and file B in the figure with their own flow entries.

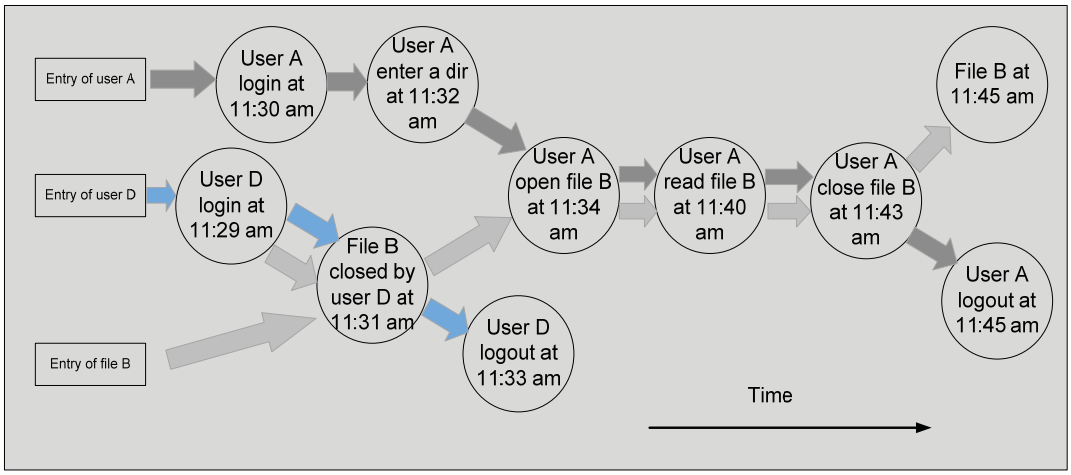


Fig. 2 Flow-net model

### C. *Communication between kernel space and user space*

After we have built the cross-reference structure in the array in kernel, we need a user space program to read the array out to write to the disk. There are several ways available to communicate with a user space program in kernel, such as named pipe, copy_to_user, copy_from_user and netlink socket.

In these three approaches, the named pipe is FIFO. Because we only desire to use a program in user space to write the data in array to a file, maybe there is no close relation to FIFO. For either the copy_to_user approach or copy_from_user approach, it will be suitable for our implementation due to the fact that we use an array to store those cross-reference structures. copy_from_user and copy_to_user can easily copy a part of memory in kernel to user space. For the netlink approach, a special inter-process communication (IPC) called Netlink socket is for the purpose of two-way communications between user-space processes and the kernel [4].

In essence, we can use copy_to_user and netlink socket to write the data in kernel to user space.

## IV. Performance Evaluation

When we have implemented the flow-net model in the Linux, we now desire to evaluate the system performance and check the overhead from the logging. The test scenario is that we first open a new file and write a given volume of data to the file. Subsequently, we delete the file and persist on doing the same thing for 2000 times. First we run the test program in the old default kernel and count the average running time. Next, we run the same test program in the kernel in which we implemented the flow-net model and count the average running time. Then, we may roughly estimate the overhead in the new kernel.

The program running in the old kernel took roughly 283 seconds. The program running in the new kernel took roughly 404 seconds. The performance is affected by (404-283)/283=43%.

Due to the fact that this time is for the overall running time, we cannot clearly know whether it uses the CPU cycle during the executing procedure. In Linux, we have a times () function storing current process times in struct tms:

```
struct tms {
clock_t user_time
clock_t system_time;
clock_t child_user_time;
clock_t child_system_stime;
```

Therefore, we may use this structure to know more about the process time.

In the original system, the process took 14998(time ticks) for user time and 13426 for system time. In the new system, the process took 23695 for user time and 17254 for system time, illustrated in Table 2. Due to the logging logic in kernel, the system time is increased. In the kernel, we modified the read and write functions and everything will affect the system time. Due to the fact that the user time also increased because we launched a user space-logging program in order to write the buffer in kernel to files when we launch the test program. The two programs that run simultaneous are all IO-intensive and will wait for each other to do IO. Maybe their waiting for IO will increase the user time.

Table 2 System performance of flow-net implementation

| | User time | System time |
|---|---|---|
| Original system | 14998 | 13426 |
| System with flow-net Implementation | 23695 | 17254 |

When we have implemented the accountable log model in the Linux, we then want to evaluate the system performance and check the overhead from the logging scheme. The test scenario is designed as follows: first, open a new file, write some random numbers to the file, and then delete the file. We call this set of actions “1 time action”. We record different running times for different action frequencies. We also run the test program in the old default kernel and count the average running time. Then, we run the same test program in the kernel in which we implemented the flow-net model and count the

average running time. Therefore, we may roughly get the overhead in the new kernel. The implementation environment is described in Table 3.

Table 3 Implementation environment

| Component | Description |
|---|---|
| system | Ubuntu 9.10 |
| bus | 0G8310 |
| memory | 512MiB DIMM SDRAM Synchronous 533*2 |
| processor | Intel(R) Pentium(R) 4 CPU 3.00GHz |
| storage | 82801FB/FW (ICH6/ICH6W) SATA Cont, 40GB WDC WD400BD-75JM |

We can see the performance result in our previous work [4]. Also, we may notice that the system performance varied by the number amount of data written to the disk. If we were to write more data to the disk in one action, the performance can be improved.

We may notice that the system performance is affected because we add some protection policy before the write () and delete () system calls to forbid delete and write actions to some protected files. We may see the performance from Fig. 3. Therefore, we can notice from the experiment that because of the SE Linux policy, the system becomes time increased. In the old kernel, we only need to run the test program. We can also see that the system performance varied by the number amount of data written to the disk. If we write more data to the disk in one action, the performance can be improved, as showed shown in Fig. 3.

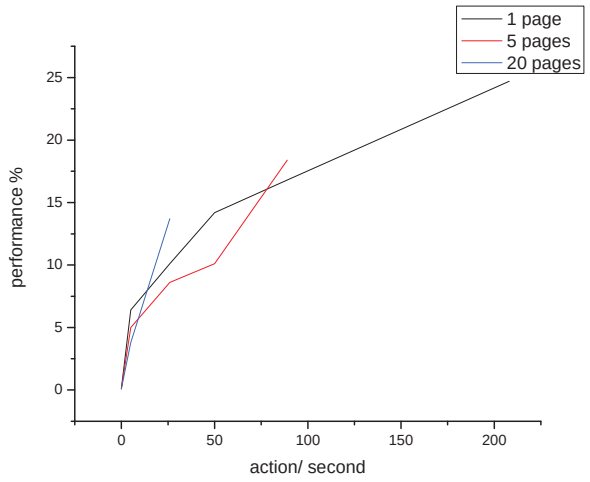


Fig. 3 Performance evaluation (flow-net model)

We may also see that the performance is improved compared to the evaluation in our previous work [4]. In that study, we used copy_to_user to deliver messages from the kernel space to the user space. But in this study, we adopt netlink socket to achieve the goal. Why is the performance improved? Netlink as any socket application programming interface (API) uses a queue to save messages to smooth the socket traffic. When a message of netlink is sent, the reception handler of the receiver was invoked after the message is queued in the netlink queue of the receiver. The queued message may be either processed immediately in the context of the reception handler or left in the queue to process it later in another context. Therefore, Netlink is asynchronous. System calls other than netlink are synchronous so that when a message passing from the user space to the kernel space, the scheduling granularity of the kernel cab be influenced due to the long delay of message to be handled.

## V. Conclusion

We have implemented the flow-net model to make the log accountable using the netlink approach and also written on our policy use the platform provided by SE Linux. We also evaluated the performance.

## Acknowledgement

This work was supported partially by the Natural Science Foundation of China under grant #61374200.

## References

[1] J. Mirkovic and P. Reiher, “Building Accountability into the Future Internet,” Proceedings of the IEEE ICNP Workshop on Secure Network Protocols (NPSec), 2008.

[2] Y. Xiao, “Accountability for Wireless LANs, Ad Hoc Networks, and Wireless Mesh Networks,” IEEE Communications Magazine, Vol. 46, No. 4, Apr. 2008, pp. 116-126.

[3] K. Kent and M. Souppaya, “Guide to Computer Security Log Management: Recommendations of the National Institute of Standards and Technology,” NIST Special Publication 800-92, Sep.2006.

[4] L. Zeng, H. Chen, and Y. Xiao, "Accountable Administration and Implementation in Operating Systems," Proc. of IEEE GLOBECOM 2011

[5] Y. Xiao, "Flow-Net Methodology for Accountability in Wireless Networks," IEEE Network, Vol. 23, No. 5, Sept./Oct. 2009, pp. 30-37.

[6] “The Debian Administrator's Handbook,” available: http://debian-handbook.info/browse/wheezy/sect.selinux.html

[7] “Security-Enhanced Linux,” http://en.wikipedia.org/wiki/Security-Enhanced_Linux