Trustworthy Audit Logs: Deterring Lies about a File System's Past*

Da Xiao, Jiwu Shu, Rongrong Huang, Kang Chen
Department of Computer Science and Technology, Tsinghua University,
Beijing 100084, China
{xiaoda99, huangrr06, ck99}@mails.thu.edu.cn, shujw@tsinghua.edu.cn

Abstract

Audit logs that trace the changes of file system data to prevent fraudulent manipulation of data play an important role in regulatory compliance. However, most existing approaches fail to provide the trustworthiness of such logs in the presence of a powerful and determined insider adversary with incentives to alter data covertly. The adversary can modify file system data without being logged by bypassing the logging mechanism of the file system. She can also modify the audit log itself to erase evidence of conducted modifications. This paper proposes an approach to providing trustworthy audit logs for file systems under insider attacks by leveraging tamper-evident hardware devices such as secure coprocessors and HSMs. Covert modifications of data can be detected in the audit process by having each modify operation authenticated by the device, and the audit log is also protected from tampering by the device-generated MACs and counters. We have designed secure and efficient interaction protocols between the instrumented file system and the device for various file system modify operations and implemented our design with the ext3 file system. Experimental results show that adding the logging function to ext3 incurs a performance overhead of 5.7% under normal workloads.

1. Introduction

An audit log for a file system is a persistent record of how and when data in the file system has changed, providing an evolution history of the file system over time. Such audit logs have important applications in various areas. Traditionally, they have been a useful tool for intrusion detection and computer forensics. More recently, regulatory compliance has created new usage scenarios for audit logs. Due to the increasing incidences of fraudulent manipulation of electronic records, many laws and regulations have been established that require proper logging of access to electronic records and an audit process. These regulations can be found in financial, life sciences, healthcare and government sectors (e.g., HIPAA [9] and Sarbanes-Oxley Act [23]). Audit logs have thus been shifting from good practice to a mandate by these regulations. For example, HIPAA mandates proper logging of access and change histories for medical records [9]. In these cases, audit logs serve as evidence for organizations' compliance to, or violence of, regulations.

For an audit log of a file system to serve the aforementioned purposes effectively, it must be *trustworthy*, which means that the log should contain authenticate information about every modify operation on file system data. However, the trustworthiness of the audit log is hard to guarantee in the new usage scenario where the attacker is likely to be an insider or even the owner of the data with incentives to alter data covertly. For example, the CFO of a corporation may order the alteration or destruction of the corporate financial records after they come under suspicions of malfeasance. Similarly, a hospital might attempt to amend or delete a patient's medical records to hide evidence of improper treatments. The insider attacker is assumed to be very powerful and determined. She may have full control over the software (e.g., the file systems, the device drivers) and even the hardware (the disks) of the machine on which the file system resides. Specifically, she could perform the following two types of attacks:

(1) Modifying file system data covertly without the modify operation being logged. Most conventional approaches enforce logging at file system or OS level by instrumenting the file system or the OS with the logging function (e.g., Windows NTFS auditing and Syslog on Linux). Such a logging mechanism can be 'bypassed' by an insider attacker. She can modify on-disk data structures of the file system directly (e.g., using the dd command). The file system code can also be subverted to

*The work in this paper is supported by the National Natural Science Foundation of China under Grant No. 60473101; the National Grand Fundamental Research 973 Program of China under Grant No. 2004CB318205; the Program for New Century Excellent Talents in University under Grant No. NCET-05-0067.

disable the logging function. Thus the modify actions will not be captured and thus logged by the logging function embedded in the file system.

(2) Undetectably modifying the audit log (e.g., modifying or deleting a log entry) after it is generated. To protect the log contents from tampering, the most obvious solution is to store the log on non-rewritable media such as CD-ROM disks or WORM devices [3] [24]. However, the immutability provided by this method is not verifiable: an attacker could read the log from the original WORM device, modify it, and write it to another seemingly identical device. Another solution is to protect the integrity of the log contents using Message Authentication Codes (MACs) or digital signatures. Unfortunately, an insider adversary is likely to have access to the MAC or signing key, especially when she is the owner of the data. Therefore, she can change the contents of the log arbitrarily.

In this paper, we present an approach to providing trustworthy audit logs for file systems in the presence of insider attackers. The trust is rooted from a tamper-resistant hardware device attached to the machine called the *audit agent device*, which can be instantiated by a secure coprocessor [1] [2] or a HSM [18]. The basic idea of our approach is as follows: Each modify operation on a file system object needs to be authenticated by the device by generating a log entry for the operation which contains the content hash of the modified object and having the device authenticate the log entry. If the attacker tries to modify data covertly by bypassing the logging mechanism, the unauthenticated operation will be detected in the audit process by identifying the discrepancy between the file system data and the audit log. The audit log itself is also protected from tampering by counters and MACs which are generated by the device with a key held securely by it.

It is worth noting that with our approach, an attacker is still able to corrupt file system data or the log. However, she cannot do so covertly because the corruption is guaranteed to be detected in a later audit process. In this sense, our goal is not to *prevent* faulty behaviors, but rather to *deter* such behaviors by detecting and providing evidence for them.

To make our approach transparent to end users, an ordinary file system is instrumented to interact with the audit agent device to generate and authenticate log entries. The major challenges of a practical design include: (1) How to design the interaction protocols between the instrumented file system and the device to minimize the overhead of generating and authenticating log entries while ensuring security? (2) How to conduct the audit efficiently using the audit log? By tackling these challenges, this paper makes the following main contributions:

(1) We propose the idea of having each modify operation authenticated by the trusted device so that unauthenticated covert modify operations can be detected in the audit process by identifying the discrepancy between the file system data and the audit log.

(2) We present efficient and secure interaction protocols between the instrumented file system and the audit agent device for logging various types of file system modify operations.

(3) We explore the structure of the audit log to enable two audit methods: a relatively expensive method of auditing the whole log based on MACs and counters, and a more efficient method of auditing the operations on a particular object based on entry chains.

We have built a prototype implementation of our approach with the ext3 file system, which is the default file system for Linux and has been widely used. A kernel module-simulated audit agent device is also implemented according to parameters of typical commercial products. Experimental results show that audit logging incurs a performance overhead of 5.7% under normal workloads.

2. Audit and Threat Model

There are two players in our audit model: the owner of the file system data and the auditor. An audit cycle includes two stages: the normal usage stage and the audit stage. In the normal usage stage, the owner uses the file system in an ordinary manner. She may perform various operations on the file system (e.g., create/remove a file, read/write a file, create/remove/lookup a directory) and those operations that render data changes will be logged. In the audit stage, the owner is required to present both the file system data and the audit log to the auditor, who determines what changes have been made to file system data by examining the contents of the log and reports the facts to relevant parties. If the owner has behaved properly, the data and the log will pass the audit successfully. On the other hand, if the owner has tried to do something bad, the auditor should be able to detect the faulty behaviors by verifying the data and the log. If this is the case, the owner is hold responsible for the misconduct.

The distinguishable difference between our threat model and traditional file system security model is that in our model the attacker is not an external intruder but is the owner herself. The owner is semi-trusted. During most of the time, she is assumed to behave properly. However, occasionally she is motivated to modify file system objects covertly without leaving any trail for some reason. At this moment she becomes the attacker. She can

launch two types of attacks: (1) modifying file system objects without being logged; (2) undetectably modifying the log contents (e.g., modifying or deleting a log entry).

Our goal is to prevent these two types of attacks. Intuitively, with the assumption that the owner has complete control over the computer and the disk on which the file system data and the log are stored, it is relatively easy for her to manipulate the data at her own will without leaving any evidence. However, we will show that with the help of a tamper-resistant hardware device that acts as the trusted computing base, it is possible to achieve our goal in an efficient manner.

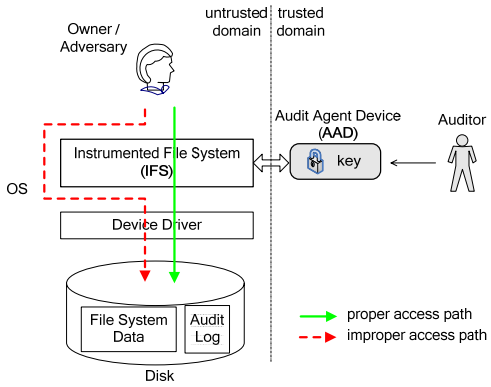
3. Design

3.1 Overview

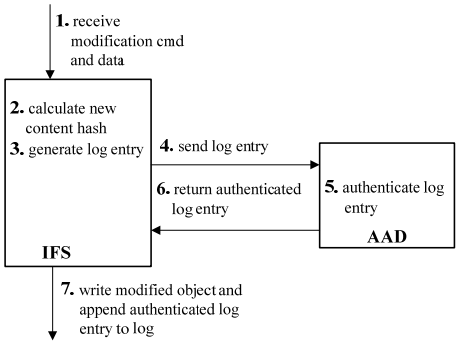
Figure 1 shows the overview of our approach. As shown in Figure 1(a), the design consists mainly of two parts: the instrumented file system (IFS) and a tamper-resistant hardware device called the audit agent device (AAD). An ordinary file system is instrumented by adding the logging function to it. The AAD, the only trusted component in the system, is a secure coprocessor [1] [2] or a HSM [18] that is attached to the machine. It is issued by the auditor to the owner of the file system. It holds some key materials securely, which are used to authenticate log entries. On each modify operation (e.g, creating / writing / deleting a file), the IFS interacts with the AAD to generate an authenticated log entry for the modify operation. The log entries can be stored by the IFS on the same disk as the file system data.

If the owner behaves properly, i.e., accesses data through the IFS, all her modify operations to file system objects will be correctly logged. On the other hand, if she modifies data by some other means in an effort to escape being logged, e.g., by manipulating on-disk data structure directly via device drivers, the improper modification will be detected in the audit process because the file system objects thus modified will not match with the log which contains the content hash of the object.

Figure 1(b) gives a high-level illustration of the interaction process between the IFS and the AAD. Upon receiving a modify command and the new data, the IFS first calculates a new content hash for the modified object. Then the IFS generates a log entry describing the operation and makes a request to the AAD to authenticate the log entry. On receiving the request, the AAD authenticates the log entry using the key held by it and returns it to the IFS, which writes the modified object and appends the log entry to the log.



(a) Components and deployment.



(b) Interaction process.

Figure 1: Overview of our approach.

Now let us have a closer look into the AAD. Typically, such a device contains the following main components interconnected by an internal bus: non-volatile secure memory for storing the key materials and the internal states (e.g. counters) as well as for holding the loaded programs, a crypto engine for performing cryptographic operations (e.g., SHA-1 hashes), a real-time clock for generating timestamps, and a microprocessor for executing the loaded program. It may also has a protective shield with sensors and a tamper detecting and responding circuitry to protect against physical attacks in an attempt to extract the key materials. The device is connected to the host machine via a PCI or PCI-X interface. During ordinary usage, the device responds to requests from the IFS through a simple interface. It also provides a special set of commands that can only be used by the auditor, e.g., by providing the correct PIN, to setup or retrieve the key material and the internal states.

Due to the special requirements on security, the processors on such devices are typically tens of times slower than the host CPU. Therefore, the interaction with the device must be designed to minimize the computation done by the device.

3.2 Data Structures

3.2.1 File System Objects and Content Hashes

To generate log entries, the IFS first needs to compute a cryptographic content hash for the modified file system object. To reduce the computational and I/O overhead of generating content hashes, we propose to calculate content hashes based on incremental cryptographic hash functions called AdHash [6]. We adapt the algorithm to calculate content hashes for file system objects.

In our design there are two types of file system objects: files and directories. The contents of a file are its data blocks. We define the contents of a directory D as the names of directory entries (including both files and subdirectories) of D. These directory entries are called the children of D. Therefore, when a child of D is created, renamed or removed, the contents of D will change, and a new log entry will be generated for the change.

The content hash of a file F comprising n blocks $b_1, \ldots, b_n$ is calculated as $H_f = \sum_{i=1}^{n} H(<i>\| b_i) \bmod M$, where $H()$ is a collision-resistant cryptographic hash function, M is a well-known large integer. In our current implementation the hash function is instantiated by SHA-1 and M is set to 2^{256}. If in a write to F, b_i is changed to b_i', the new content hash for F can be calculated as:

$$H_f' = H_f - H(<i>\| b_i) + H(<i>\| b_i') \bmod M \quad (3.1)$$

The above equation can be extended trivially to any number of changed blocks, and to situations where blocks are added to (e.g., append) or removed from (e.g., truncate) a file.

The content hash of a directory D containing n children with names $child\_name_1, \ldots, child\_name_n$ is calculated as $H_d = \sum_{i=1}^{n} H(object\_id_i \| child\_name_i) \bmod M$, where $object\_id_i$ is the object ID for the ith child. If a child changes its name from $child\_name_i$ to $child\_name_i'$, the new content hash for d can be calculated as:

$$H_d' = H_d - H(object\_id_i \| child\_name_i) + H(object\_id_i \| child\_name_i') \bmod M \quad (3.2)$$

Similar to files, the equation applies when an entry is created in d or removed from d.

With our method, a new content hash can be derived from the old content hash and the modified data, which is likely to be in the disk cache. Thus the I/O overhead is reduced. Computational overhead is also reduced because it is proportional to the amount of data being modified, rather than the size of the object.

3.2.2 Log Entry Structure

A log entry (LE) is defined as: $LE = (EB, EA)$. It consists of two parts: the entry body (EB), which contains information about the object and the modify operation, and the entry authenticator (EA), which is used to authenticate the log entry. EB and EA are generated by the IFS and the AAD respectively. The structure of the log entry is shown in Figure 2.

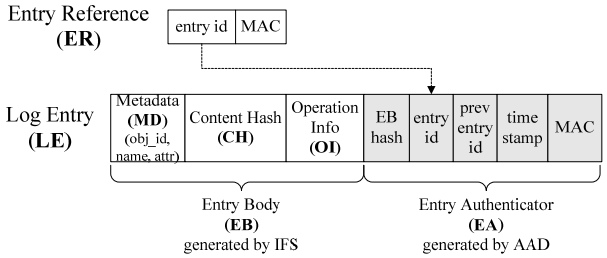


Figure 2: Log entry structure.

An EB is defined as: $EB = (MD, CH, OI)$. It is divided into three components: the object's metadata (MD), the object's content hash (CH), and the operation information (OI). The MD contains the ID, name and attributes of the object, i.e., $MD = (obj\_id, name, attributes)$. The *name* field is the file or directory name. As the name of an object may change during the lifetime of an object, we use a numeral object ID to uniquely identify an object. The *attributes* field contains attributes whose changes need to be audited, such as the ownership and permission of the object.

The OI component includes the type of the operation, e.g., CREATE, WRITE, RENAME, SETATTR, ADD_CHILD, and REMOVE_CHILD. The last two types of operations are for directory objects when a child is added to or removed from the directory. In these two cases, the name and ID of the relevant child object is also included in OI. The CH component is calculated with Equation 3.1 or 3.2, depending on whether the object is a file or directory.

The EA is defined as: $EA = (EB\_hash, entry\_id, prev\_entry\_id, timestamp, MAC)$. EB_hash is the cryptographic hash of EB, i.e., $EB\_hash = hash(EB)$. $entry\_id$ is the numeral ID of the log entry. $prev\_entry\_id$ is the ID of the log entry corresponding to the previous operation on the same object. The function of this field will be described in Section 3.4. $timestamp$ indicates the time of the operation. Finally, MAC is the MAC of the other fields of EA, i.e., $MAC = MAC_K(EB\_hash, entry\_id, prev\_entry\_id, timestamp)$, where K is the secret key held by the AAD. In our current implementation we use HMAC-SHA1 [5] to instantiate MAC_K. The authenticity

of the fields in EA is critical to the trustworthiness of the audit process so they are reliably generated by the AAD (except the *EB_hash* field, which is provided by the IFS).

Besides the log entry, there is another structure called entry reference (ER). It is used to relate a file system object to its most recent log entry. An ER contains the entry ID of the most recent LE and a MAC used to authenticate the ER structure, i.e., *ER* = (*entry_id*, MAC_K(*entry_id*)). The function of ER will be further explained in Section 3.4 and Section 4.

3.3 Interaction between the IFS and the AAD

3.3.1 The Protocol for Log Entry Authentication

When the IFS performs a modify operation and generates a log entry, it needs to send the entry to AAD to authenticate it. This is a basic protocol for the other more complex interaction protocols. The protocol for log entry authentication is shown in Figure 3.

IFS:
h ← *hash*(*EB*)
get *ER* from object metadata
if *ER* = NULL **then** /* creating an object*/
ER.entry_id ← –1
IFS → AAD:
send *h*, *ER*
AAD:
EA.EB_hash ← *h*
if *ER.entry_id* ≠ –1 **then**
check integrity of *ER*
EA.prev_entry_id ← *ER.entry_id*
EA.entry_id ← NEXT_ENTRY_ID_AAD
NEXT_ENTRY_ID_AAD ++
EA.timestamp ← *get_current_time*()
compute *EA.MAC*
ER´.entry_id ← *EA.entry_id*
compute *ER´.MAC*
IFS ← AAD:
send *EA*, *ER´*
IFS:
LE ← (*EB*, *EA*)
append *LE* to log
store *ER´* in object metadata
NEXT_ENTRY_ID_IFS ← *EA.entry_id* + 1

Figure 3: Log entry authentication protocol.

First, the IFS computes the hash of the entry body EB. To reduce the size of the data transferred to the AAD, the IFS sends only the hash of EB to the AAD instead of EB itself. The IFS gets the entry reference ER from the object metadata and sends it along with the EB hash. If the object is just being created and does not have an ER, the IFS creates one and sets its entry_id field to –1.

On receiving the EB hash and ER, the AAD first checks the integrity of ER by its MAC if it is not a newly created one. The ER is then used to set the *prev_entry_id* field of the entry authenticator EA. The AAD maintains an internal counter NEXT_ENTRY_ID_AAD which is used to set the entry ID. The counter increments by one on each transaction thus the entry IDs are continuously increasing. Then the AAD sets the timestamp according to its internal clock and computes the MAC of EA. A new entry reference ER´ is created and is sent back along with EA to the IFS.

On receiving EA and ER´, the IFS forms the complete log entry LE and appends it to the audit log. ER´ is stored in the object metadata for further use. The IFS also maintains a counter NEXT_ENTRY_ID_IFS, which keeps synchronized with the AAD counter NEXT_ENTRY_ID_AAD. The counter is used to assign object ID for a newly created object, as will be described in the next subsection.

3.3.2 Protocols for Various Operations

Various modify operation could be performed to file system objects via different system calls provided by the file system. All these operations require an interaction protocol between the IFS and the AAD to generate log entries describing the operation. We divide the modify operations into four different groups according to their actual effects on the relevant objects: an operation can affect either the targeted object or its parent directory or both. The operation groups and the operations (presented as system calls) included in each group is listed in Table 1. The last group contains only one special operation: the one of creating the root node of the file system.

Table 1: Four groups of modify operations.

group No.	affected object(s)	system calls
1	targeted object	write(), chmod(), chown()
2	targeted object's parent directory	link(), unlink(), symlink()
3	targeted object and its parent directory	creat(), mknod(), mkdir(), rename(), remove(), rmdir()
4	root object	mount() (the first one)

Now for each group we will describe an interaction protocol for a representative operation in the group. Protocols for the other operations can be derived from the

one for representative operation with minor changes. Note that in all protocols only the entry body authentication step involves interaction with the AAD; the other steps are executed solely by the IFS. A log entry is generated and authenticated for any affected object(s) of the operation. The protocols are shown in Figure 4.

write (Group 1)
$EB.MD \leftarrow (obj\_id, name, attributes)$
compute $EB.CH'$
$EB.OI \leftarrow$ (OP_WRITE)
authenticate EB

unlink (Group 2)
$EB_{parent}.MD \leftarrow (obj\_id_{parent}, name_{parent}, attributes_{parent})$
$EB_{parent}.CH' \leftarrow EB_{parent}.CH - hash(obj\_id \,||\, name)$
$EB_{parent}.OI \leftarrow$ (OP_REMOVE_CHILD, obj_id, $name$)
authenticate EB_{parent}

create (Group 3)
$obj\_id \leftarrow$ NEXT_ENTRY_ID_IFS
$EB.MD \leftarrow (obj\_id, name, attributes)$
$EB.CH \leftarrow 0$
$EB.OI \leftarrow$ (OP_CREATE)
authenticate EB
$EB_{parent}.MD \leftarrow (obj\_id_{parent}, name_{parent}, attributes_{parent})$
$EB_{parent}.CH' \leftarrow EB_{parent}.CH + hash(obj\_id \,||\, name)$
$EB_{parent}.OI \leftarrow$ (OP_ADD_CHILD, obj_id, $name$)
authenticate EB_{parent}

mount (Group 4)
$EB_{root}.MD \leftarrow$ (0, "/", $attributes$)
$EB_{root}\ CH \leftarrow 0$
$EB_{root}\ OI \leftarrow$ (OP_CREATE_ROOT)
authenticate EB_{root}

Figure 4. Interaction protocols for various operations.

In a write operation on a file, the IFS first constructs the MD component of EB according to the metadata of the file. Then it computes a new content hash of the file with Equation 3.1. The operation type of OI is set to OP_WRITE. Finally, the IFS initiates the EB authentication protocol as described in the previous subsection.

In a unlink operation of a file from a directory, the IFS first constructs the MD component of the parent directory's EB. Then it computes a new content hash of the file' parent directory with Equation 3.2, subtracting the hash of the unlinked object from the old content hash. The operation type is set to OP_REMOVE_CHILD. The object ID and name of the removed file are also included in the OI component. Finally, the IFS authenticate the EB.

In a create operation of a file in a directory, the protocol consists of two phases: constructing the log entry for the file to be created, and constructing that for the parent directory. In the first phase, the IFS first assigns an object ID for the file to be created. The object ID is assigned as the entry ID of this entry which will be given by the AAD. This ensures that any two entries never share the same ID, and that the correctness of the entry ID can be checked in the audit process. Then the MD component is constructed. The content hash is set to zero as no data has yet been written to the file. The operation type is set to OP_CREATE. In the second phase, the IFS first constructs the MD component of the parent directory's EB. Then a new content hash of the parent directory is computed with Equation 3.2, adding the hash of the created file to the old content hash. Finally, the OI component is constructed and the EB is authenticated.

In the first mount operation of the file system, a log entry for the creation of the root node of the file system is authenticated, which is the first entry in the audit log. The object ID and name of the root object are set to zero and "/" respectively. The content hash is also set to zero. The operation type is OP_CREATE_ROOT.

3.4 Auditing the File System Using the Audit Log

Once generated by the IFS and authenticated by the AAD, the integrity of the log entries and the whole audit log must be verifiable in the audit process. This is achieved by using MACs and counters generated by the AAD with the key K held securely by it. Whenever the AAD authenticates a log entry, it increases the entry ID counter (NEXT_ENTRY_ID_AAD) by one. Later the auditor can use these MACs and continuously increasing entry IDs to check if an attacker has tried to alter or remove log entries.

In a whole log audit process, the auditor first extracts the NEXT_ENTRY_ID_AAD counter and MAC key K from the AAD. She then traverses the audit log from the first entry with entry ID 0 to the latest entry with entry ID NEXT_ENTRY_ID_AAD – 1. For each entry, the auditor checks the integrity of EA using the MAC field and K, and checks the integrity of EB by recalculating the EB_hash. Then she increases the entry ID by one and checks the next entry.

Apparently an attacker without knowing K is not able to modify, insert or remove entries in the log without being detected in the whole log audit. Especially, she is also not able to remove entries at the end of the log (i.e. truncating the log) because the number of total entries in the log is maintained by the AAD as an internal counter (NEXT_ENTRY_ID_AAD), which can be inspected by the auditor.

The whole log audit requires checking the whole log entry by entry, which may be expensive. It is desirable to be able to audit the operations on a particular object without traversing the whole log. To enable this, the

entries pertaining to the same object are linked into an *entry chain* by the *prev_entry_id* field of each entry. The first entry of the chain is corresponding to the create operation of the object, whose entry ID equals to the object ID and whose *prev_entry_id* field equals to –1. The last entry of the chain is corresponding to the most recent operation on the object, which is pointed to by the ER of the object.

Here we give an example. Suppose at time t_1 a directory *xd* is created in */mnt/auditdir*, which is the mount point of the file system to be audited. At t_2 a file *file1.txt* is created in *xd*. This operation produces two log entries: one for the parent directory *xd* indicating that a child (*file1.txt*) has been added to it, and one for the created file which is the first log entry of object. These two entries have the same timestamp t_2. At t_3, some new data is written to *file1.txt*, producing the second entry for it, whose *prev_entry_id* field points to the first entry. The structures of the file system tree and the audit log at time t_3 are shown in Figure 5. Only *name*, *operation type* and *timestamp* fields are shown for each log entry.

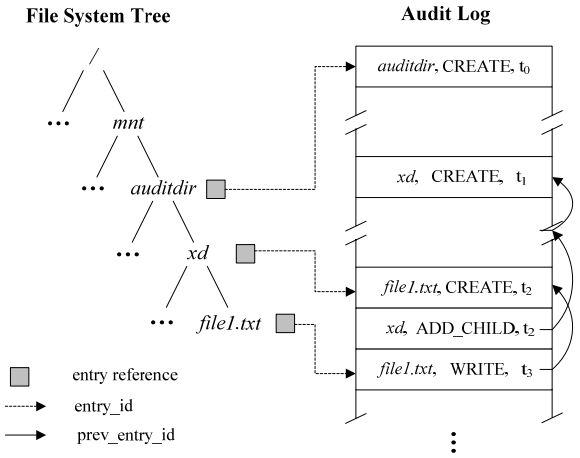


Figure 5: An example of audit log.

To audit the operations on a particular object, the auditor traverses the entry chain of that object, from the latest entry to the first one. For each entry in the chain, the auditor checks the integrity of EA and EB as is done in the whole log audit process. Then the auditor checks the integrity of the object by recalculating its content hash and comparing it with the one stored in the latest entry.

It is worth noting that our design does not make any assumption about the physical storage of the audit log. While the AAD-generated MACs provide a means for the auditor to detect tampering with the log, in our audit model it is the file system owner who is responsible to *prevent* the audit log from alteration. For this purpose, she can store the log together with the file system data and prevent write to it by policy. She can also store it on separate CD-ROM disks or WORM devices to achieve physical immutability.

4. Security Analysis

To state the security our approach provides, we first give the definition of a *consistent state*. The file system is consistent with the log if and only if every object in the file system has an ER pointing to a matching authenticated LE. It is clear that if the owner always behaves correctly, the file system will always be consistent with the log. The security of our approach can be stated as follows:

If before time t *the owner behaves correctly and at* t *the system is in a consistent state α, then the attacker can not change the system to another state α′ without the modify operation being correctly logged.*

Here we only give an intuitive explanation for the correctness of the statement instead of a formal proof: If any file system object is modified, the modified object will not match the original LE pointed by its ER because its content hash has changed. Therefore, the IFS must generate a new EB and have the AAD authenticate it and return a new ER. In addition, the IFS has to include authentic information about the change. Otherwise, the modified object will still not match the new LE.

In our approach, object IDs and Entry References play an important role in ensuring security. By including object IDs in the computation of content hashes for directory objects (Equation 3.2), an object is bound to its parent directory until deletion once it is created in that directory. The attacker can not covertly create a new object or use another object in a difference directory with the same name to replace the original object, because the new object will have a different ID from the old one and thus the content hash of its parent directory is changed. By using ERs, a newly generated LE for an object is always linked to the tail of the entry chain of that object; it can not be linked to a LE in the middle of the chain to skip some previously generated entries. Therefore, the completeness of the entry chain is guaranteed.

The above security statement embodies the notion of forward security, i.e., if the owner "turns bad" at time *t*, he can not fabricate log entries generated before *t*, but we can not make any secure guarantee about the log entries generated after *t*. The attacker can launch various attacks to compromise the secure of the audit log by subverting the IFS or directly operate the AAD. For example, the attacker can modify an object several times and generate only one log entry for the last modify. Such attacks,

which make some objects temporarily inconsistent with the log, are difficult to prevent with our basic approach. One possible countermeasure is to audit the file system periodically to see if it is still in a consistent state..

5. Implementation

We have built a prototype implementation of our approach with the ext3 file system by instrumenting it with logging functions. The instrumented file system is called ext3audit. In our prototype implementation, we use a kernel module called the audit agent module to simulate a real AAD. Both the function and the performance, e.g., bandwidth and processing power, of a real device are simulated by the audit agent module. Both ext3audit and the audit agent module call the Linux kernel crypto API [16] to perform cryptographic operations such as SHA1 hashing. The software architecture of the prototype is shown in Figure 6.

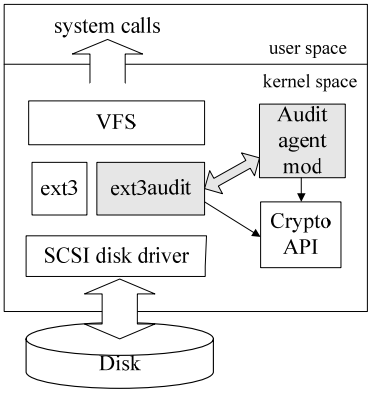


Figure 6: Prototype architecture. The shaded boxes are modules added by us.

5.1 Ext3audit

In our implementation, three types of audit-relevant metadata need to be stored with file system metadata: the content hash, the object ID, the ER and the NEXT_ENTRY_ID_AAD counter. The first three are for every file system object and are stored as ext3 extended attributes, while the last one are for the whole file system and are kept in the ext3 superblock. To minimize the degradation of performance when storing or retrieving extended attributes, we store a copy of the authenticator in the memory data structure of ext3 inode (*ext3_inode_info*). To reduce overhead, write operations are logged based on open/close sessions. If a file is written multiple times before it is closed, only one log entry is generated.

Currently, ext3audit stores the log entries in a log file. The log entries are appended to the end of the file. Alternatively, the log entries could be stored in an in-kernel Berkeley DB [14] to facilitate retrieval of entries with specific fields.

We added code to various ext3 file, inode and superblock operations to handle content hash calculation, interaction with the audit agent module and log writing. For example, in a file write operation, if it is a overwrite, the old blocks to be overwritten are first read into the page cache if they are not already there (in ext3_prepare_write()), and their SHA-1 hashes are calculated and subtracted from the content hash, then the new blocks' SHA-1 hashes are calculated and added to the content hash (in ext3_commit_write()). Before the file is closed (in ext3_release_file()), the EB structure is created and the ER structure is read from extended attributes, both of which are sent to the audit agent module, who returns the EA structure and a new ER. Ext3audit writes the complete LE to the log file and stores the new ER in extended attributes. Handling for other types of modification operations such as create and rename are similar. Nothing is done in operations that do not make changes, such as a file read.

5.2 Audit Agent Module

We implemented the AAD as a kernel module, which exposes a set of methods for ext3audit to invoke. To simulate the behavior of a real device, we constrain the "processing power" of the module. For example, assuming that the CPU on the device is 20 times slower than the host CPU, each method is called 20 times internally in the module upon an external call by ext3audit. We also insert a delay in each method to simulate the data transmission latency between the host and the device. The current system time is used for timestamping. The MAC algorithm was implemented using the HMAC-SHA1 keyed-hash functions provided by the Linux kernel cryptographic API [16].

6. Experimental Results

We measure the performance of ext3audit and compare it with native ext3 to evaluate the overhead introduced by audit logging. We evaluated the overall performance of ext3audit under various workloads using two macro-benchmarks: a CPU-intensive benchmark (am-utils build [26]) and an I/O-intensive benchmark (Postmark [15]). We also investigate the interaction overhead.

In each test, the overhead of ext3audit is measured under five different configurations:

HASH: Only the new content hashes of files are calculated by ext3audit using incremental calculation. The hashes are calculated on the fly and are not stored to or retrieved from the extended attributes, and ext3audit does not interact with the audit agent module. This configuration is used to quantify the overhead of the calculation of content hashes.

XATTR: The hashes are calculated and saved in the files' extended attributes. This configuration is used to find the overheads of setting and getting extended attributes.

AUTH: Ext3audit calculates content hashes and interacts with the audit agent module to generate and authenticate log entries. The entries are not written to the log file. This configuration is used to identify the overhead of interaction with the audit agent module.

ALL: Based on the AUTH configuration, the log entries are also written to the disk by ext3audit. This configuration gives the overall overhead of audit logging.

FULL-HASH: Same as the ALL configuration, except that the content hashes are calculated using full calculation. This configuration compares the efficiency of incremental calculation of content hashes with that of full calculation.

All measurements were performed on a machine with Xeon 2.40GHz CPU, 1GB RAM and a 34GB Seagate Cheetah ST336704LC disk, running Redhat Linux 9 with kernel version 2.6.10. All tests were run five times and the average results are reported.

We assume the AAD is connected to the host via a 32-bit, 33MHz PCI interface so the bandwidth of the device is 133MB/s. The transmission latency of each interaction can be derived from this bandwidth and the amount of data transferred. We constrained the processing power of the audit agent module according to typical parameters of commercial products. A representative secure coprocessor, the IBM 4764, computes SHA-1 hashes at a throughput of 1.42MB/s for 1KB blocks [2], which has a slowdown factor of 39.1 compared to the result measured on our test machine (55.5MB/s). We therefore assume that the processing power of the audit agent device is 40 times weaker than that of the host CPU. The audit agent module is configured to fulfill these parameters according to methods described in Section 5.2.

6.1 Am-build Build Results

The first macro-benchmark we used is a build of Am-Utils [26]. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, several binaries, scripts, and documentation. The Am-Utils build process is CPU intensive, but it also exercises the file system because it creates a large number of temporary files and object files. The elapsed, system, user and wait time are reported for each configuration. Wait time is the elapsed time less CPU time and user time and consists mostly of disk I/O. The results are shown in Figure 7.

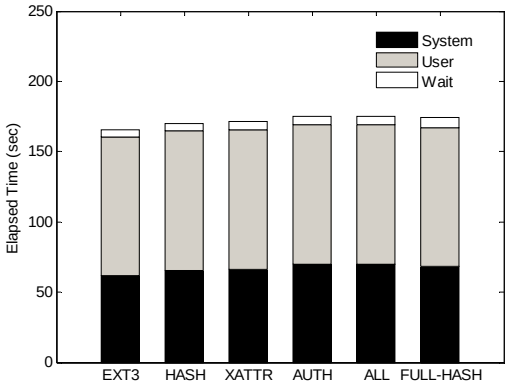


Figure 7: Am-utils build benchmark results.

For all configurations, the elapsed time overhead is within 5.7%. The system time overhead, which is mainly due to calculation of SHA-1 hashes and interaction with the audit agent module to revalidate authenticators, is within 13.6%. The interaction overhead constitutes most the system overhead. Note that in our current implementation, ext3audit communicates with the audit agent module synchronously, i.e., it sends a request to the audit agent module and waits until the module sends a response. If an asynchronously interaction model is used with a real device with its own CPU, the data processing on the audit agent device and that in the file system can be parallelized, and the interaction overhead will be reduced.

The wait time overhead, which is due to storing and retrieving extended attributes and writing log entries, is within 37.0%. We can also see that the FULL-HASH configuration outperforms the ALL configuration a little. It is because most of the file operations in Am-utils build are creates. In create operations, incremental calculation of content hashes has no advantage over full calculation because all the data must be processed for both methods. Since an Am-utils build represents a normal user workload, we conclude that ext3audit performs reasonably well under normal conditions.

6.2 Postmark Results

Postmark [15] is an I/O intensive file system benchmark designed to simulate heavy small-file workloads for applications such as electronic mail, netnews, and web-

based commerce. First it creates a large pool of files with the range of file sizes randomly distributed. It then runs transactions on these files, randomly creating and deleting them or reading and appending to them, and lastly it deletes all these files.

We configured Postmark to create 20,000 files and perform 100,000 transactions in 20 directories. Files range between 500 bytes and 100KB in size. Parameters not mentioned here are set as default. We measured the time taken by each of three stages of a run: creating the directories and files, performing transactions on these files, and deleting these directories and files.

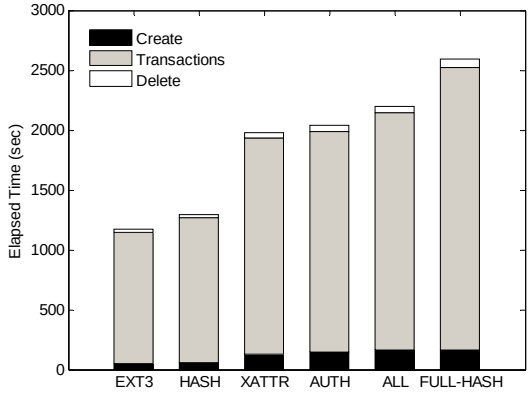


Figure 8: Postmark benchmark results.

Figure 8 shows the overhead of ext3audit for Postmark under different configurations. Unlike the Am-utils build, for Postmark we see a wide range of overheads for different configurations. The HASH configuration has an elapsed time overhead of 10.8%, which is mainly because of the calculation of SHA-1 hashes for blocks. The XATTR configuration has an elapsed time overhead of 68.7%. The increased overhead is due to setting and getting extended attributes. The AUTH configuration incurs another overhead of 67.4 seconds, which is due to interaction with audit agent module. Ext3audit with the ALL configuration is 1.9 times slower than ext3. The overheads for create and delete stages, which are 212% and 114% respectively, are higher than that of the transaction stage (81.3%). This is because parts of the transactions are read operations, which do not incur any overhead. The FULL-HASH configuration is 2.2 times slower than ext3, showing the advantage of incremental calculation of content hashes over full calculation in Postmark workload: when data is appended to a file, the old data need not be read from the disk first to calculation the hash.

Since Postmark creates 20,000 files and performs 100,000 transactions within a short period of just 20 minutes, it generates a rather intensive I/O workload. In normal multi-user systems, such workloads are unlikely. The above results show a worst case performance of ext3audit. Another thing to note that is the overhead related to the handling extended attributes contributes to the majority (about 60%) of the overall overhead. Although we cache authenticators in the memory inode, this performance penalty is still notable for a large pool of relatively small files in Postmark test. If we increase the size of the on-disk ext3 inode structure a bit so that the related structures can be stored in disk inodes instead of as extended attributes, the overall overhead will be further reduced.

6.3 Interaction Overhead

Now we have a detailed look into the overhead incurred by interacting with the AAD. For the four groups of modify operations listed in Table 1, the interaction overheads and their compositions are explored respectively. The results are shown in Table 2.

Table 2: Overhead of interacting with the AAD.

Op. group No.	**Interaction overhead**			**Max trans. rate (/s)**
	incurred by	time (ms)	total (ms)	
3	transmission ((|EB_hash| + |EA| + 2|ER|) x 2 = 240 Bytes)	0.0018	0.44	2273
	computation (5 HMACs, 3 for ERs and 2 for EAs)	0.44		
1, 2, 4	transmission ((|EB_hash| + |EA| + 2|ER| = 120 Bytes)	0.0009	0.26	3846
	computation (3 HMACs, 2 for ERs and 1 for EA)	0.26		

The total overhead is constituted of two parts: the time spent on transmitting data between the IFS and the AAD, and the time of computation on the AAD. The latter is comprised mainly of computing HMACs for ERs and EAs. For each group, we present the amount of data transferred and the number of HMAC operations the device performs (the results for group 1, 2 and 4 are the same). The time is measured based on our assumptions mentioned above, i.e., the bandwidth of the PCI interface is 133MB/s and the CPU on the AAD has a slowdown

factor of 40 compared to the host CPU. The maximum numbers of transactions the AAD can support are also shown in the table.

We can draw two conclusions from the table. First, the transmission overhead is negligible compared to the computational overhead. Second, because cryptographic operations done by the AAD are only lightweight HMACs operations, the computational overheads are small (within or a millisecond). For the operations in Group 3 and those in Group 1, 2 and 4, the AAD can support a transaction rate of 2273 and 3846 respectively. These figures indicates that log entry authentication by the AAD will not become the performance bottleneck for applications with a moderate I/O workload.

7. Related Work

A number of approaches have been proposed to instrument file system with the functioning of collecting useful information about the operations on the file system (e.g., Windows NTFS auditing, Provenance-aware storage (PASS) [17], the Lineage File System [21], journaling file systems, and file system trace collectors [4]). While these approaches collect various types of information for various purposes, none of them guarantees the trustworthiness of the information collected in a potentially untrusted environment. They rely fully on the OS or file system to collect the information to be logged, and are thus vulnerable to attacks that manipulate on-disk data structures directly bypassing the file system, that modify file system code to disable the log collecting function, or that tamper with the log data itself. Our approach focuses on the trustworthiness of the log by combining file system instrumentation with operation authentication by a trusted audit agent device.

By detecting improper modifications, our trustworthy audit log can be viewed as a technique to provide integrity for file system data. This problem has also been addressed in research on cryptographic file systems [8] [13] and file system integrity checkers [19]. A general approach used by these systems is to sign a cryptographic hash of the file with a private key held securely by the owner. The signature can later be verified by a reader. Our threat model differs from this model in that the attacker is the data owner herself who must provide evidence for fidelity to the auditor. In fact, our approach can be viewed as a variant of the signature approach with the signing key held securely by the AAD and thus protected from the trusted owner.

Recently, the research on content-immutable storage and WORM devices [3] [27] [10] has been motivated by the advent of regulatory compliance. While our approach prevents covert modifications to data objects by recording each modification, immutable storage prevents modifications to them at all. It achieves a higher level of security at the cost of changing the conventional semantics of file and storage systems. Immutable storage can also be used to store the audit log to protect it from tampering. Applied in this way, it is complementary to our work; while our approach deters such tamperings by detecting them after they occur, this technique prevents such tampers from occurring by mechanism.

Two recent works [20] [25] share a similar threat model with ours to prevent improper alteration of data by the file system owner. Both of them propose to preserve the history of file systems through continuous versioning; a new version of the object is created on each modification. The old versions themselves form in effect an audit log. They both advocate publishing content hashes of versions to a trusted publishing media periodically. Instead of relying on a trusted publishing media on the network, our approach rely on a tamper-resistant device that is attached to the local machine. We make this choice because we believe that the security and availability of such publishing media are hard to guarantee in the real world. In addition, our audit log based approach is more general than the versioning based approach because it can be applied to both versioning and non-versioning file systems.

Schneier and Kelsey propose a system for securing audit logs on untrusted machines using hash chains [22]. While their work mainly focuses on securing general audit logs, they do not address the problem of how to generate log information for operations on a file system, which is dealt with by our interaction protocol. In addition, they do not consider any log structure similar to our entry chain which can be used to audit the operations on individual object efficiently.

Secure hardware has been employed in a number of research works on system security to remedy the limitations of pure software based solutions. Itoi [12] and Hughes [11] propose to use smartcards for secure key management in cryptographic file systems. Microsoft BitLocker [7], which is part of Windows Vista Ultimate, combines TPM technology and disk encryption to protect sensitive data on stolen laptops. Our idea of using a tamper-resistant hardware device to aid in log authentication is inspired by these works. We contribute to this research area by making novel use of secure hardware to guarantee the trustworthiness of file system audit logs in an untrusted environment.

8. Conclusion and Future Work

We have proposed the design and implementation of an approach to log modify operations on a file system trustily in the presence of insider attacks. The audit log can be used in an audit process to evaluate the validity of data. The key idea is that a trusted tamper-resistant device is used to cooperates with an instrumented file system to authenticate each modify operation. We have implemented our design with the ext3 file system to evaluate its feasibility and efficiency. Experimental results show that the instrumented ext3 performs reasonably well compared to plain ext3 under normal workloads. The overhead under I/O intensive workloads is relatively high, but is mainly due to storage and retrieval of audit-relevant metadata with extend attributes and can be reduced with an improved implementation. In the future we plan to further evaluate the performance of our prototype system using file system traces. We also plan to evaluate the performance of auditing described in Section 3.4. Another possible direction of work is to integrate trustworthy audit logs into a continuous versioning file system.

References

[1] IBM 4758 PCI Cryptographic Coprocessor. http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml, 2006.

[2] IBM 4764 PCI-X Cryptographic Coprocessor. http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml, 2007.

[3] Axelle Apvrille and James Hughes. A Time Stamped Virtual WORM System. In *Proceedings of the SEcurite de la Communication sur Intenet workshop (SECI'02)*, September 2002.

[4] Aranya A, Wright C. P, Zadok E. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, 2004.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - Crypto'96*, pp. 1–19, 1996.

[6] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. *In Advances in Cryptology - EUROCRYPT' 97*, pp. 163-192, 1997.

[7] BitLocker. http://www.bitlocker.com

[8] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145. ISOC, 2003.

[9] United States Congress. The Health Insurance Portability and Accountability Act (HIPAA), 1996.

[10] Lan Huang, Windsor Hsu, Fengzhou Zheng. CIS: Content Immutable Storage for Trustworthy Electronic Record Keeping. In *Proceedings of the 23rd NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST'06)*, pages 101–112, 2006.

[11] James Hughes. Architecture of the secure file system. In Proceedings of the 8th IEEE Symposium on Mass Storage Systems, April 2001.

[12] Naomaru Itoi. SC-CFS: smartcard secured cryptographic file system. In *Proceedings of the 10th USENIX Security Symposium*, p.20-20, August 2001.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, 2003.

[14] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using Berkeley Database in the Linux kernel. http://www.fsl.cs.sunysb.edu/project-kbdb.html, 2004.

[15] J. Katcher. Postmark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[16] J. Morris. The Linux kernel cryptographic API. *Linux Journal*, 108, April 2003.

[17] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, Margo Seltzer. Provenance-aware storage. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[18] nCipher Hardware Security Module**.** http://www.ncipher.com/hardware-security-module.html

[19] Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[20] Zachary N. J. Peterson et al. Design and implementation of verifiable audit trails for a versioning file system. In *Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[21] Can Sar and Pei Cao. Lineage file system. Online at http://crypto.stanford.edu/ cao/lineage.html, January 2005.

[22] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and Systems Security*, 1999, 2(2): 159-176.

[23] United States Congress. The Sarbanes-Oxley Act (SOX). 17 C.F.R. Parts 228, 229 and 249, 2002.

[24] Yongge Wang and Yuliang Zheng. Fast and Secure Magnetic WORM Storage Systems. In *Proceedings of the 2nd IEEE International Security in Storage Workshop (SISW'03)*, San Francisco, CA, 2003.

[25] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[26] E. Zadok. Am-utils User Manual, version 6.2a3, November 2006, www.am-utils.org.

[27] Qingbo Zhu and Windsor Hsu. Fossilized Index: The Linchpin of Trustworthy Non-Alterable Electronic Records. In *Proceedings of the ACM SIGMOD Conference*, June 2005.