IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SECURITY AND PRIVACY IN EMERGING DECENTRALIZED COMMUNICATION ENVIRONMENTS

Received July 31, 2019, accepted August 20, 2019, date of publication August 26, 2019, date of current version September 25, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2937637

Prototyping Flow-Net Logging for Accountability Management in Linux Operating Systems

YANG XIAO[1,2], (Senior Member, IEEE), LEI ZENG[2], HUI CHEN[3,4], (Senior Member, IEEE), AND TIESHAN LI[1], (Senior Member, IEEE)

[1]Navigation College, Dalian Maritime University, Dalian 116026, China
[2]Department of Computer Science, The University of Alabama, Tuscaloosa, AL 35487-0290, USA
[3]Department of Computer and Information Science, Brooklyn College, City University of New York, Brooklyn, NY 11210, USA
[4]Graduate Center, City University of New York, New York, NY 10016, USA

Corresponding author: Yang Xiao (yangxiao@ieee.org)

This work was supported in part by the National Natural Science Foundation of China under Grant 51939001, Grant 61976033, Grant U1813203, Grant 61803064, and Grant 61751202, and in part by the Science and Technology Innovation Funds of Dalian under Grant 2018J11CY022.

ABSTRACT Accountability in conjunction with preventative countermeasures is necessary to satisfy the needs of real-world computer security. A common method to achieve accountability is via logging and auditing. To achieve better accountability, a logging system should be capable of capturing activities as well as the relationships among the activities in a computer system or network. Existing logging techniques record activity events in isolation and rely on attributes and time stamps of the logged events to establish their relationships, and this approach leads to probable loss of event relationships among large and complex logs and a confusion during auditing. Prior works have indicated that flow-net is effective in addressing this problem by organizing events in a direct acyclic graph and preserving event relationships during logging. In this work, we provide a prototypical design and implementation of a flow-net accountable logging framework in the Linux operating system. Particularly, it can be applied to Internet of Things (IoTs) with Android Operating Systems. We measure the performance overhead introduced by the flow-net logging prototype via experiments in Linux. The results indicate that the flow-net prototype only introduces a small overhead when compared with existing logging methods. In addition, we show by examples enforcement of accountability policies in the flow-net logging framework and its performance overhead. This work thus constitutes a further step to advance flow-net in addressing accountability in computer systems and networks.

INDEX TERMS Computer security, accountability, logging, auditing, flow-net, IoT.

I. INTRODUCTION

The common approach to ensure security is via preventative countermeasures, such as, access control and intrusion detection that attempt to prevent secure policy violations from security attacks [1]. Another approach is via accountability [1]–[8] — an after-the-fact examination of system activities to discern whether a violation of a security policy has occurred, and what has caused or what chain of events has led to the policy violation [1], [2], [9], [10].

Real-world experiences from dealing with security indicate that accountability is not only complement to preventative countermeasures, but also necessary.

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaojiang Du.

First, some attacks, e.g., Distributed Denial-Of-Service attacks are non-differentiable from normal activities, which makes it a challenge to design effective intrusion detection mechanisms to counter them [11], [12]. Second, preventative countermeasures requires often beforehand knowing what constitutes an attack, which is difficult for zero-day attacks [13], [14]. Preventative countermeasures often either tempt loose access control in favor of functionality and features, lower expenses, or convenience of using software or systems, or invite over-stringent access control in favor of policy compliance or out of paranoia, in particular, when users' voices are unheard. Aside from these, as networked computers from big to small have become ubiquitous and online privacy becomes an increasingly important concern to individuals and societies, we have again learned that preventative countermeasures are inherently ineffective in dealing

with online privacy because "*information is easily copied*" and "*aggregated and automated correlations and inferences across multiple databases uncover information even when it is not revealed explicitly*" [1], and it is no easy feat if not outright a futile effort to deploy access controls that work in concert in multiple independent administrative sites to prevent online privacy violation from happening.

Complement to preventative countermeasures, accountability isn't designed with the sole emphasis on preventing an attack from happening or detecting an ongoing attack. Instead, it is an approach of reward/penalty, e.g., at the event an attack is happening or happened, we want to discern responsible parties for the attack and apply a remedy afterward — this is consistent with how real world *physical* security works, which "*is not about provide perfect defenses against determined attackers, instead, it is about value, locks, and punishment*" [15]. Preventative countermeasure and accountability work in tandem is the approach to realize this vision in deal with real-work *computer* security. As such, accountability is becoming increasingly a sought-for approach for security in addition to preventative countermeasures.

A common mechanism to achieve accountability is through logging and auditing [1]. Logging is the process of recording records of system and network activities. The collections of the records are often referred to as logs, logging data, audit trails, or audit logs. Auditing involves conducting reviews and examinations of system activities based on the logs, and the purpose is to ascertain 1) whether there is a security policy violation, and 2) if any, whether we can identify the responsible system entity or entities from a number of log events that are leading to the policy violation.

There are technical challenges to apply logging and auditing effectively to achieve accountability. First, it requires security policy maker and system administrators to have a foresight on what events to log; otherwise, the auditing process would lack adequate data to go on. For instance, Trusted Solaris has the capability to define a set of predefined events to collect in logging and a list of auditing queries to answer with these logged events [16]. Second, the auditing process must be able to infer adequate information to answer accountability questions.

To address these challenges, researchers have examined approaches that records and organize logs as direct acyclic graphs (DAG) and recognized the advantages of the DAG approaches [17]–[20]. Among these DAG approaches is flow-net [19]. This paper introduces a prototype of a flow-net logging framework to support accountability.

Flow-net as a logging mechanism builds comprehensive logs to track events. It maintains explicitly the relationships of events in the logs, and the events and the relationships form a DAG [19], [20]. Advantages of this flow-net approach becomes apparent when we compare it with traditional approaches, i.e., logging mechanisms that record log events individually without explicitly considering relationships among the events. In the traditional approaches, we must infer the relationships of the events from the values of the attributes of the logged events during auditing. The relationships inferred may not be reliable, and the inference process may be time consuming.

Prior works have proposed flow-net to achieve accountability in computer networks and systems, and these works have laid a theoretical foundation and suggested application scenarios for accountability via flow-net [19]–[22]. These works began the introduction of the flow-net concept in computer networks and systems [5], [19]. Although accountability as a concept is well perceived from early computer system design [2], it isn't trivial to define accountability policy including the types of entities we should identify as a cause of a policy violation, and the level of accountability we desire to achieve, e.g., how confident we are when we identify the root cause of the policy violation. As such, in practice, accountability policy is often manifested as a logging and auditing policy that include types of events to log and set of auditing rules to detect policy violation [16], [23]. Xiao et al. and Fu et al. proposed the concepts of P-Accountability and Q-Accountable logging in the flow-net framework to define accountability requirement qualitatively or quantitatively [24], [25].

Given these prior works, our aims are at three interconnected problems primarily at addressing practical concerns of achieving accountability using flow-net. We are 1) to provide a proof-of-concept design and implementation of the proposed flow-net logging framework in Linux operating systems, 2) to assess the performance overhead introduced by the flow-net logging; and 3) to demonstrate the ability of the framework to enforce accountabiliy policies. We summarize the contributions of this paper as follows:

- We identify improvements for traditional logging techniques in modern operating system.
- We introduce a logging framework called flow-net to eliminate identified drawbacks incurred by the traditional logging techniques.
- We provide a proof-of-concept design and implementation of the flow-net logging framework in Linux. Particularly, it can be applied to Internet of Things (IoTs) with Android Operating Systems.
- We evaluate the performance overhead introduced by the flow-net logging implementation.
- Finally, we demonstrate by examples how we express accountability policies and how much the performance overhead introduced to enforce these policies.

The organization of this paper is as follows. Section II provides a brief review of Linux logging, Linux auditing, and SELinux on which we built the prototypical implementation of the flow-net logging framework. We provide an overview of flow-net, and compare traditional and flow-net logging techniques, and describe the design and implementation of the flow-net framework in Linux in Section III. Section IV presents an evaluation of the performance overhead introduced by the flow-net logging implementation and discusses

TABLE 1. Linux kernel logging.

Source	Access Mechanism	Explanation
The `printk()` ring buffer	`syslog`[a]	the kernel `syslog()` system call, which reads or clears the ring buffer
	`/proc/kmsg`	Provide a process with read-only access to the ring buffer
	`/dev/kmsg`	Provide processes with both read and write access to the ring buffer

[a]This refers to the `syslog` system call whose `glibc` wrapper function is `klogctl` that is different from the `glibc` function `syslog` that interacts with the `syslogd` service

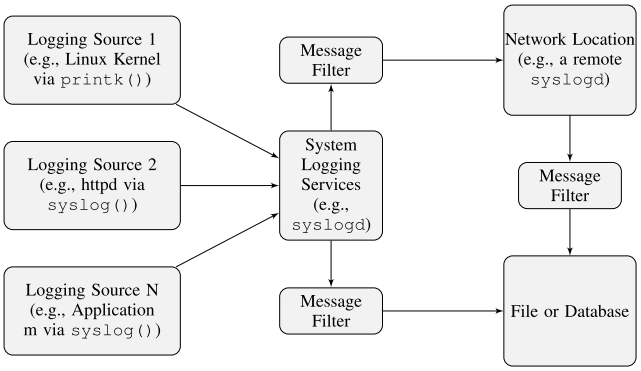


FIGURE 1. Logging overview.

accountabiliy policy examples and their enforcement cost. Finally, we conclude this paper and presents the future work in Section V.

II. BACKGROUND

Relevant to the prototypical design and implementation of the Flow-net logging framework in Linux are the Security-Enhanced Linux (SELinux), Linux logging subsystems, and Linux auditing framework. We provide a brief overview of these three. Note that we discuss Linux in general, but it can be applied to Internet of Things (IoTs) with Android Operating Systems [26].

A. SYSTEM LOGGING IN LINUX

Figure 1 is a high-level overview of logging in Linux systems. Linux logging consists of two major components. First, it is logging in the Linux kernel. The Linux kernel maintains buffers to store log messages. These buffers are typically cyclic buffers (also called ring buffers). Kernel may overwrite the log messages in the ring buffers even if they are not consumed, i.e., transferred to user space buffers or to user space processes. Table 1 provides an overview of the kernel logging access mechanisms. Second, a Linux system has logging services interacting with the Linux kernel. These logging services maintain the user space logging buffers and consume logging messages in the user space buffers. There are a variety of user space logging services in Linux systems. Popular ones include `syslog`[2], `rsyslog`[3], and `syslog-ng`[4]. Besides additional enhancements, these logging services generally implement the syslog protocol [27], and route the log messages arriving in the user space buffer to destinations like the system console, or mail the messages to a specific user, or save the messages in a log file or a database, or pass the messages to another syslog service instance.

[2]See http://www.infodrom.org/projects/sysklogd/
[3]See https://github.com/rsyslog/rsyslog
[4]See https://github.com/balabit/syslog-ng

B. OVERVIEW OF SELINUX

SELinux was initially designed to realize Mandatory Access Control (MAC). It has evolved to support one or combination of multiple security models including Multi-Level Security (MLS), Multi-Category Security (MCS), Type Security, Identity-Based Access Control, and Role-Based Access Control [28], [29]. It consists of two major components, a policy language and a policy enforcement component.

A SELinux policy consists of essentially a set of rules called Access Vector (AV) rules or Access Vector Cache (AVC) rules. An AV rule is a 5-tuple, i.e., `(rule type, source, target, class, permission set)`. Of the 5-tuple, the pair of `(source, target)` serves as an index to the rule, i.e., the Access Vector of the rule. Figure 2(a) is an example of AV rules. The interpretation of these rules is in Figure 2(b) where we represent the rules in an Access Control Matrix-like format. In this context, we can consider an AV rule as a 2-tuple (a pair), i.e., a pair of `(AV, rights-set)` where `AV` is `(source, target)` and `rights-set (class, permission set, rule type)`.

Its policy enforcement component is a Linux Security Module (LSM) [31]. The component consists of Object Manager, Access Vector Cache (AVC), and Security Server. LSM has a number of hook function pointers. The Linux system bootstrp process initializes SELinux by pointing the LSM hook function pointers to SELinux routines. Figure 3 illustrates the relationships of these components and how SELinux enforces its security policies [28], [29], [32]. For instance, when a process requests to access a file, we first form an AV by obtaining the source type from the process and the target type from the file, and send it to the Object Manager. If the Object Manager does not find the AV, the file is not subject to MAC of SELinux and the access control falls back to traditional Linux Discretionary Access Control (DAC). If the Object Manager finds the AV, SELinux uses the AV to look up the AV rules from the AVC. If the Object Manager does not find an AV rule in the AVC, SELinux

```
allow initrc_t acct_exec_t:file {getattr read execute};
allow kernel_t filesystem_type:filesystem mount;
allow files_unconfined_t file_type:{file chr_file} ~execmod;
```

(a) Three SELinux AV rules, where the rule types are all `allow`, the sources of the three rules are `initrc_t`, `kernel_t`, and `files_unconfined_t`, targets are `acct_exec_t`, `filesystem_type`, and `file_type`, the classes are `file`, `filesystem`, and `{file chr_file}`, and the permission sets are `{getattr read execute}`, `mount`, and `~execmod` (`~execmod` means any permissions other than `execmod`). The SELinux Wiki page provides more in-depth discussion about this example [30].

Permission / Class / AV	file	filesystem	chr_file
(initrc_t, acct_exec_t)	{getattr read execute}		
(kernel_t, filesystem_type)		mount	
(files_unconfined_t, file_type)	~execmod		~execmod

(b) The access control matrix-like representation of the 3 AV rules. Since the 3 rules are 3 "`allow`" rules, we call this the `allow` access control matrix of the 3 rules. Whenever a process attempts to access an object, SELinux determines whether the process has the permission by using the process's type (e.g., `initrc_t`) and the object's type (e.g., `access_exec_t`) to look up the permissions set.

FIGURE 2. 3 SELinux AV rules and their access control matrix-like representation.

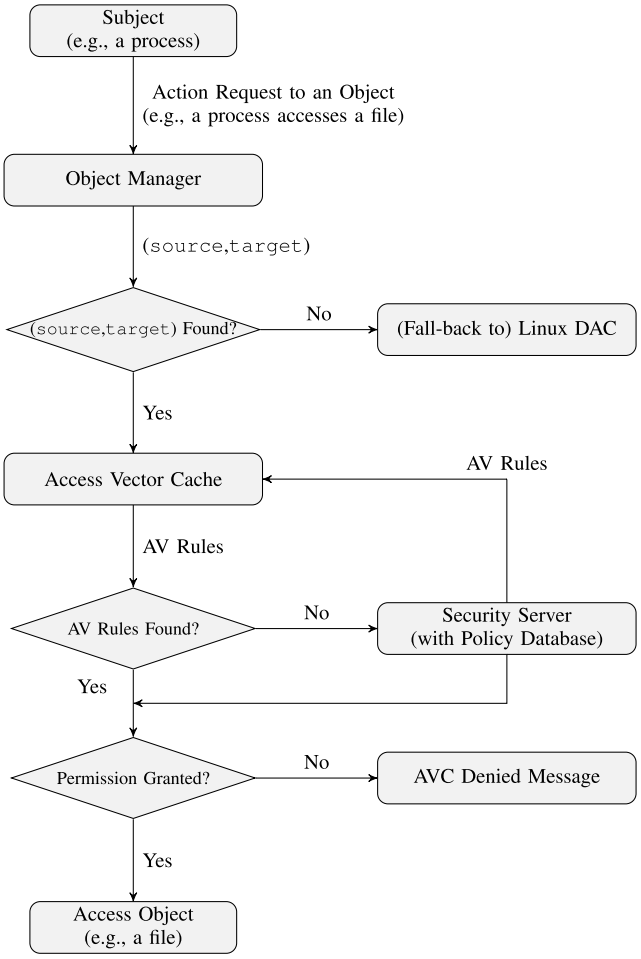


FIGURE 3. SELinux policy enforcement logic [28], [29], [32].

queries the Security Server and retrieves the AV rules using the AV. SELinx then caches these AV rules in the AVC. With the AV rules, SELinux determines whether it grants or denies the process's access to the file. If it denies the access, it forms an *AVC denied message*, and as we shall see, logs the message via the Linux audit framework.

Table 2 is an example of an AVC denied message that corresponds to an SELinux policy violation. In this example, we also see examples of SELiux Security Context that consists of `user`, `role`, `type`, and the optional field of `range`. An object controlled by a SELinux policy, such as a file has an association with a SELinux `type`. The `type` of a process, as we have described in the above in effect specifies a protection domain and the `type` of an object specifies a permissions-set on the object, as in Figure 2. SELinux associates an SELinux `user` with one or more SELinux `roles`, and associates each `role` with one or more SELinux `types` to which the `roles` have access. The `range` field is optional. It is present when the policy supports MCS or MLS, and specifies a *level* of MLS (e.g., level s0) and a *category* of MCS. Linux loads a SELinux policy when it boots. It establishes the associations between the SELinux `types` and Linux users and files that are under the policy's control. The association process is in effect SELinux "labeling". The AVC denied message in Table 2 essentially states that the permissions-set retrieved by the AV `(http_t, tmp_t)` does not contain a `read` permission.

C. LINUX AUDIT SYSTEM AND LOGGING

Linux audit system is in fact a logging component for the Linux kernel and provides a service to record audit events in logs and tools to query the logs [33]. It consists of a kernel component and a user space component. A user can instrument the kernel to specify events to log, such as, at the entry or at the exit of a system call or both. The audit subsystem writes these events to a kernel buffer and transferred to the user space component via the `netlink` socket, a kernel interface for efficient access of kernel data in user space [34]. Linux audit subsystem interactions with LSM and SELinux, e.g., SELinux writes AVC denied messages via Linux audit framework, which in turn writes the messages to a file or to a syslog service.

TABLE 2. An SELinux logging record.

Record:

```
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
```

Message Component	Description
`avc: denied`	a denied operation
`{ read }`	the operation access attribute is `read`
`pid=3002`	the process id is 3002
`comm.= "httpd"`	the httpd program's instance
`name="index.html"`	index.html is the name of the target object
`dev=hda3`	hda3 is a disk where the target object is on hda3.
`ino=32004`	32004 is the inode number of the object
`scontext= user_u:system_r:httpd_t:s0`	the security context (including user, role, type and security level) of the process who runs the operation
`tcontext= system_u:object_r:tmp_t:s0`	The security context of the target object
`tclass=file`	A file is the target object

III. FLOW-NET AND FLOW-NET LOGGING IMPLEMENTATION

We present a prototypical design and implementation of the flow-net based logging.

A. FLOW-NET

A flow-net is a log recorded in a directed acyclic graph (DAG) that maintains events and their relationships as part of the log [19]. Inside a system, there are entities, such as a file, a user, a system call, and a process. Each entity has activities associated to it, either performed by it or performed on it. We order all of the activities associated with an entity in a temporal order from its creation to the present time or to its termination. These activities form the activity *flow* of the entity. Each activity involves typically two entities, i.e., a subject and an object, and the subject and the object's flows thus intersect with each other. The flows in the system form a network called a flow-net, a DAG where logging events representing the activities are vertices (or nodes) and a temporal relationship between two closest events in a flow becomes an edge.

For example, consider a subject, e.g., a user. All of the events involved with the user from the time when a system administrator creates the user account in the system to the present time or to the time when the system administrator removes the user account from the system forms a flow along with time, called the user's flow. Consider an object, e.g., a file. All of the events involved with the file from the time when a process creates the file to the present time or to the time when a process deletes the file are a flow along with time, called the file's flow. An event such as that the user opens the file is on both the flow of the user and that of the file. The two flows thus intersects at this common event. All of the flows in a system forms a flow-net.

Figure 4 includes an example of a flow-net log and illustrates this concept. The log shows that 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. These events form 3 flows in the flow-net log. For convenience, we call them Flow User A, Flow File B, and Flow User D. Flow User A records actions the subject, i.e., User A performed, Flow File B actions performed on the object, i.e., on File B, and Flow User D actions User D performed.

To achieve accountability, e.g., to determine the causes of an event of interest, e.g., an unauthorized access of a user file, we trace back to events leading to the event of interest, and infer the cause of the unauthorized access from the preceding events. The relationships among events are essential for achieving this. A requirement to allow back-tracing is to know first what events relate to the unauthorized access event, and the temporal order of these related events, i.e., which occur earlier, and which later. Figure 4 also include a snapshot of a ''traditional log''. In Figure 4, the traditional log has 9 events. To determine the set of related events, we must examine the values of the attributes of the events, such as, whether two events share an attribute and the values are identical. To determine the temporal order of these events, we rely on the time stamps of these events.

The back-tracing can be difficult in the traditional logs, if we only depend on the time stamps to figure out what happened. These time stamps can be either the time when the events occur or the time when the system records the events. The consistency of the time stamps can be a problem. In addition, due to concurrency and asynchronous I/O in the system, the order of event arrivals in a log alone does not necessarily establish any causality or correlation of the events because the events may arrive out of order and may occur concurrently; the correct temporal order is essential to achieve accountability accurately.

In addition, given the set of events in the traditional logs, an auditor must use an algorithm to infer the relationships among the events using event types, event attributes, and the values of the attributes. The accuracy of the relationship inferred depends on the auditor's knowledge, skills, and experience or on the sophistication of the algorithm used.

The above back-tracking process in the traditional logs implies the assumption that sufficient preceding events are in the log for us to analyze during the auditing process.

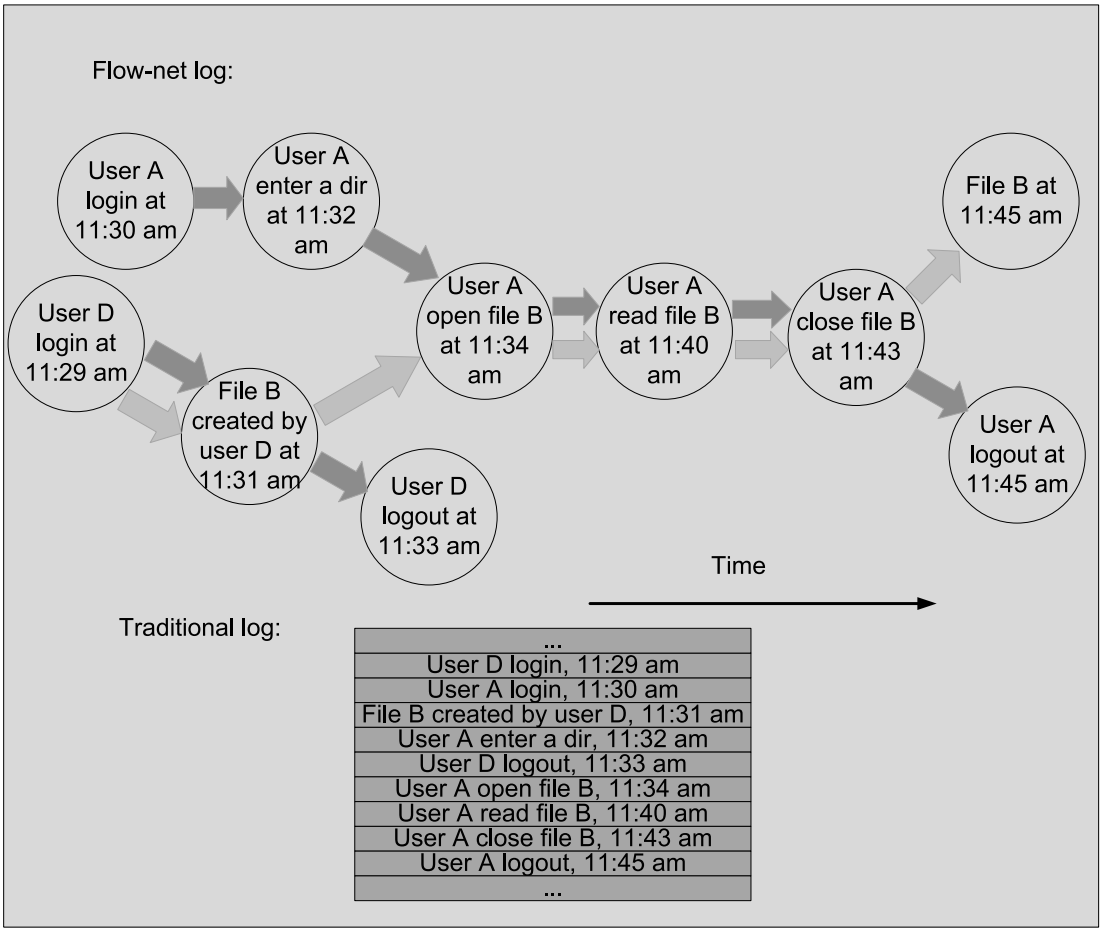


FIGURE 4. Flow-net log vs. traditional log.

To record log events, system administrators must configure the system beforehand. Since the purpose of the log is to support auditing, it requires the administrators to have a foresight on what events are necessary to support auditing process. It is possible the system has never logged necessary events to support effective auditing process to answer queries of interest. Furthermore, as shown in Figure 1 some events may not be present in the log files since they were 1) dispatched to other network locations, 2) consumed by another process, and 3) overwritten by later event data before the system writes them out to log files or databases.

Next is to consider the time complexity to build complete relationships from the events in a log if the relationships are not explicitly recorded, i.e., given that we want use a traditional log to build the relationships of the events, how much time it will take. Consider a traditional log of m events. First, we begin with one event and determine its relationship with all the rest $(m-1)$ events. Second, we choose the second event and determine its relationship with all the rest $(m-2)$ records. So on so forth, finally, we choose the last event and finish building the relationships. The total steps are therefore $(m-1)\cdot(m-2)\ldots 1 = m!$.

Contrast to the traditional log, a flow-net log contains both events information and relationships among these events. This has two advantages. First, the relationships among events in the flow-net log allow us to recover activity flows because this is how the system builds a flow-net log. This is useful to trace events in the log and to discover the root causes of some important events, such as, an unauthorized access to a user's file. Second, during the process of recording flows, the flow-net logging must record necessary events along the flow regardless whether an administrator configures it or not, which ensures that the system must record necessary events otherwise it wouldn't be able to build the flow-net log. Third, to support auditing, we only sequentially examine events along flows, which is in linear time. Therefore, flow-net logging is a better approach than traditional logging to support logging and auditing for accountability.

B. EVENT TYPE AND STRUCTURE

1) COMMON EVENTS IN LINUX

In order to record user activities in flow-net in Linux, we will explore major events from Linux system booting to powering

off. Understanding these is necessary to help us design data structures to build flow-net logs.

- Linux boot. Linux boot is a common event as we must boot the system before we use it, and it is the time when the system initializes a flow-net.
- Linux reboot. From time to time, a reboot is necessary, such as, a kernel upgrade. Reboot is also a common event, and it is the time when the system reinitializes a flow-net.
- User login. When Linux boot finishes, the system will authenticate and authorize a user to run a process. In an interactive system, this often leads to a user login event. If a whole-system flow-net is not of interest but a user is, it is the time we initialize a user specific flow-net.
- Switch user. When a user logs in the system, the user can sometimes switch to another user if authorized to do so, which is the time that the user switches protection domains.
- Opening and closing shell. Linux users use system shells to interacts with the system. Opening and closing shell are also important and common events.
- Shell commands. Users in Linux can launch programs through shell commands. As such, users' interaction with shell are also important common events.
- Graphic interface interactions. Uses often interacts with applications via graphical user interface. For instance, users can use a desk shortcut to launch an application. Therefore, users' interactions with graphic interfaces are also common events.

These events are high-level events. These provide a beginning point for us to explore more event types by breaking down these events to finer events. We explore event hierarchy in next section.

Prior works in Operating Systems logging and auditing provide a blueprint for common events that we should consider in our design, for instance, Trusted Solaris Operating System defines a list of events to support auditing queries [16, Appendix B Audit Record Descriptions].

2) EVENT TYPE HIERARCHY

In previous section we describe major events in Linux. These events are high level events, each consists of low level events, and we can continue to break down the low level events to even finer events. The high-level events and low-level events form a hierarchy. The hierarchy helps us determine event types we should consider in the design.

Figures 5 and 6 illustrate two event hierarchies, the Linux boot and user login events. For instance, Linux boot includes two lower level events, PC boot and Linux process initialization. PC boot contains three lower level BIOS events, starting from BIOS checking Master Boot Record (MBR) to boot loader loading operating system while Linux process

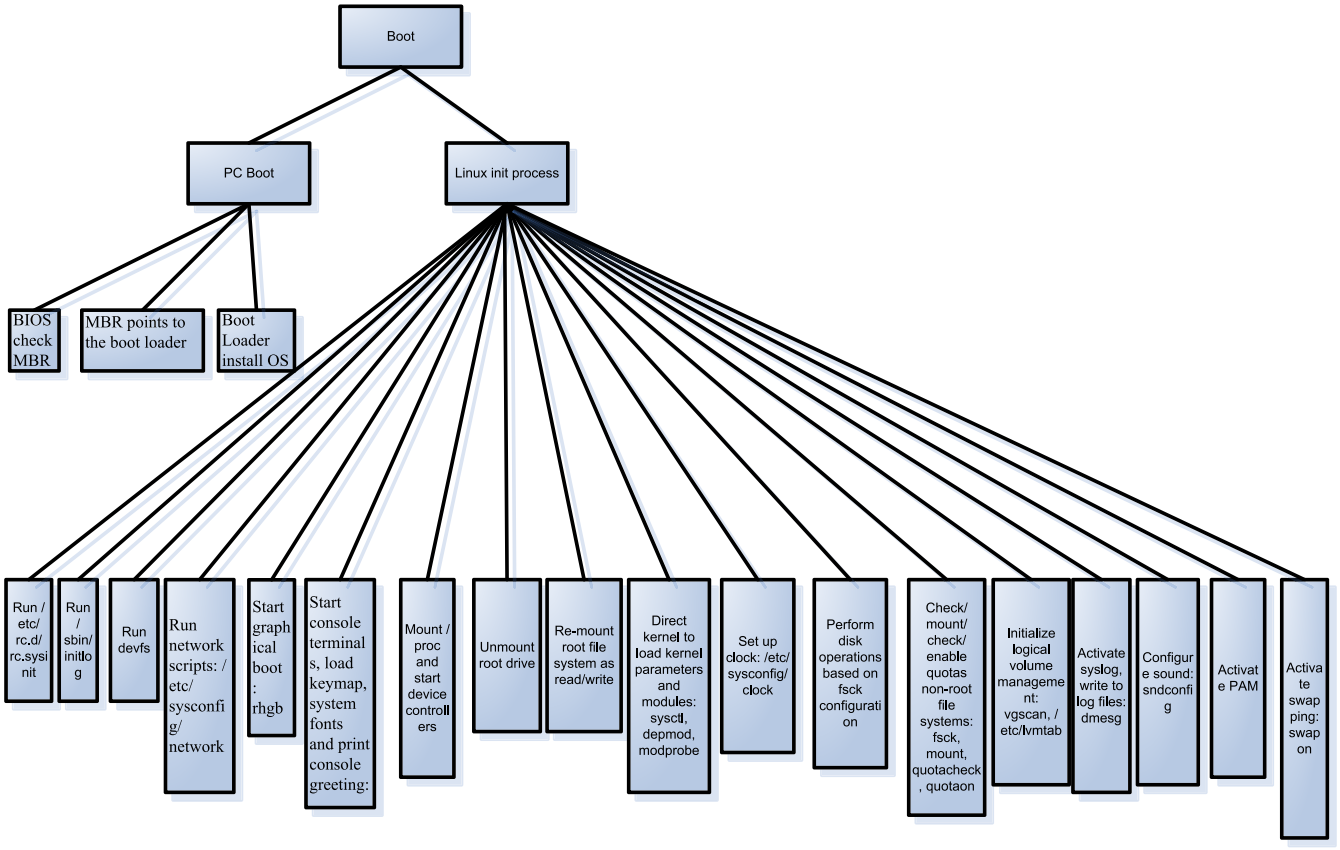


FIGURE 5. Boot structure tree.

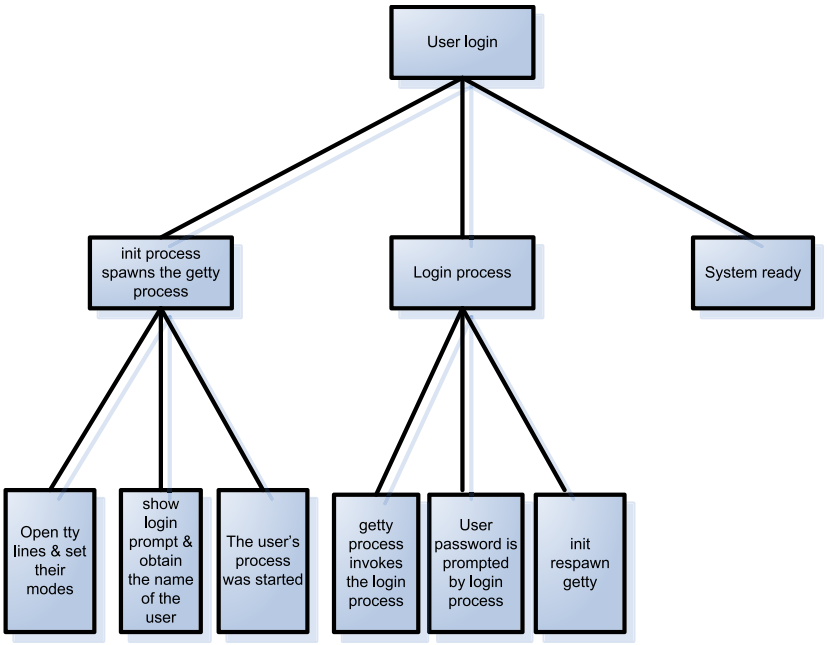


FIGURE 6. Login structure tree.

initialization includes all lower level process initializations necessary for booting Linux operating system. User login event has three lower level events, initialing process to spawn `getty`, login process and running users' sessions. In order to spawn `getty` process, system needs opening the `tty` lines, setting their modes, printing the login prompt, obtaining the user's ID, and starting the login process. Login process consists of authenticating user name and its password. At last, users are able to run their own sessions.

3) STRUCTURE OF LOGGING RECORD

A log consists of logging events, each corresponding to a logging record. We adopt the logging record format defined in Trusted Solaris [16], as such, a logging record consists of a sequence of logging tokens, typically, *header token*, *subject token*, *slable token*, and *return token* [16].

A Header token is the first token in a logging record. A subject token is a token that records information about a subject. A slable token holds sensitive label (i.e., "slabel") information that implies a confidentiality policy we must enforce for these tokens. At last, the return token stores the status of an involved operation, such as, a system call.

We adopt the set of token types defined in the Trusted Solaris Operating Systems [16], and Table 3 are examples of those token types. Each token type has its own format. In a logging record, a token may have data specific to the token. Since the type and the amount of data following a token vary, we adopt a specific format for each type of token. Table 4 lists the formats for selected token types, each is a list of attributes. For instance, "`acl, user_object, user id, r`" is an ACL token (an Access-Control-List token). It means a user with `user id` has read permission (i.e., permission `r`) on a specific object type, i.e., a `user_object`.

TABLE 3. Selected tokens implemented in accountable log system (Adopted from trusted solaris [16]).

Toke	Description
ACL	Access Control List information
arbitrary	Data including information of type and format
arg	The argument value of the system call
attr	Attributes of the file
Exec_args	Arguments for Exec system call
Exec_env	Environment variables for Exec system call
Exit	Information about program exit
File	information about audit file
Header	Beginning of record
Ip	Information about IP header
Iport	Internet port address
Process	Information about process token
return	System call status of
slabel	Sensitivity label information
Socket	Type and address of socket
Subject	Information about subject

C. FLOW-NET IMPLEMENTATION

1) LOGGING KERNEL EVENTS

LSM provides a number of function hooks (i.e., function pointers). These hooks are for loading security modules, such as, the SELinux module. For proof of concept, we simply adopt these hooks to capture the events with modification to SELinux, i.e., we modify SELinux in the event-capturing part of the program to build the flow-net structure where events are cross-referenced at flow intersections. We record events given in Section III-B. For instance, we ensure that the prototype records all the activity flows shown in Figure 4, i.e., we log events associated with User A, File B, and Use D.

TABLE 4. Format of selected tokens (adopted from trusted solaris [16]).

Token	Format
ACL	token id, object type, user/group id, permission
arbitrary	token id, print format, item size, number items
arg	token id, argument #, argument value, text length, text
attr	token id, file mode, owner uid, file system id, file inode id, device id

Additionally, we need to consider another problem: when do we create a new entry in the flow-net for an entity? If we maintain the cross-reference for all entities in the system at all times, it may dramatically slow the system and the flow-net may become too complex to be useful. To reduce flow-net size, we only record events associated with files or users. To further reduce flow-net size, we also consider, at what time do we need to and not need to insert a new event in the flow-net when an activity happens? For this, we determine whether the events along the same flow are equal, e.g., multiple modifications by User A to File B are equal events along Flow User A without interleaved by other events when file modification event token does not have the content of the file as one of its attribute. When an equal event occurs, we only record a reference to the preceding event with a new time stamp.

Listing 1 outlines the changes to Linux kernel to record the events when writing to a file. We captured the `write` event and logged it in `logEventBuf` in the kernel space. The next step is to forward the data in the kernel space to the user space.

In order to build a complex flow, we need to capture important events. For example, a `login` event is critical. To determine when the `login` event happens is also a concern. Because when the machine finishes booting, the system creates a `getty` process, the process will in turns create another process called `login`, both are via the `exec` system call. Therefore, a `login` event happens whenever a call to `exec` to create a `login` process. After the `init` process respawns the `getty` process, the `login` event has transpired. In addition, all file-related operations are essential and the prototype must log events for these operations, such as, `chmod`, `chown`, and so on.

```
1  #include <linux/cred.h>
2  #include <asm/current.h>
3  #include <linux/sched.h>
4
5  static ssize_t write (struct file *file, const char
       _user *userbuf, size_t bytes, loff_t *off) {
6    ...
7    static char logEventBuf[4096], prevLogEventBuf[4096];
8    int uid=current->real_cred->uid;
9    int euid=current->real_cred->euid;
10   struct dentry *dentry=file->f_path.dentry;
11   if (dentry->d_inode->i_iflog==1)
12     printk(KERN_INFO "%s want to write to log file %s\n
       ",current_euid(),dentry->d_name);
13   build_write_event(prevLogEventBuf, logEventBuf);
14   ...
15 }
```

List. 1. Revision made to the Linux kernel source code file `linux-2.6/fs/sysfs/bin.c`.

2) BUILDING FLOW-NET

In this prototype, we build flow-net in the user space. There are two user space components. One is to build the flow-net. Upon receiving an event in the user space transferred from the kernel space, we first check whether the involved entities are new. If an entity is new, we instantiate an entity object, and initialize a flow object for the entity. The entity object has a pointer referencing the flow object. If an entity is already in the flow-net, we append the event to the end of the entity's flow. The other like a syslog service program is to write the flow-net to a file on the disk.

We use an array list to represent a flow. Each flow has their own events. Listing 2 are example event data structures in the flow-net. Each event has a reference to next event in the flow. There are a list of flow intersection objects, and each contains pointers referencing intersecting events.

D. COMMUNICATION BETWEEN KERNEL SPACE AND USER SPACE

When we build a flow-net in the user space, we need to arrange the kernel to transfer the event to the user space. Linux kernel has methods to communicate with a user space program, and these methods include named pipes, the `copy_to_user` and `copy_from_user` kernel APIs, and the `netlink` socket [34], [35].

```
1  struct time_m {
2    int sec;
3    int min;
4    int hour;
5    int mday;
6    int mon;
7    long year;
8    int wday;
9    int tyday;
10 };
11
12 struct write {
13   char eventname[8]; // the name of the event
14   time_m time;       // the time event happens
15   dentry file;       // involved file's dentry
       structure
16   int effectiveid;   // involved user's effective id
17   int id;            // involved user's id
18   int next;          // pointer to the next event
19 };
20
21 struct read {
22   char eventname[8]; // the name of the event
23   time_m time;       // the time event happens
24   dentry file;       // involved file's dentry
       structure
25   int effectiveid;   // involved user's effective id
26   int id;            // involved user's id
27   int next;          // pointer to the next event
28 }
```

List. 2. Data stuctures for the `write` and the `read` events.

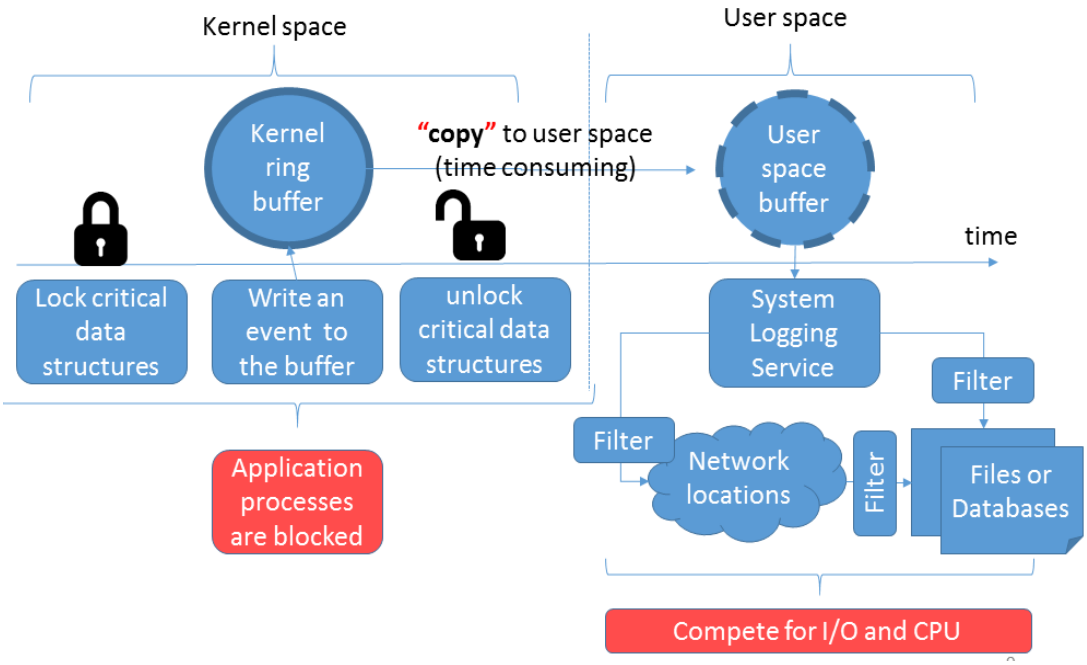


FIGURE. 7. Kernel and user space data exchange in flow-net logging.

Exchanging data between the user and the kernel spaces can be a significant overhead. Figure 7 provides an insight how we may reduce logging overhead. When the kernel writes a logging event to the kernel ring buffer, it must ensure mutual exclusive access to the ring buffer or some other critical data structures, for which, the kernel uses a lock. Since we access these data structures exclusively, additional requests to the data structures must wait, thereby, causing a delay. In addition, the system also runs a set of processes that moves log events from the kernel space to the user space, from the user space to a network location, or to a non-volatile storage. These processes compete CPU and I/O resources with other applications. Therefore, to reduce logging overhead, we may examine 1) the methods to reduce locking; 2) the methods to run logging services more holistically with other applications, for instance, via operating system scheduling, while user applications are busy at performing CPU intensive tasks, the logging services can make use of idle I/O resources; and 3) the methods to exploit the cost of store or dispatch logging events to different locations (typically, it is the fastest to write to memory, and the secondly fastest to write to a high speed network location, and the slowest to write to a local file.

In these three approaches, the named pipe follows the FIFO discipline. Because we desire to use a program in the user space to process events in an array, and the order of the processing isn't necessarily FIFO. The `copy_to_user` or `copy_from_user` kernel APIs copies data between two buffers, one in the kernel space, and the other the user space. The `netlink` socket is a two-way message passing interface between the kernel space and the user space [7], [34], [35]. In essence, we can use either `copy_to_user` or the `netlink` socket to transfer the event data in the kernel space to the user space. We tested both of these two communication methods in Section IV.

E. SOME ADVANTAGE OF FLOW-NET LOGGING

As we all know, the minimal time unit in common data exchange formats is 1/100 seconds, which is not sufficiently precise to establish the sequence of logging events. Precision of timing can be a source of challenge in keeping track of logging events, in particular when we transform one format to another for auditing or other purposes. For instance, it is common we choose the precision of 1/100 seconds when we write out logged events. This can confuse the order of events that happen closely in time. Traditional logging may easily loose the ordering of the these events seemingly happening at the same time; however, this won't be a problem for flow-net logging because it arranges events in a temporal order in a flow in the flow-net.

Second, the traditional logging mechanism merely shows discrete events with their time stamps. We infer the relationships among the events using the attributes of the events and the time stamps. Below is an example that a confusion may arise in Linux or other UNIX-like operating systems. Consider file access events. The recorded logging events often only use a file name to distinguish different files. For instance, we open a file "`file1`" and don't close it; we delete the "`file1`" and create another file with the same name "`file1`", which has totally different data. Then we close the original "`file1`". The logging record will be confusing because Linux does not delete an open file before

TABLE 5. Specification of testing computers.

Component	Description
Linux Distribution	Ubuntu 9.10
Processors	4 Intel Pentium CPUs at 3.00 GHz
Motherboard	Dell 0G8310
Primary Memory	512MiB DIMM SDRAM Synchronous 533*2
Secondary Storage	82801FB/FW (ICH6/ICH6W) SATA Cont, 40GB WDC WD400BD-75JM

it closes the open file. Therefore, we cannot figure out what happened exactly, i.e., whether we are operating a single file or two files with an identical name based on the file name using traditional logging records. If we use the flow-net model, the cross-references will show what was going on before, which offsets the problem.

IV. EVALUATION

A. PERFORMANCE OVERHEAD OF FLOW-NET LOGGING IMPLEMENTATION

The advantage of flow-net logging to aid auditing is evident as discussed above. The essential question we must answer is how much overhead the flow-net logging incurs on the system.

To answer this question, we design the following test scenario. We prepare two Linux systems, one with the original Linux 2.6 kernel (called the original system), and the other with the modified kernel with the flow-net logging (called the flow-net system). The specifications of the computers that run the two system are in Table 5.

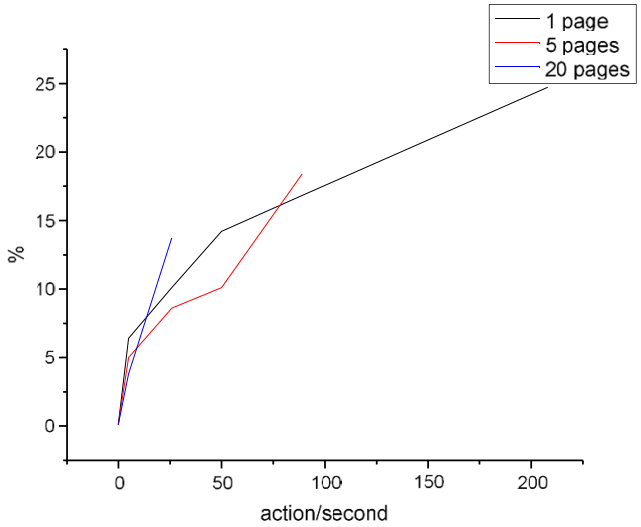
We then prepare a workload program. In the workload program, we open a new file, write a given volume of data to the file (e.g., a number of random integers), and subsequently, delete the file. We call these three steps a "time action". After each time action, we put the process in sleep for a period of time. By controlling the *sleeping period*, we can control the frequency (or rate) of the time actions. Using this, we can simulate either I/O-bound workloads or CPU-bound workloads. In the workload program, we repeat the time action-sleep cycles for a number of times, e.g., 2,000 times. We run the work load program in the two Linux systems and record the running times at different action frequencies. The difference of these running times of the workload program in the two systems tells us the performance overhead of the flow-net logging if any when compared to the original system.

TABLE 6. Overhead of implemented flow-net logging at an extreme case.

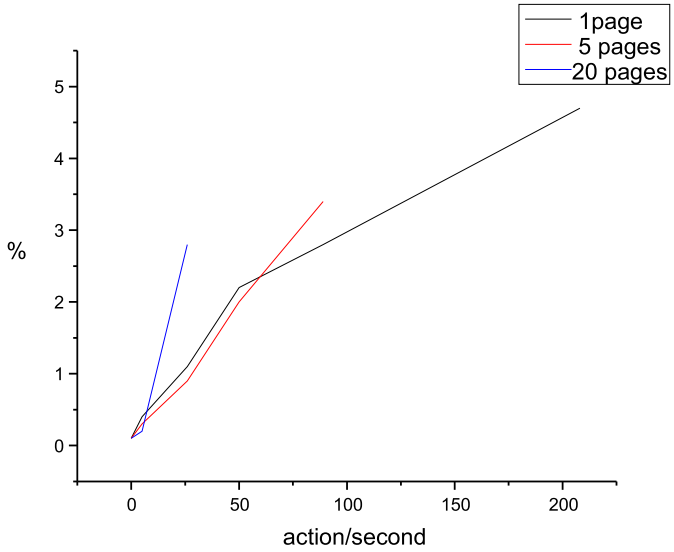
System	User time	System time	Total time
Original system	14,998	13,426	28,424
Flow-net system	23,695	17,254	40,949

We prepared two implementations, one uses `copy_to_user` kernel API to transfer log events in the kernel space to the user space (the `copy_to_user` implementation), and the other the `netlink` socket (the `netlink` implementation).

We first obtain the results using the `copy_to_user` implementation. We set the sleeping period as 0 (the smallest period) and write one random integer at a time (also the



(a) Implementation using `copy_to_user`



(b) Implementation using the `netlink` socket

FIGURE. 8. Performance overhead of flow-net logging.

smallest amount of data). This represents an extreme case as it is one of the most I/O intensive workloads. The results are in Table 6. The table shows that in the original system, the process took 14,998 time ticks of user time and 13,426 system time, and in the flow-net system, the process took 23,695 user time and 17,254 system time.

Because of the added logging logic in the kernel, such as, we modified the `read` and `write` functions in the kernel source code, we observe an increase in the system time. In the user space, we launched user space-logging programs to build flow-net and write the flow-net logs to files. The two programs that run simultaneously will also incur an overhead. This will increase the user time as well.

TABLE 7. Common accountability queries.

	Description
Query 1	List an entity's activities in a specific time range
Query 2	List all entities' activities in a specific time range
Query 3	List an entity's activities in a specific time
Query 4	List an entity's activity

Denote the total wall-clock running time of the workload program on the flow-net system as T_f and that on the original system as T_o. The performance overhead (or cost) of the flow-net logging system over the original system as $C = T_f/T_o$. Using this definition, we estimate the overhead at the extreme

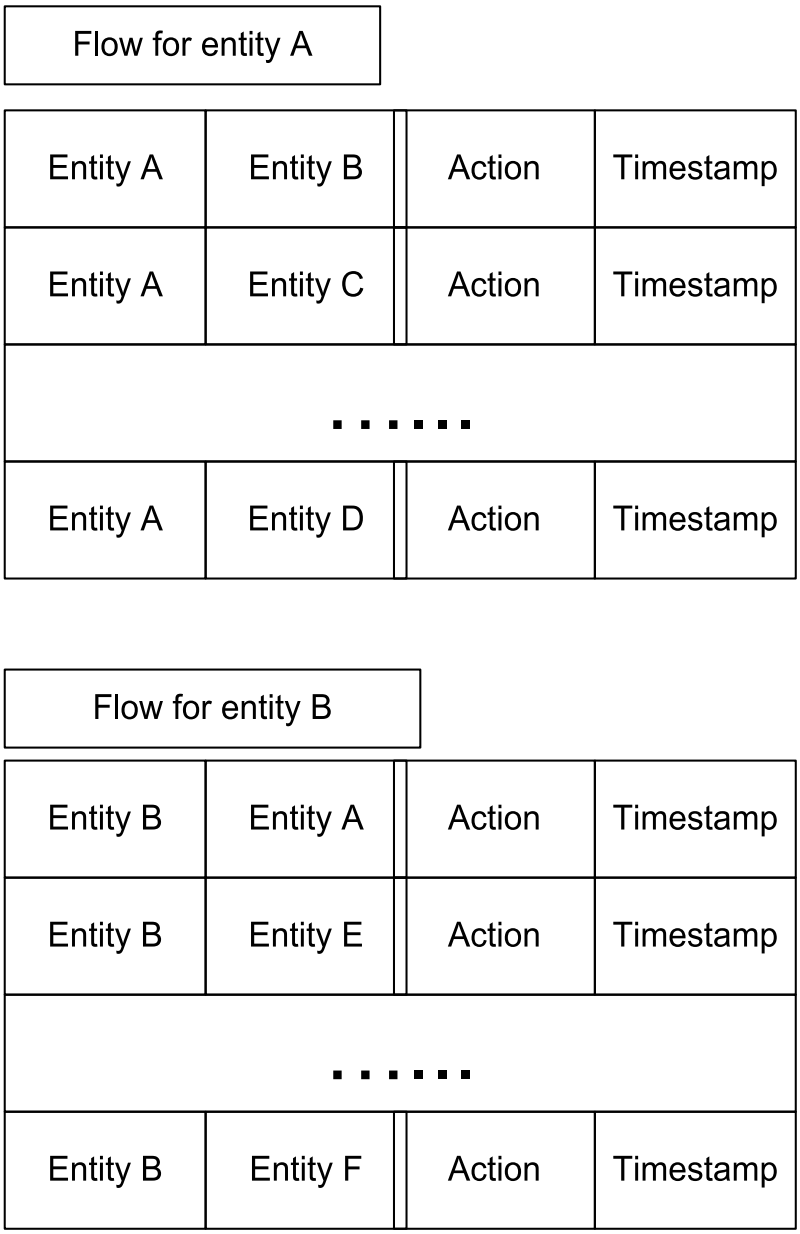


FIGURE. 9. Format of flow-net log.

case as $C = (40,949 - 28,424)/28,424) \approx 44\%$. This overhead is in effect the upper bound of the performance overhead.

Two factors impact flow-net logging. One is time action frequency. The other is the kernel buffer size for flow-net logging events. Figure 8 is the performance overhead versus action frequency (in actions/second) and kernel buffer size for flow-net logging (in pages where a page is 4 KB in the test systems). Logging events become more as action frequency increases, and the overhead becomes higher. Relationship between the buffer size and the overhead is more complex. In general, large buffer size leads to fewer transfers from the kernel space and the user space and to fewer disk I/O requests to write logs to files, which leads to less overhead; however, it isn't consistently observed in Figure 8.

B. QUERY PERFORMANCE OF LOGS

Accountability requires to trace back the events using logged data to answer some queries of interests. In this section, we will use common queries required by accountability and evaluate the query performance using the logged data. Table 7 lists four common accountability queries.

Throughout the remainder of the paper, we will refer to these queries as Queries 1, 2, 3, and 4. We evaluate the performance of these queries in traditional log and flow-net log in query time. For traditional log, we organize a log as a sequence of records of log events in a file ordered by the event time stamp, while for flow-net log, we do as flows, as illustrated in Figure 9.

We executed the 4 types of queries on traditional logs and flow-net logs and measured the query times. Figure 10 are the results when we vary log size, i.e., the number of log events.

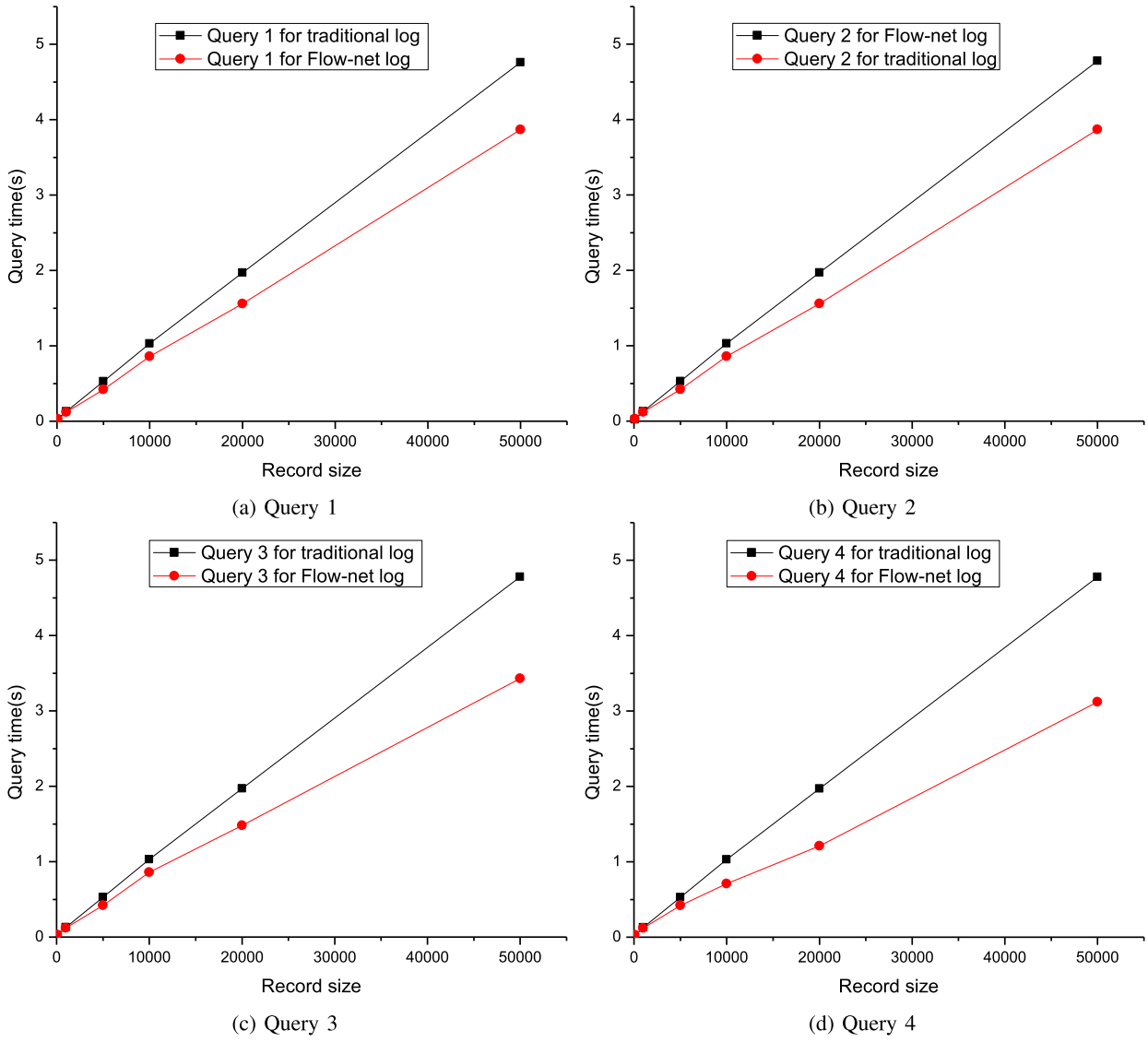


FIGURE. 10. Query times on traditional logs and flow-net logs.

The query times of Query 1 on flow-net logs are less than those on traditional logs. This is because we must conduct a sequential search over the entire log when querying a traditional log; however, we only need to follow a flow when querying a flow-net log.

The query times of Query 2 on flow-net logs are greater than those on traditional log. This is the result of the two following factors. In our implementation of flow-net logs, a log event may have multiple copies because multiple flows intersect at the event as the result that the event involves more than one entities, such as User A and File B (e.g., in the case that User A opened File B). Therefore, a flow-net logs tend to have more log records due to duplicative log events. Also, events in a traditional log is already in temporal order while in a flow-net log, events are not in temporal order when they are in different flows. As a result, to list *all* of the entities' activities in a specific time range, it requires checking all of the log records in flow-net logs while only partially in a traditional log, as such, the query times on flow-net log is greater than those on traditional logs.

The query times of Query 3 on flow-net log are less than those on traditional logs. This is the result of the following observation. To list an entity's activities at a specific time, we only need to search the entity's flow in a flow-net log while we must search the entire log on a traditional log.

The query times of Query 4 on flow-net log are also less than those on traditional logs. To answer this query on a flow-net log, we only need to locate the entity's flow while on traditional log, we ought to search the entire log. This explains the observation.

C. ENFORCING ACCOUNTABILITY POLICIES

We examine accountability policy enforcement in the flow-net logging framework. For this, we test these policies: 1) regular users cannot delete a log that records regular users' activities; 2) peer administrators can delete a log that logs regular users activities; 3) peer administrators cannot delete a log that records the activities of him/herself; and 4) peer administrators can delete a log that records the activities of other peer administrators.

We take advantage of SELinux's security policy language and express accountability policies as SELinux policies. Listing 3 are the policies written as a set of SELinux policies in SELinux policy language.

In the policies, `ourreglogtype_t` and `ouradmlogtype_t` are two new SELinux types with attribute `file_type` and `sysadmfile` to allow access by the administrator. `sysadm_t` has a full access (`read`, `write`, etc.) to `ourlogtype_t` type files. `relabelfrom` and `relabelto` indicate that `staff_t` can re-label `ourtype_t` type files from and to a different type.

Line 5 means that regular users, i.e., any user labeled as `staff_t` type can read log files, i.e., any files labeled as the `ourreglogtype_t` or the `ouradmlogtype_t` type, but cannot write to or delete these files. Line 6 indicates a system administrator, i.e., a `sysadm_t` user can read log files, and Line 7 tells that a system administrator can write or delete these log files.

```
1 type ourreglogtype_t,file_type,sysadmfile;
2 type ouradmlogtype_t,file_type,sysadmfile;
3 allow sysadm_t ourreglogtype_t:file { create_file_perms
      relabelfrom relabelto };
4 allow sysadm_t ouradmlogtype_t:file { create_file_perms
      relabelfrom relabelto };
5 r_dir_file( staff_t, { ourreglogtype_t ouradmlogtype_t } )
      ;
6 r_dir_file( sysadm_t,{ ourreglogtype_t ouradmlogtype_t } )
      ;
7 w_dir_file( sysadm_t,{ ourreglogtype_t ouradmlogtype_t } )
      ;
```

List. 3. Accountability policies written as SELinux policies.

Upon observing these policies are successfully enforced by running a set of testing applications, we also want to measure how much overhead there is in the flow-net logging implementation. The test scenario to measure flow-net logging overhead is identical to that in Section IV-A, albeit we use the improved solution, i.e., to use the `netlink` socket to transfer logging data from the kernel space to the user space. Figure 8(b) is the testing results.

V. CONCLUSION AND FUTURE WORK

In addition to preventative countermeasures, accountability has been increasingly viewed as a necessary means to address real-world security. A common approach to provide accountability is via logging and auditing. We posit that Flow-net is an effective technique to provide accountability by preserving event relationships during logging and by answering accountability queries efficiently during auditing.

Building upon prior works in flow-net, we designed and implemented a prototype of the flow-net logging framework to support accountability. The primary objective is to examine an important practical concern, i.e., to demonstrate the feasibility of the flow-net framework to support accountability. The experiments using the prototype indicate that the overhead of the flow-net logging is moderate, in particular, when we use the `netlink` socket as the means to transfer logging data from the kernel space to the user space. In SELinux policy language, we also wrote accountability polices. Tests indicate the flow-net prototype can enforce these policies also with moderate overhead.

Although the results are overall positive to meet the primary objective, the work has limitations. First, our prototype introduced modification to the Linux kernel source code, and it can introduce stability problems to the kernel and degrade its quality. Second, it remains uncertain whether race conditions exist in our implementation. The future work includes to structure the flow-net logging framework as a loadable kernel module, and to introduce formal methods to ensure the quality of the design and implementation.

REFERENCES

[1] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman, "Information accountability," *Commun. ACM*, vol. 51, no. 6, p. 82, 2008.
[2] H. Nissenbaum, "Computing and accountability," *Commun. ACM*, vol. 37, no. 1, pp. 72–81, 1994.
[3] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable Internet protocol (AIP)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 339–350, 2008.
[4] J. Mirkovic and P. Reiher, "Building accountability into the future Internet," in *Proc. 4th Workshop Secure Netw. Protocols*, Oct. 2008, pp. 45–51.
[5] Y. Xiao, "Accountability for wireless LANs, ad hoc networks, and wireless mesh networks," *IEEE Commun. Mag.*, vol. 46, no. 4, pp. 116–126, Apr. 2008.
[6] Z. Xiao, N. Kathiresshan, and Y. Xiao, "A survey of accountability in computer networks and distributed systems," *Secur. Commun. Netw.*, vol. 9, no. 4, pp. 290–315, 2016.
[7] L. Zeng, H. Chen, and Y. Xiao, "Accountable administration in operating systems," *Int. J. Inf. Comput. Secur.*, vol. 9, no. 3, pp. 157–179, 2017.
[8] Y. Lu, X. Wang, C. Hu, H. Li, and Y. Huo, "A traceable threshold attribute-based signcryption for mhealthcare social network," *Int. J. Sensor Netw.*, vol. 26, no. 1, pp. 43–53, 2018.
[9] R. Shirey, *Internet Security Glossary*, document RFC 2828, Internet Requests for Comments, May 2000.
[10] R. A. Nadi and M. G. H. A. Zamil, "A profile based data segmentation for in-home activity recognition," *Int. J. Sensor Netw.*, vol. 29, no. 1, pp. 28–37, 2019.
[11] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in DDoS attacks: Trends and challenges," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2242–2270, 4th Quart., 2015.
[12] A. Jose, R. Malekian, and B. B. Letswamotse, "Improving smart home security; Integrating behaviour prediction into smart home," *Int. J. Sensor Netw.*, vol. 28, no. 4, pp. 253–269, 2018.
[13] L. Bilge and T. Dumitraş, "Before we knew it: An empirical study of zero-day attacks in the real world," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 833–844.
[14] Q. Liu, F. Chen, F. Chen, Z. Wu, X. Liu, and N. Linge, "Home appliances classification based on multi-feature using ELM," *Int. J. Sensor Netw.*, vol. 28, no. 1, pp. 34–42, 2018.
[15] B. W. Lampson, "Computer security in the real world," *Computer*, vol. 37, no. 6, pp. 37–46, Jun. 2004.
[16] Sun Microsystems, Mountain View, CA, USA. (Nov. 1999). *Trusted Solaris Audit Administration*. [Online]. Available: https://docs.oracle.com/cd/E19109-01/tsolaris7/805-8057/805-8057.pdf
[17] W. Lee, A. Squicciarini, and E. Bertino, "The design and evaluation of accountable grid computing system," in *Proc. 29th IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 2009, pp. 145–154.
[18] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan, "The fuzzyLog: A partially ordered shared log," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 357–372.
[19] Y. Xiao, "Flow-net methodology for accountability in wireless networks," *IEEE Netw.*, vol. 23, no. 5, pp. 30–37, Sep. 2009.
[20] B. Fu and Y. Xiao, "A multi-resolution accountable logging and its applications," *Comput. Netw.*, vol. 89, pp. 44–58, Oct. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128615002042
[21] Y. Xiao, K. Meng, and D. Takahashi, "Accountability using flow-net: Design, implementation, and performance evaluation," *Secur. Commun. Netw.*, vol. 5, no. 1, pp. 29–49, 2012.
[22] B. Fu, Y. Xiao, and H. Chen, "FNF: Flow-net based fingerprinting and its applications," *Comput. Secur.*, vol. 75, pp. 167–181, Jun. 2018.
[23] Oracle and/or its Affiliates. (Sep. 2010). *Oracle Solaris Trusted Extensions Administrator's Procedures*. [Online]. Available: https://docs.oracle.com/cd/E18752_01/pdf/819-0872.pdf
[24] Z. Xiao, Y. Xiao, and J. Wu, "P-accountability: A quantitative study of accountability in networked systems," *Wireless Pers. Commun.*, vol. 95, no. 4, pp. 3785–3812, 2017.
[25] B. Fu and Y. Xiao, "Accountability and Q-accountable logging in wireless networks," *Wireless Pers. Commun.*, vol. 75, no. 3, pp. 1715–1746, 2014.
[26] H. Sun, S. McIntosh, and B. Li, "Detection of in-progress phone calls using smartphone proximity and orientation sensors," *Int. J. Sensor Netw.*, vol. 25, no. 2, pp. 104–114, 2017.
[27] R. Gerhards, *The Syslog Protocol*, document RFC 5424, Internet Requests for Comments, Mar. 2009. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5424.txt
[28] Red Hat. (2013). *Deployment, Configuration and Administration of Red Hat Enterprise Linux 5*. Accessed: May 10, 2019. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/index
[29] SELinux Contributors. (May 10, 2019). *Security-Enhanced Linux*. [Online]. Available: http://en.wikipedia.org/wiki/Security-Enhanced_Linux
[30] SELinux Contributors. (May 10, 2019). *SELinux Access Vector Rules*. [Online]. Available: https://selinuxproject.org/page/AVCRules
[31] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proc. USENIX Secur. Symp.*, Berkeley, CA, USA, 2002, pp. 17–31.
[32] D. P. Quigley, "PLEASE: Policy language for easy administration of SELinux," M.S. thesis, Stony Brook Univ., Stony Brook, NY, USA, 2007.
[33] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 241–254.
[34] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in Linux using netlink sockets," *Softw., Pract. Exper.*, vol. 40, no. 9, pp. 797–810, 2010.
[35] P. N. Ayuso. (May 10, 2019). *Communicating Between the Kernel and User-Space in Linux Using Netlink Sockets: Source Code Reference*. [Online]. Available: https://people.netfilter.org/pablo/netlink/netlink-libmnl-manual.pdf

YANG XIAO (M'98–SM'04) is currently a Professor with the Department of Computer Science, The University of Alabama, Tuscaloosa, AL, USA. His current research interests include cyber physical systems, the Internet of Things, security, wired/wireless networks, smart grids, and telemedicine. He has published over 200 journal and conference articles. He was a Voting Member of the IEEE 802.11 Working Group from 2001 to 2004, involving the IEEE 802.11 (WIFI) standardization work.

LEI ZENG graduated from the Department of Computer Science, The University of Alabama, Tuscaloosa, AL, USA, in December 2014.

HUI CHEN is with the faculty of the Department of Computer and Information Science, Brooklyn College, City University of New York and the Doctoral Faculty of the Ph.D. Program in Computer Science, the Graduate Center of the City University of New York. His current research interests include software engineering, wireless networks, wireless sensor networks, and system and network security. He has been teaching courses in software development, computer networks, computer security, and web programming among others. He is a member of the ACM and a Senior Member of the IEEE. He served and has been serving on technical program committees of conferences sponsored by the IEEE or ACM.

TIESHAN LI (M'09–SM'12) received the B.S. degree in ocean fisheries engineering from the Ocean University of China, Qingdao, China, in 1992, and the Ph.D. degree in vehicle operation engineering from Dalian Maritime University (DMU), China, in 2005, where he was a Lecturer, from September 2005 to June 2006, an Associate Professor, from 2006 to 2011, has been a Ph.D. Supervisor, since 2009, and a Full Professor, since July 2011. From March 2007 to April 2010, he was a Postdoctoral Scholar with the School of Naval Architecture, Ocean and Civil Engineering, Shanghai Jiao Tong University. From November 2008 to February 2009 and from December 2014 to March 2015, he visited the City University of Hong Kong, as a Senior Research Associate (SRA). Since September 2013, he visited the University of Macau as a Visiting Scholar for many times. His research interests include intelligent learning and control for nonlinear systems, multi-agent systems, and their applications to marine vehicle control.