

國立交通大學

資訊工程學系

碩士論文

保護資料完整的硬碟系統

A Secure Disk System: Using File System Level

Knowledge for Disk Data Protection



研究生：蔡德聖

指導教授：張瑞川 教授

林正中 副教授

中華民國十三年八月

保護資料完整的硬碟系統

A Secure Disk System: Using File System Level Knowledge for Disk Data Protection

研究生：蔡德聖

Student : De-Sheng Tsai

指導教授：張瑞川教授

Advisor : Prof. Ruei-Chuan Chang

指導教授：林正中副教授

Advisor: Prof. Cheng-Chung Lin



A Thesis

Submitted to Institute of

Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science and Information Engineering

August 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年八月

保護資料完整的硬碟系統

研究生：蔡德聖

指導教授：張瑞川教授

指導教授：林正中副教授

國立交通大學資訊工程所

論 文 摘 要

系統遭到入侵之後，入侵者通常能得到系統管理者的權限，有了這個權限，系統上一些重要的資料可能會遭到破壞或竄改。然而這些資料是被儲存在硬碟上的，因此當系統遭到攻擊時，如果硬碟上有保護的機制來確保資料的完整安全性當系，系統就能從攻擊中存活過來。由於微處理器及記憶體晶片愈來愈快、小及便宜，將這些加入到硬碟系統上使得硬碟系統更為強大是可能的。有此能力，就可以在硬碟系統上加些功能來增加效能。因此，我們提出了一個安全硬碟系統的架構，這架構是一獨立的系統，當系統受到攻擊而意圖去竄改或破壞資料時，不受上層系統被入侵的影響，這硬碟系統可以確保資料的安全完整性。

在我們所設計的硬碟系統裡，為維持系統的一致性，我們不強制禁止這些不被允許的資料請求，而是採用備份的方法來達到保護的目的。一旦察覺不能被更改的資料遭到破壞，就將此資料做備份。而為了確保這份備份是安全不受入侵者更改，備份資料是存在硬碟系統中一個被保留不曝露給上層系統看到的地方。藉此，我們可以達到確實保護資料的目的。

因為目前硬碟無法讓我們加入自己的安全機制，為了評估此種機制的可行性，我們在一 Linux 的系統上實作此架構。我們在檔案系統和硬碟系統內插入一模組來擷取之間的資料請求。由我們的實驗顯示，此一機制並不會對系統效能有明顯的影響。其所需的記憶體的花費也是可以接受的。

A Secure Disk System: Using File System Level Knowledge for Disk Data Protection

Student: De-Sheng Tsai

Advisor: Ruei-Chuan Chang

Advisor: Cheng-Chung Lin

Computer System Lab
Department of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

When the system is attacked and the privilege of the administrator is gained by the intruder, some important system files may be tampered with or destroyed. We propose a secure disk system, called SecDisk that is local disk knowing the detailed knowledge of how the file system above is using the disk to protect on-disk data from intrusions. As chips of microprocessors and memory become smaller, faster and cheaper, it is feasible to place these into disks. Once a disk system has power computing capability, some functionality can be put into disks to improve system performance.

Disks have vantages to do data protection. When the system is attacked, we can safeguard data against intrusions because disks with computing power are compromise-independent of client systems. We protect data by backing up files in disk hidden space that is originally used for bad block remapping. We extend and exploit it for our backup storage. In our experimental results, the cost of memory space and time is reasonable and acceptable.

Acknowledgement

My advisor, Prof. Ruei-Chuan Chang, is the first one I would like to express my gratitude to. With the wonderful research conditions he provided and his attentive instructions, I came to discover the pleasure of research. I am also grateful to Dr. Da-Wei Chang for giving me a lot of advices on my research and thesis.

I would like to thank all the members of the Computer System Lab for their generous advices. Finally, I am grateful to my family for their encouragement and support.

De-Sheng Tsai

Department of Computer Science and Information Engineering

Nation Chiao-Tung University

2004/08



CONTENTS

論 文 摘 要	i
Abstract.....	ii
Acknowledgement	iii
CONTENTS.....	iv
List of Figures.....	vi
List of Tables.....	vii
1. Introduction.....	1
2. Related Work.....	5
2.1 Traditional Intrusion Detection	5
2.2 Disk Intrusion Detection.....	5
2.3 Extract FS-level Knowledge.....	6
3. Design.....	7
3.1 On-Disk Structure Extraction.....	11
3.2 Semantic Translation between Users and Disk System	12
3.3 Block Protection and Intrusion Detection	13
3.3.1 Block Protection	14
3.3.2 Intrusion Detection	14
3.4 Protection Policies	15
3.4.1 Warning	15
3.4.2 Backup	16
4. Implementation	19
4.1 Set Rule Process	21
4.2 Interceptor Wrapper	22
4.2.1 Rule Table	22
4.2.2 Interceptor and Extractor	23
4.2.3 Logger	24
4.2.4 Protection Policy and Recovery	24
5. Performance Evaluation.....	27

5.1 Installation Overhead	27
5.2 Runtime Overhead	28
5.2.1 Memory Cost	28
5.2.2 Legal Access Overhead	30
5.2.3 Backup Overhead	30
5.2.3.1 The Backup Time of Different File Size	30
5.2.3.2 Make Kernel	31
5.2.3.3 Delete Files	32
6. Conclusions and Future Work	34
Reference	35



List of Figures

Figure 1: Architecture of SecDisk	8
Figure 2: Processing Flow of Requests	10
Figure 3: On-Disk Layout of Ext2	12
Figure 4: A Backup Example	17
Figure 5: Infrastructure of SecDisk	20
Figure 6: Entries of Root Directory	22
Figure 7: Inode Hash Table	23
Figure 8: A Backup Example	26
Figure 9: Set Rule Time	28
Figure 10: Backup Time	31
Figure 11: Make Kernel	32



List of Tables

Table 1: Rule Example	9
Table 2: Model File Sets	29
Table 3: Memory Cost	29
Table 4: Time of PostMark	30
Table 5: Time of Deleting Files	32



1. Introduction

Nowadays, the attack technologies are numerous and diverse. It is difficult to prevent all of these attacks despite the best efforts of system managers. While new security technologies may make these attacks more difficult and less frequent, intrusions still exist as long as there are security holes in the system such as buggy software or fallible users. Once an intruder is successful to break into the system, he can obtain the privilege of the administrator. With that privilege, he can disable the intrusion detection mechanisms in the host operating system or disrupt the system by accessing, modifying, or destroying the important system data. He can also hide his activities through tampering with the audit log content so that the administrator cannot restore the system to the safe state according to the log.

Nearly all of the intrusion actions are visible in disk systems. For example, disk systems are involved when modifying important system data, tampering with audit log content (to scrub evidence), resetting files' attributes (to hide changes), or replacing system utilities (e.g. adding backdoors or Trojan horses) [16]. Moreover, Intrusion Detection Systems (IDS) that are placed in disk systems can not be disabled by intruders that successfully bypass the hosts' OS-level protection. Thus, a disk system is a good point for intrusion detection and data protection [16].

With the rapid development of the semiconductor technology, it becomes feasible to embed processors and memory in a disk system to enhance its power [1, 5, 13]. As a sequence, some works in the host system may be offloaded into the disk to improve the total system performance. For example, as computation occurs near data, we can improve performance by reducing traffic between the host processor and the disk system. Acarya [1] proposals an Active Disk architecture that integrates sufficient processing power and memory into a disk drive and allows the application-specific

code to be downloaded to process data that is being read from (or written to) disk. Sivathanu [15] proposes a new functionality, called File-Aware Caching SDS (FAC-SDS). FAC-SDS can smartly cache files with suitable sizes in order to improve the cache hit ratio.

It is difficult to accomplish these functions in a real disk system since it only uses a simple block interface for communicating with the host system. A request the disk receives through this interface means nothing but block transmission. Therefore, disks need to know more information for accomplishing these functions. In [15], it addresses this by extracting on-disk layout and understanding FS-level knowledge. FS-level knowledge means that the disk system understands knowledge of how the file system is using it, and exploits the knowledge to increase throughput or enhance functionality. For example, if the disk system knows which blocks constitute a file, it can perform smart pre-fetching on a per-file basis. Based on this concept, we want to add data protection and intrusion detection mechanisms into the disk system.

In this thesis, we describe the design of a secure disk system named SecDisk. It uses the knowledge of the file system running on it to protect important system data and detect intrusions. If the host system wants to modify a block that belongs to a need-to-be-protect file, SecDisk can notice this and back up the original data to a hidden area that is not exposed to the host. As a result, the intrusion can be detected and the original data can be recovered. Since existing disk drives do not allow us to implement our mechanisms on them, we use an in-kernel implementation as the proof of concept.

We briefly describe our architecture as follows. Currently, we focus our design on Linux Ext2 file system to develop these mechanisms. In SecDisk, we have a component, called extractor, to get the FS-level knowledge. Based on the disk layout structure defined by Ext2, we extract information from on-disk blocks. There is a rule

table in SecDisk that stores blocks' permissions (not allowed for read/write/delete) and rule policies (warning/backup). The administrator sets which files have which permissions and rules and SecDisk translates them into blocks through file-to-block semantic translation. With the FS-level knowledge and rule table, the component in SecDisk, called interceptor, can filter and inspect block requests issued from the file system. To protect data, SecDisk backs up the original data when the changes to data are issued from the file system and we have a recovery mechanism that helps the administrator to restore the data. Besides, SecDisk logs the data access traffic between the file system and the disk for intrusion detection and this log content is stored in the hidden space.

Our contribution is that we design a system that enables a block-interface based local disk system to provide data protection and intrusion detection. According to the experimental results, the cost of memory is acceptable and in normal cases, the overhead is nearly none. When violations occur and SecDisk does backup actions, although the time that backup actions take is long and degrades the system performance in our experiments, we think that it is acceptable and reasonable. The backup actions are taken when the protected data are modified and that means that someone is attacking the system. Degrading the system performance not only slows the intrusions down but also lets the administrator be aware of the intrusions right away. From the experimental results, it is feasible to add these mechanisms into real disk drives with computing power. Moreover, although we design the mechanisms for a local disk, we consider that the mechanisms can also be deployed in Storage Area Network (SAN) [6].

The remainder of this thesis is organized as follows. In Chapter 2, we discuss related work. In Chapter 3, we describe our design objectives and issues. Chapter 4 describes the implementation of our secure disk system. Chapter 5 evaluates the

performance. Finally, we conclude in Chapter 6.

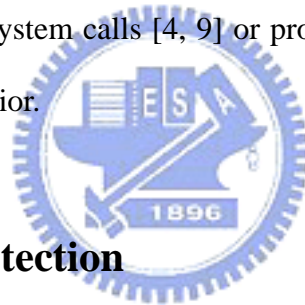


2. Related Work

The work related to our effort fall into three categories: 1. traditional intrusion detection, 2. disk intrusion detection, 3. extract FS-level knowledge.

2.1 Traditional Intrusion Detection

In order to prevent intrusions, many Intrusion Detection Systems (IDSs) [2, 12] have been developed, which fall into following the two categories: network-based and host-based IDSs. Network-based IDSs [3, 11] are embedded in sniffers or firewalls for examining suspicious traffic or traffic with attack signatures. Host-based IDSs are embedded within host operating systems for inspecting information, such as execution of an abnormal sequence of system calls [4, 9] or process profiling [10], for signs of intrusions or suspicious behavior.



2.2 Disk Intrusion Detection

In self-securing storage system (SSS) [16], it proposes a new place to detect intrusion, that is, disk systems. Some intruder actions (such as modifying the audit log content, adding backdoors, and etc) are visible in the storage system and it is very hard to disable the IDSs that deployed in the storage system. The difference between our system and SSS describes as follows. SSS's storage system is a NFS server (file server) with an object-based store. An object-based store [8] simplifies access control when compared to a standard block-based disk. In it, a file consists of a single object stored as a unit on the disk rather than blocks in a traditional block disk. That is, the basic unit between the file system and the disk is an object rather a block. Because of that, data protection with versioning is easier and more efficient. However, most modern computer systems still access disks via standard block interfaces (e.g., ATA,

SCSI). Therefore, their research results can not be applied directly to these systems. In our architecture, we design data protection and intrusion detection in a local disk with a standard block interface. This makes it possible for most computer systems to use our architecture to protect data. Moreover, data protection in a block-based local disk is more complicated than that in an object-based disk.

2.3 Extract FS-level Knowledge

Currently, it is feasible to embed processors and memory chips in a disk system to enhance its power. As a consequence, some researches [1, 5, 13, 15] put forward the concepts that some works in the host system may be offloaded into the disk to improve the total system performance. But it is hard to implement these functions in a block-based interface disk without enough FS-level information. To address this, semantically-smart disk system (SDS) [15] proposes that the disk system automatically gains the detailed knowledge of how the file system above is using the disk system. Based on this concept, we design data protection and intrusion detection mechanisms in block-based disk systems. Our goal is different from SDS's. SDS exploits the FS-level knowledge to improve performance, while we use the knowledge to protect data and detect intrusions.

3. Design

The design objectives of SecDisk are as follows. First, when our system encounters attacks, the integrity and safety of the protected data should always be guaranteed. Second, SecDisk can detect some intrusion actions that may be hidden by intruders at the file system level. These actions include adding backdoors or tampering with log content, and etc.

The first objective is our primary goal. When a malicious intruder breaks into the system, he usually can get the privilege of the system administrator. With that privilege, he can access and control all of the system resources without any limitation. For example, he can arbitrarily read, write, or modify system files (such as password, configuration, executable files, and etc.). These actions may make our system messy or even crashed. Thus, we want to ensure the integrity and safety of the important system files. Besides, for intrusion detection, SecDisk logs the traffic of the important system files between the file system and the disk. With that log content, the administrator can determine the damage caused by the intruders and fix the system.

Figure 1 shows the architecture of a system based on SecDisk. The SecDisk may be a local disk that attacked direct to the client system, or a network attacked disk that exports a block interface, which is used in a SAN [6] environment. The client system is the host system including operating system, file system, and etc. The SecDisk system has extra computing power and memory in which there are storage and an interceptor wrapper. Between the two systems there is a standard block interface (e.g. SCSI, IDE/ATA). The storage includes two parts: exposed storage (used by the client system to store data) and hidden storage (used by SecDisk to back up data and log information). There are several components in the interceptor wrapper. The rule table stores the access permissions (no-read/no-write/no-delete) and violation policies

(warning/backup) of the protected files in block formats.

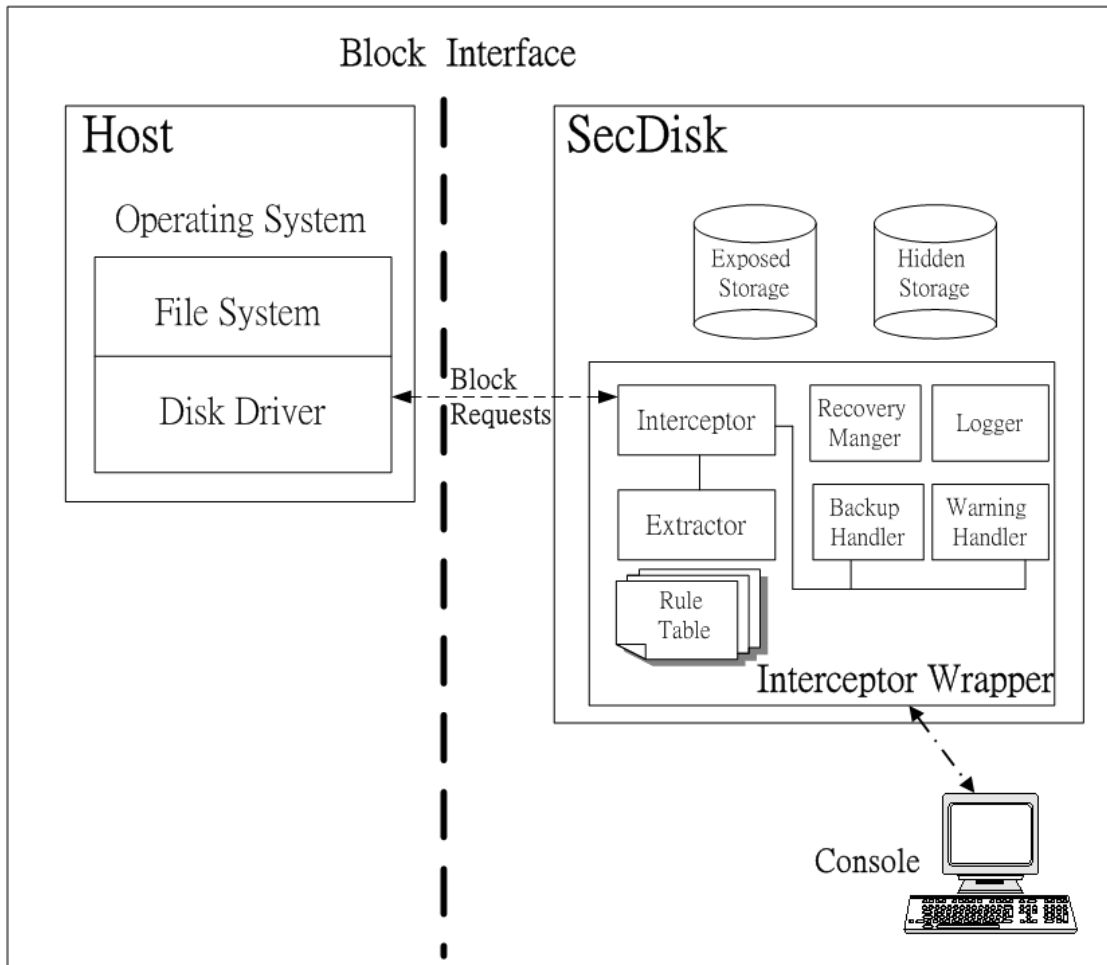


Figure 1: Architecture of SecDisk

We illustrate a rule example in Table 1. In our experiments, we use it as our file model to protect. All files in Table 1 are set with permissions of no-write and no-delete. And their policies are set either warning or backup as the administrator wants. When the block requests violate the access permissions, the set policy action of the protected data will be taken. The extractor extracts the on-disk layout structures and the interceptor uses the layout information to inspect block requests and judges them by consulting the rule table. The recovery component recovers the data from the

backup and the logger component logs the violation messages for the administrator to detect intrusions. The console is a communication channel for the administrator to use SecDisk system. Through this channel, the administrator can setup the files that needed to be protected and the corresponding rules.

File	Permission	Policy
/etc	no-write no-delete	backup
/boot	no-write no-delete	backup
/bin	no-write no-delete	warning
/sbin	no-write no-delete	warning
/usr/sbin	no-write no-delete	warning
/usr/bin	no-write no-delete	warning
/lib	no-write no-delete	warning

Table 1: **Rule Example**

Figure 2 shows the flow of how SecDisk handles the block requests. First, all block requests which are issued from the host system are intercepted and checked by the interceptor. Second, the checking is performed by examining the protection rule of the blocks, which are stored in the rule table. Third, if a block request is judged legal, the interceptor is allowed to get the data block from the exposed storage. Otherwise, the interceptor sends the request to the logger component. Fourth, the logger records the request in the hidden storage for future intrusion analysis and recovery. Fifth, according to the violation policy of the request stored in the rule table, the request is sent to either the warning or the backup handler. Under the warning policy, the warning handler just delivers a warning sign about protection violation. Under the

backup policy, the backup handler not only delivers a warning sign but also backs up the requested block in the hidden storage when it is changed. Note that these two policies can be used in different conditions, which will be discussed in Section 3.4. Finally, SecDisk allows the access request to the exposed storage.

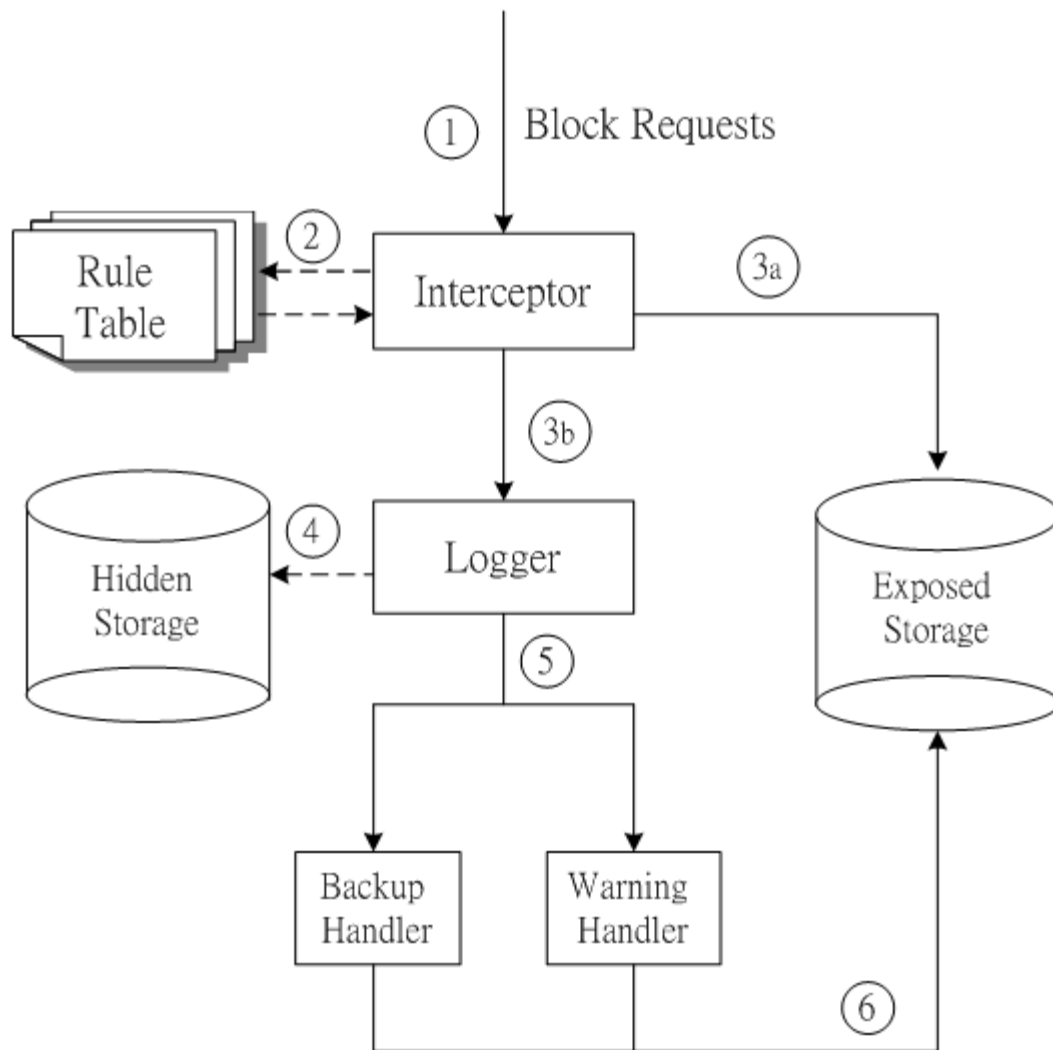


Figure2: **Processing Flow of Requests**

In the following, we discuss the design issues of the SecDisk system and the way we address these issues.

3.1 On-Disk Structure Extraction

Traditionally, a disk system receives a series of block requests, which are either read or write. For a read request, the disk can not tell if the block belongs to a file with secret data. For a write request, the disk does not know whether the write which corresponds to a file content update, a file deletion, or a metadata modification. Therefore, in order to provide useful data protection functionality in the disk system, the disk system must have some file system knowledge. That is, it must know the information of the file system running on top of it. The way to know that information is to extract the on-disk layout structure. In an Ext2 file system, for example, if an existing file is modified by an intruder, the updated inode and data blocks will be written back to the disk. The disk system must know the ranges of address of inode blocks and data blocks so that the disk system can efficiently and correctly exam these requests to find changes. Therefore, in order to add protection mechanisms into disks, our first step is to discover and acquire the detailed information of on-disk structures.

The best way to obtain the on-disk information is to automatically infer the file system type without the administrator involvement. SDS [15] has evaluated the costs and benefits of acquiring on-disk knowledge of Berkeley FFS-like file systems, finding that it is feasible and valuable. In SDS, it exploits the knowledge to improve system performance. Based on the concept of SDS, we use the knowledge to develop protection mechanisms. So far, inferring all kinds of file systems automatically is nearly impossible. Moreover, we put our focus on developing protection mechanisms. Therefore, we choose one of the most popular file system, Ext2, as the target file system when we design SecDisk.

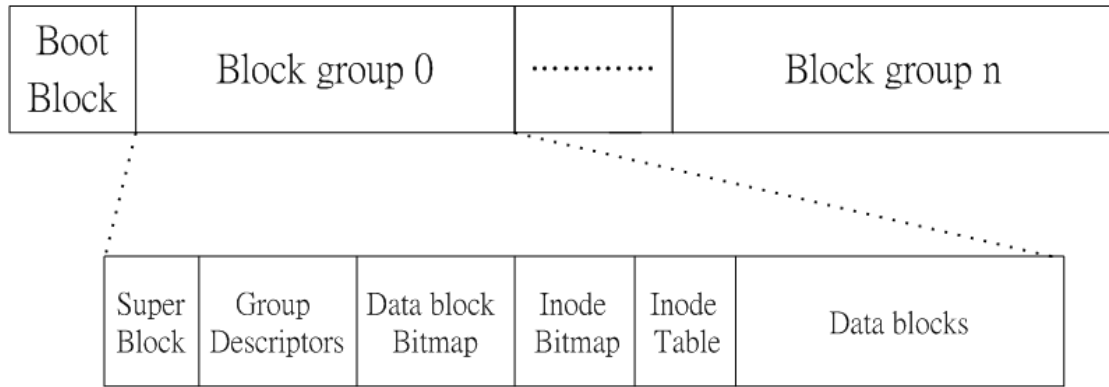


Figure 3: On-Disk Layout of Ext2

Figure 3 shows the on-disk layout of an Ext2 file system. In an Ext2 file system, a disk is partitioned into several block groups. In each block group, there are several types of blocks: superblock, group descriptors, data bitmap, inode bitmap, inode table blocks, and data blocks. Superblock is the most important block in an Ext2 file system because it includes all file system wide information. By extracting the information from a superblock, we can obtain the positions of the other types of blocks. For example, we may get a result that is similar to the following. Block 0 is superblock, block 1 is group descriptor, block 2 is data bitmap, block 3 is inode bitmap, block 4~49 are inode table blocks, and block 50~1000 are data blocks. With this information, we can just inspect the blocks that we concern about. For example, if we want to protect the system password file from being modified, we only have to inspect inode or data blocks to see if the inode or data of the password file is modified.

3.2 Semantic Translation between Users and Disk System

Knowing the type of a given block is not enough to achieve the goal of file protection. As we mentioned above, a system administrator has to specify the files to be protected and the protection policies via a private secure channel before using the SecDisk. This indicates that the administrator deals with files, instead of disk blocks.

Traditionally, the file-to-block translation is done by the file system. However, as a disk system, the help of the file system is not available. Moreover, the file system is not trusted. Once the system is invaded, an intruder can gain control of the file system and give incorrect information to cheat and mislead the administrator. Therefore, it is necessary for SecDisk to perform the file-to-block semantic translation. This gives the administrator a file-system environment that they are familiar with to set the protection rules of files. A protected file will be translated into the corresponding inode and data block numbers on the disk. With these block numbers, our disk system can then set protection rules on these blocks to protect them. In the following, we describe how the translation is done in the SecDisk system.

Basically, we use the same mechanism as the file system. In a file system, each file has an inode that stores the positions of all the data blocks for that file. Therefore we can know the data block numbers if we get the inode. The inode can be gotten by examining data blocks of the parent directory of that file. Thus, if we can get the inode position of the root directory, we can find out the inodes and the data blocks of any given files. In an Ext2, the inode number of the root directory is fixed as 2. Given a file path, SecDisk locates position of the root inode and extracts its data blocks to get the inode number of the next entry following the root directory in the file path. This step repeats until the inode of the target file is found.

3.3 Block Protection and Intrusion Detection

In this section, we describe our key issues about what block requests our disk system inspects for protection and intrusion detection. In addition, we also describe what intrusion detection we implement on SecDisk system.

3.3.1 Block Protection

In SecDisk, there is a rule table for storing rules of protected blocks. The rules specify that what permissions of blocks are allowed and what reactions are taken as block requests against rules occur.

Our primary goal is to protect files against attacks. Through file-to-block translation, we can know the inode and data blocks of a file. After the translation, we can ensure that a block request does not violate the rules for the file. Currently, the protection focuses on regular files and directories. The protection for other file types (e.g., symbolic link, FIFO, char devices, block devices, and etc.) is left as the future work. The rules of a file can be the combination of the following options: no-read, no-write, and no-delete. For the first two options, SecDisk inspects all block requests that corresponds to data blocks to look for the block requests that belong to that file. For the no-delete option, SecDisk inspects all block requests that corresponds to inode blocks to look for the block request that belongs to that file. As mentioned before, an inode is metadata of a file in Ext2 file system. A file's inode with a non-zero value of deletion time represents the deletion of the file. Thus, to protect a file from not being deleted, SecDisk inspects the deletion time value of the watched file's inode to see if a deletion occurs.

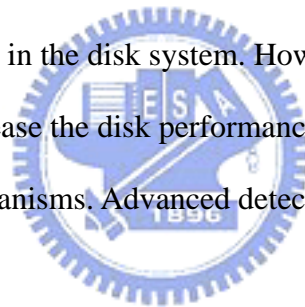
From above discussion, in order to realize these three options, SecDisk just needs to inspect two types of blocks: inode and data blocks. Examining these block types, we can efficiently ensure the protected files to be safe.

3.3.2 Intrusion Detection

System log content is an important source for the administrator to detect and analyze attacks. For example, we may not allow some important files or directories to be changed, such as adding files to the directories or modifying the files' content. But

some crafty intruders may tamper with or erase the log content. Thus, the administrator has no idea what intruders have done to the system. In most attacks, intruders may access and modify disk data to damage the system and these actions are all seen in the disk system. Therefore, SecDisk keeps the disk access log in the hidden storage to allow the administrator to analyze it. Because the log content is stored in the hidden storage that can avoid attacks, the administrator can use it to find out the attacks, determine the damage caused by intruders and restore the changed files to original status. In order to reduce the size of the log content and make the analysis more efficient, SecDisk just logs the accessing actions that are related to the watched files (set by the administrator for protection).

Besides, there are other complicated intrusion detection methods, such as virus scanning, can be implemented in the disk system. However these methods require more resources and may decrease the disk performance. Thus, we currently focus our efforts on the protection mechanisms. Advanced detection methods are left as future work.



3.4 Protection Policies

When a request against the rule occurs, its corresponding protection policy is taken. In SecDisk, we design two policies: warning and backup. The administrator can set different policies for different files according to his consideration. In this section, we elaborate on these two policies.

3.4.1 Warning

Warning policy does a simple action. When a violation occurs, SecDisk just delivers the warning sign of the violation to the console. These actions help the administrator detect the attacks right away when the intruders modify the protected

system data, and the administrator can take some actions to fix the system. Delivering the warning signs causes less overhead to our system. Therefore, in our design, this policy is suitable for protecting less important files, or in the condition that the administrator does not want to sacrifice the system throughput.

3.4.2 Backup

In our system, backup is the main policy for protecting data. When a protected file is modified by an intruder, SecDisk backs it up in the hidden storage for future recovery. A file consists of one or more data blocks. Only the modified blocks are backed up. We illustrate our backup mechanism with an example in Figure 4. Given that a file “foo” is composed of data blocks on the disk. If one of the protected blocks is modified by the high-level system, SecDisk backs up the original data of the modified block to the hidden space. After the backup action, SecDisk should record the relation between the protected files and backup blocks. That includes which file the backup block belongs to, the offset of the block in the file, and etc. With that information, we can recover the original file from these backup blocks through a recovery mechanism. Through the console, the administrator can instruct SecDisk to recover backed up files from the blocks in the hidden storage to their original status.

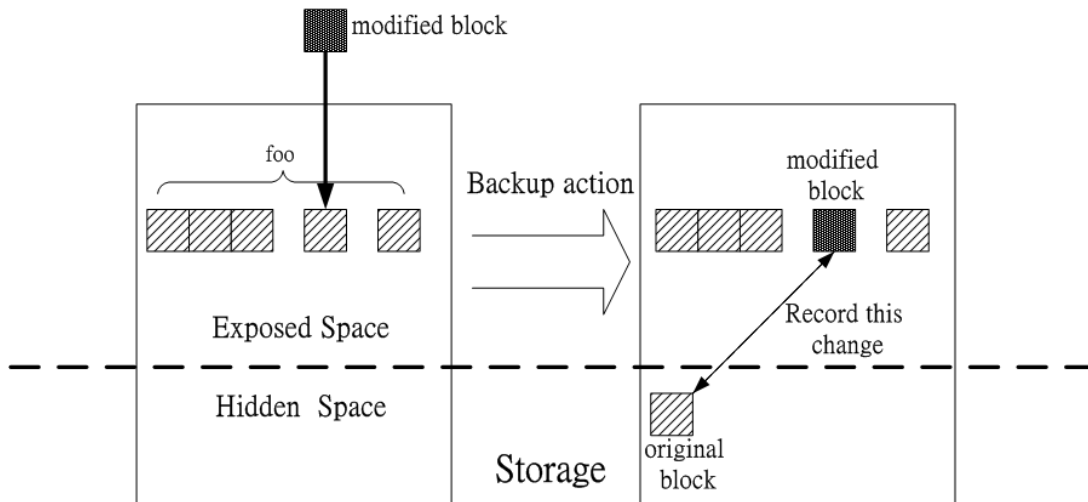


Figure 4: **A Backup Example**

Note that the backup is performed only for the first write to the block. That is, when the block of a file is first modified since the file is set protected, SecDisk backs it up and removes the block from the protected block list. This is because the goal of our protection mechanism is to guarantee that the original data is kept safe and integrity when the system is attacked. Therefore, after first modified, SecDisk does not back it up again.

If the administrator wants to ensure file safety, he can use the backup policy. But it results in more overheads than the warning policy. This will decrease the total disk performance. We will discuss this in detail in the performance evaluation section.

Besides warning and backup policies, there is one more policy we can use in SecDisk, named rejection policy. As its name indicates, SecDisk will reject block requests if they are against protection rules. In the no-read permission, its corresponding violation policy is rejection policy. This is strongest and safest protection policy. But this policy results in the stat inconsistency between the file system and the disk. Generally, a file system keeps on-disk data structures (such as superblocks and bitmaps) in memory for performance purpose. Thus, in the disk

system, if we refuse a block request that has been allowed by the file system, the state inconsistency problem occurs. That inconsistency makes our system messy. For example, if the file system deletes a file is not permitted by our disk system rules, it frees the corresponding inode and data blocks of the file and resets the inode bitmap and data bitmap. Later it may use these blocks for next creation of a new file. But in our disk system, we refuse that block request. So there is a collision between the new file and the original file. The new file does not exist on the disk and the original file is not seen by file system. This policy may be used when the administrator wants to strongly protect the files in any situation. Because of the inconsistency problem, we do not apply the policy to our system.



4. Implementation

We need a disk system with computing power to implement our design but now, it is not easy to obtain this type of disks. Thus, in order to prove our concept, we implement our mechanisms as a kernel module (named Interceptor Wrapper) and a user process (named Setup Rule Process). Our implementation is in a host PC running Linux with kernel 2.4.18 and Ext2 file system. Figure 5 depicts our infrastructure and a flow of how our system works.

First, the extractor extracts the on-disk Ext2 layout structures to gain information (such as block type position range) for protection. Second, the administrator sets file-based protection rules through the setup rule process. Then, these rules are translated into block format. Third, the block-format rules are stored into the rule table via our implemented system calls. Fourth, afterwards all block requests issued from the file system are interposed by the interceptor. Fifth, the interceptor compares the requested block number with the block numbers stored in the rule table. This comparison is to find out if this block number is set watched by the administrator. If it is not found in the rule table, the interceptor lets the request access the physical disk without any intervention. Sixth, if the requested block number is in the rule table, the logger logs the violation of the requested block number for intrusion detection. Seventh, the corresponding protection action is taken depending on what policy (Backup or Warning) is set by the administrator. Finally, SecDisk lets the request to access the physical disk.

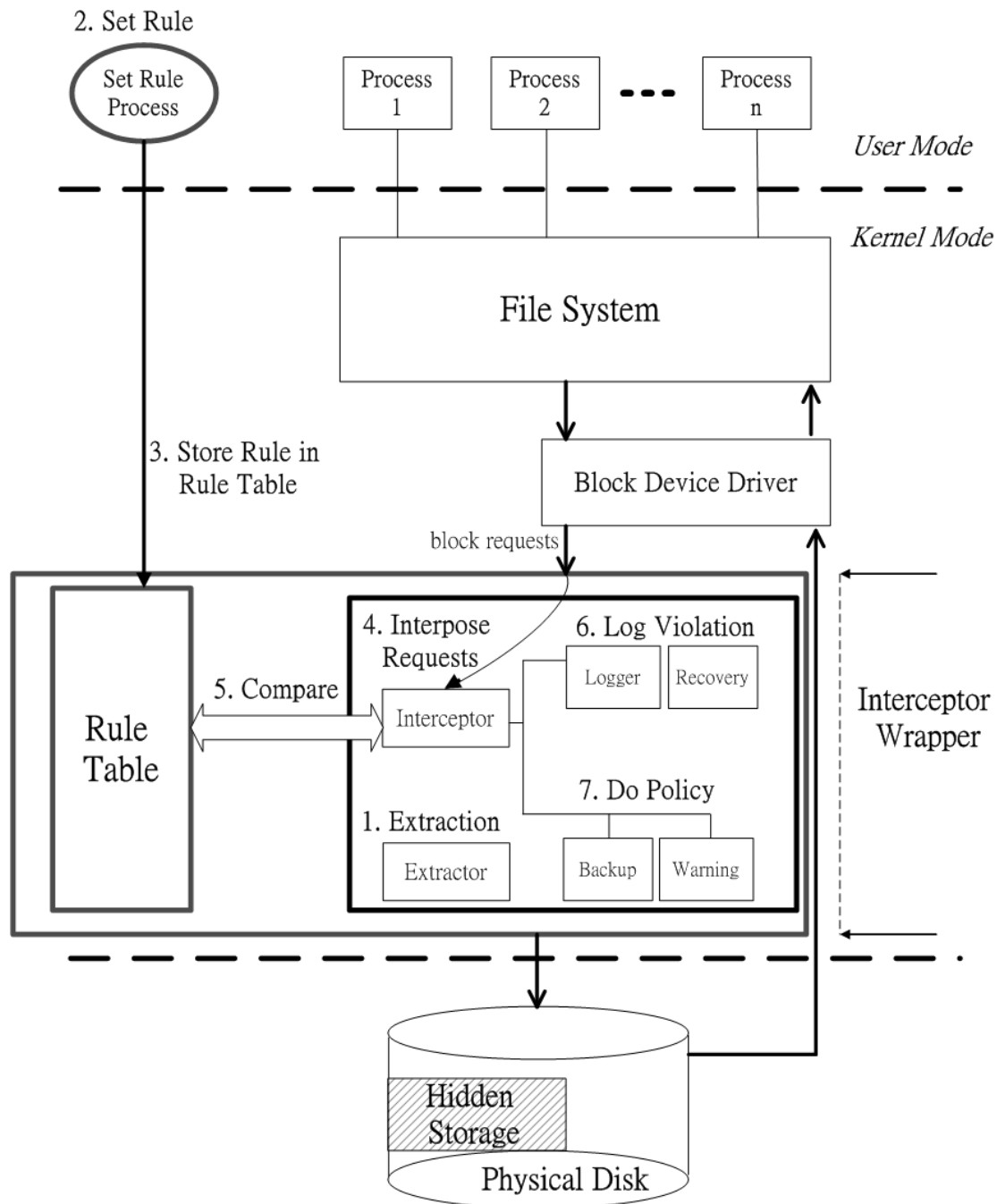


Figure 5: Infrastructure of SecDisk

The hidden storage showed in Figure 5 is false hidden storage. Because we do not implement in a real disk system, we allocate some space from the file system pretended as hidden storage. In the following, we describe detailed functionality and implementation processes of each component in our infrastructure.

4.1 Set Rule Process

The set-rule process is an interface for the administrator to communicate with the SecDisk. Its major functionality is to support a file-system environment for the administrator to set the protection rules. It performs two actions: the file-to-block translation and storing the rules into the rule table. In the following, we describe how we implement these two functions.

In the Ext2 file system, directories are implemented as a special kind of file whose data blocks store all the entries in that directory. Each entry (i.e., `ext2_dir_entry_2`) contains the corresponding inode number. Figure 6 shows an example of the data contents in the root directory. Given any file path for file-to-block translation (for easy explanation, we assume that the file path is `"/home1/usr/sam"`.), we get the root directory (inode number 2) and extract its inode fields to locate its data blocks. We compare the name following the root directory in the file path with each name field in the root directory entry to get the next entry's inode. In our example, the next entry is `"home1"` and we get its inode number 67. This step repeats until the inode of the target file, `"sam"`, is found. With the inode of the target file, we can get all the data block numbers of that file.

After getting the inode and data block numbers of a given file, we store the numbers with the corresponding rule in the rule table. This is done via the `SetRule()` system call implemented by us. In addition to `SetRule()` system call, we also implement `DelRule()` and `ListRule()` system calls for the administrator to manage rules in the rule table.

Inode #	rec_len	file_type	name_len	name							
35	12	1	2	.	\0	\0	\0				
36	12	2	2	.	.	\0	\0				
39	16	4	2	s	b	i	n				
67	16	5	2	h	o	m	e	1	\0	\0	\0

Figure 6: Entries of Root Directory

4.2 Interceptor Wrapper

Our protection mechanisms are implemented in the interceptor wrapper. In the following, we describe each component in the wrapper and their implementation details.

4.2.1 Rule Table

The rule table stores the block numbers (inodes and data blocks), permissions (no-read, no-write, and no-delete), and protection policies (warning or backup) of the protected files. As the files that need to be protected become more, the size of the rule table becomes larger. Therefore, doing comparison efficiently in such a large table is required, or the system performance will be degraded. With this performance consideration, the rule table is built up using hash tables. In our implementation, we use two hash tables for storing inode block numbers and data block numbers

respectively.

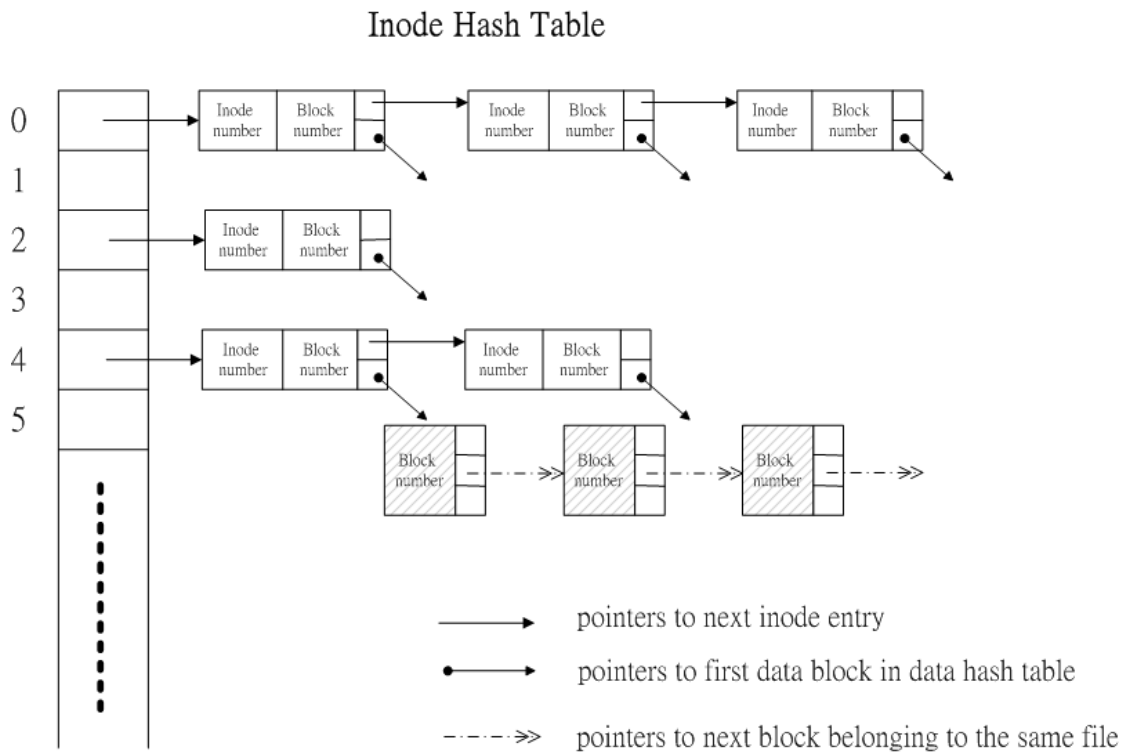


Figure 7: **Inode Hash Table**

Figure 7 shows an example of the inode hash table. Each entry means an inode and it points to its first data block. There are several inodes in a type of inode block. For efficiency of the searching, SecDisk records the inode number and its located block number of a protected file.

4.2.2 Interceptor and Extractor

The interceptor interposes the block requests and filters them. In our current implementation, SecDisk just needs to compare the block numbers that are located in the ranges of the inode and data blocks. Requests to all the other types of blocks are passed. That makes the comparison more efficient. In order to obtain this knowledge, SecDisk first performs on-disk layout discovery. That is the job of the extractor. As

we mentioned before, this is done by extracting the information from the superblock. With the on-disk layout knowledge, the interceptor can do the correct filtering when block requests come.

4.2.3 Logger

Through recording the actions of accessing the disk data, the administrator has clues to determine the damage caused by intruders. In our system, this is done by the logger but it does not log all of these actions. When a block request against the rule occurs, the logger logs this violation along with the file name that the block request number belongs to. It is very possible for intruders to modify the important system files when they attack the system. In SecDisk, most of the important system files are set protected. Thus, we can check this log content to see if our system is suffering attacks right now.



4.2.4 Protection Policy and Recovery

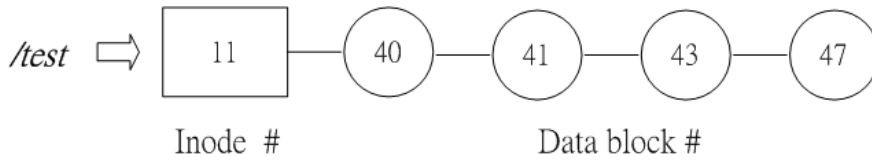
In this section, we describe the implementation of the two protection policies in the prototype system. For the warning policy, our ideal method is to deliver a warning sign to the console to notify the administrator that the protection rule is violated. In our current implementation, SecDisk just prints the messages into the system message buffer. In Linux, we can use the command, *dmesg*, to see the messages.

For the backup policy, in addition to the action of the warning policy, SecDisk backs up the protected blocks that are modified in the hidden space. As mentioned before, we allocate some space in advance from the disk and pretend it as the hidden space. Figure 8 shows an example of how the backup action works. First, we assume a file, named *test*, with inode number 11 and four data blocks, number 40, 41, 43, 47. If the blocks with number 41 and 43 are modified, we back up the original content of the

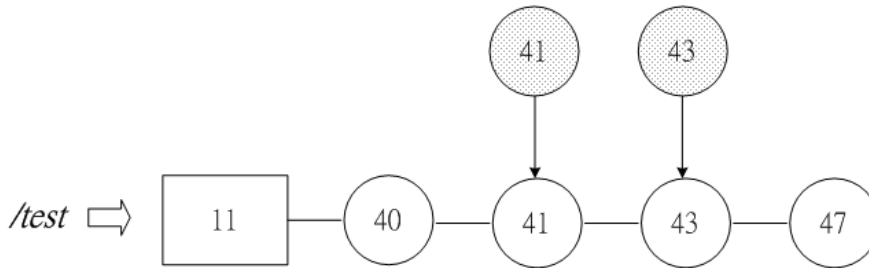
blocks in the hidden storage. For simplicity of management and recovery, SecDisk will create a new file with the same file path name of the original file. In this example, a file, named test, will be created in the root directory in the hidden space. . And the first two blocks of this file are the backup blocks. After backup actions, SecDisk correlates the new file with the original one. In our example, the modified blocks are number 41 and 43 and their corresponding positions of the original files are 2 and 3. Our backup mechanism is a block-based backup. That is, SecDisk backs up the modified blocks of a file, not backs up all blocks of the file. Therefore, SecDisk must record what block numbers of a file are backed up. To implement this, we use a data structure to record them to store the right positions of the blocks in the new file corresponding to the original file. In this example, SecDisk records that in the new file, its first block is the position 2 and its second block is the position 3. With this information, SecDisk can recover data from the hidden space correctly.

Recovery performs the inverse actions of the backup. This is the job of the recovery component. The administrator gives a file path that he wants to recover, and then SecDisk searches the hidden storage for that file. After finding it, SecDisk can recover the original data.

1. Original protected file state



2. Block number 43 and 41 are modified



3. Create a new file in the hidden storage and back up the data with block number 41 and 43

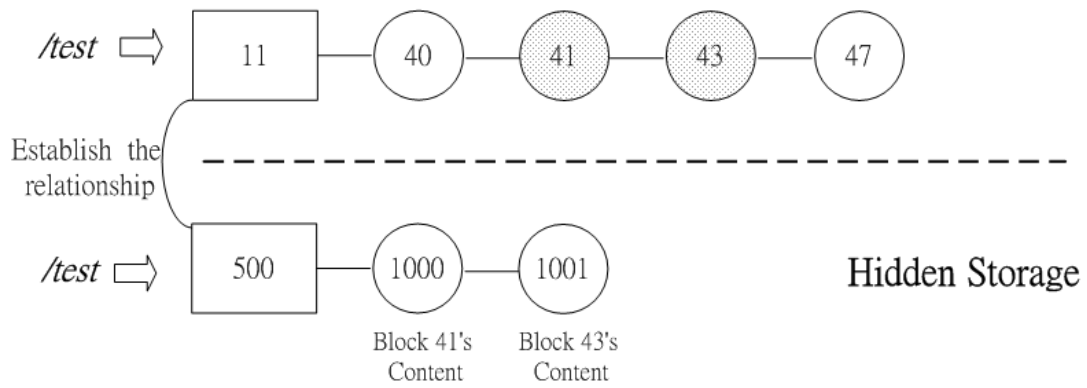


Figure 8: A Backup Example

5. Performance Evaluation

In this chapter, we use several experiments to measure the overhead of our prototype SecDisk system. These experiments aim to show that the overhead is acceptable and it is feasible to implement our mechanisms in a real disk system. Our machine is a host PC that is equipped with a Pentium 4 1.6GHz processor, 256MB DDRAM, and a ST340016A, 40G disk with 7200 RPM.

5.1 Installation Overhead

Before SecDisk can start to work, two things need to do: extracting on-disk layout and setting rule time. The former, as mentioned before, is just to locate the superblock and gain the information by extracting the superblock. This action is simple and fast, and thus it nearly costs no overhead.

We measure the setting rule time of various sets of files and Figure 9 shows the results. We give different sets of files, ranging from 1000 to 10000, and measure their translation time. The time ranges from 166 milliseconds to 2.589 seconds. We consider that generally, 10000 files are enough for the administrator to set and it is acceptable to take 2.589 seconds to translate them.

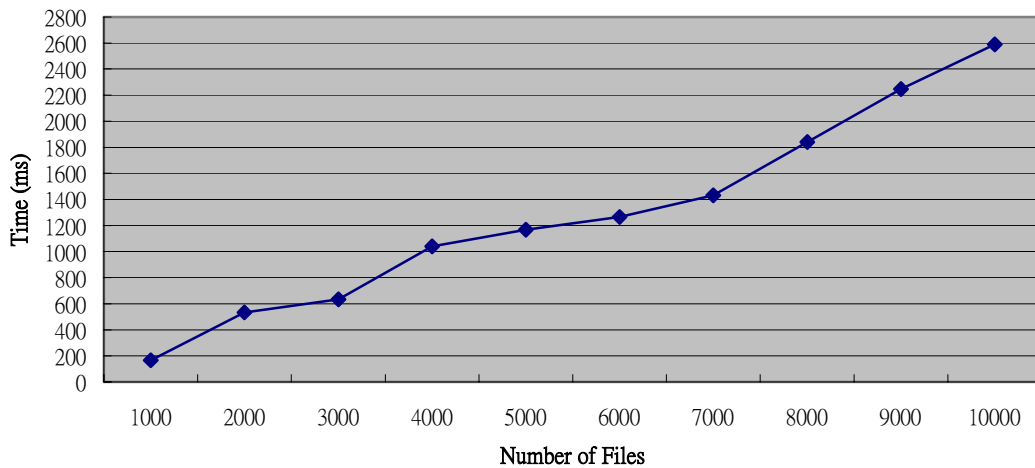
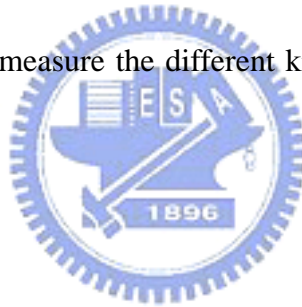


Figure 9: Set Rule Time

5.2 Runtime Overhead

In this section, we start to measure the different kinds of overhead when SecDisk runs.

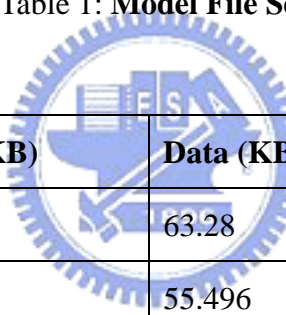


5.2.1 Memory Cost

First of all, we measure the memory cost of our data structure for storing the information of protected blocks. As showed in Table 1, we use these file sets as our experimental model and Table 2 shows the detailed information of these file sets. There are 8674 files and total size is 219124 KB. We evaluate the memory cost of each file set. In our SecDisk system, we use two hash tables for storing inode blocks and their data blocks. Their usage is showed in Table 3. If we select all of the file sets to protect, the most memory is used for data blocks and the total size is 2.335 MB. We think that this cost is acceptable if it is implemented in a real disk. The administrator can only select some files that he cares the most to protect.

File Sets	File Numbers	Size (KB)
/sbin	167	9040
/bin	72	7928
/usr/sbin	1809	115708
/usr/bin	243	83960
/lib	1625	39644
/boot	11	3824
/etc	1311	6332
/usr/include	3436	28582
	8674	295018

Table 1: **Model File Sets**




File sets	Inode (KB)	Data (KB)	Total (KB)
/sbin	4.676	63.28	67.956
/bin	2.016	55.496	57.512
/usr/sbin	78.625	809.956	888.581
/usr/bin	6.804	587.72	594.524
/lib	45.5	277.508	323.008
/boot	0.308	26.768	27.076
/etc	36.708	44.324	81.032
/usr/include	96.208	200.074	296.282
	270.845	2065.126	2335.971

Table 2: **Memory Cost**

5.2.2 Legal Access Overhead

In this subsection, we want to know what the overhead costs if our protection mechanism is on, but accessing behavior does not violate the rules. In this condition, the main overhead is the time that comparison the requested block with the ones in the rule table. We set our model file set (showed in Table 1) to protect. We use a benchmark, PostMark [PostMark], as our behavior model. PostMark is a file benchmark and is initially configured to create a number of file. When PostMark starts to run, it performs transactions, including read, write, append, create, and delete, to those files randomly, and report the transaction rate as the result. In our experiment, the environment is: 764 created, 243 read, 257 append, and 764 deleted. Table 4 shows the time. Without protection mechanism, it takes 39 seconds. With protection mechanism, it takes 41 seconds. It only cost 5% overhead in our experiment. It presents that in most conditions (without violating rules), SecDisk can work as well as the system without the protection mechanism.



Without Protection	With Protection
39 (seconds)	41 (seconds)

Table 3: **Time of PostMark**

5.2.3 Backup Overhead

In the following, we evaluate the overhead of backup mechanism. We use three experiments to display this.

5.2.3.1 The Backup Time of Different File Size

First, we evaluate the backup time of single file with different file size and Figure 10 shows it. In this experiment, our file size is presented by block numbers and one

block stands for 4096 bytes. Therefore, the experimental file size ranges from 1KB (1 block) to 1.6MB (400 blocks) and its cost time ranges from 0.17 second to 24.135 seconds. Our backup action is to backup modified blocks to hidden space. Accessing disk originally takes longer in the system and we think that this result is expectable.

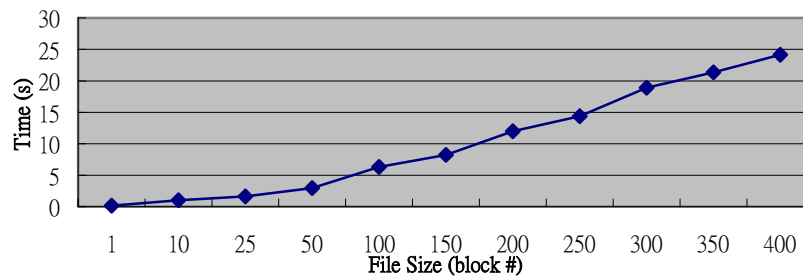


Figure 10: **Backup Time**

5.2.3.2 Make Kernel

In this experiment, we compare the performance between with backup actions and without backup actions and we use make kernel for our benchmark. In the Linux kernel source directories, there are about 11000 files in these directories and the total size is 160 MB. For convenient of measurement, we pick up some files, that is used while making kernel, from the different directories and set them with no-read permissions and backup policies. That is, in making kernel period, each file (*.c and *.h) read from the disks is backed up by our SecDisk. Figure 11 shows the result. The original making kernel time is 420 seconds and the overhead time ranges from 14.39 seconds to 133.36 seconds.

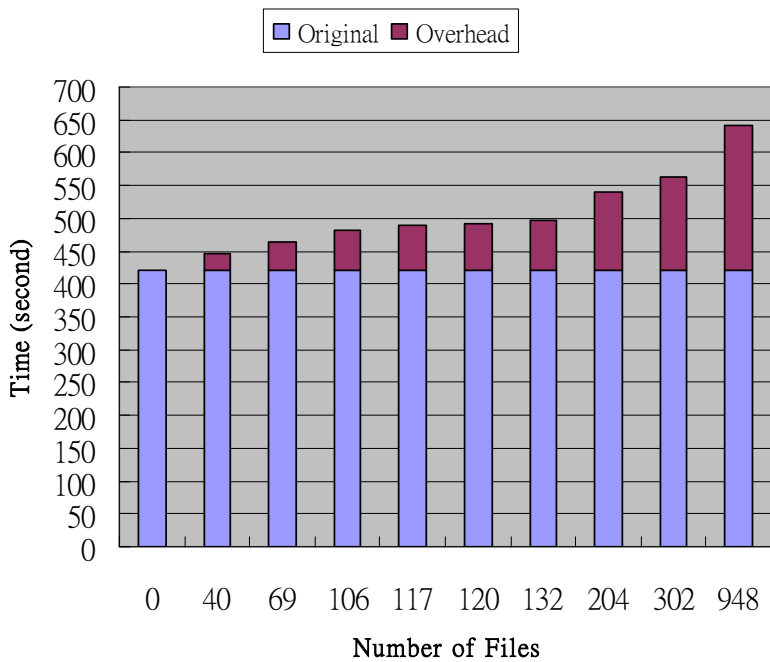


Figure 11: **Make Kernel**

5.2.3.3 Delete Files

In this experiment, we choose two file sets, etc and boot, to delete. These files are set with no-delete permissions and backup policies. Their information of numbers and size is listed in Table 5. The result is that although these two file sets have almost the same size, the time taking by deleting the boot file set is shorter than the time taking by deleting the etc file set. The reason is that the etc file set has more files than the boot set's and thus deleting the etc file set needs to take more time.

Delete Files	Number / Size (KB)	Time (s)
/etc/*	1311 / 6332	180.14
/boot/*	21 / 6164	74.212

Table 4: **Time of Deleting Files**

From the above three experiments, the overhead of the backup actions seems large and may degrade the performance. But the backup actions are taken when illegal actions violate the rules. That presents that the system may be under attack. Therefore, the degradation of the performance can notify the administrator of the possibility of the attacks. Besides, it is also able to delay the intruders' attack if the system is exactly under attack. With these two reasons, we think that the backup time is acceptable.



6. Conclusions and Future Work

In this thesis, we present a secure disk system (SecDisk) that has file system level knowledge to safeguard on-disk data. We design two policies, warning and backup, for the administrator to choose rule violation response. Main mechanism is to back up original files in disk reserved space that is not visible to high-level client system and therefore we can guarantee files security. Our contribution is that we design a local block interface disk system with data protection mechanisms. According to the experimental results, we consider that the overhead is acceptable and reasonable.

Some work remains for our future work. First, we consider that the mechanisms can be deployed in SAN. Because in SAN, exposed interfaces are also block interfaces. Therefore, we can have SAN's high throughput and data protection technology by combining our work into SAN. Second, besides data protection, we want to develop more advanced intrusion detection mechanisms that are workable in the disks with limited resources (computing power and memory). For example, detecting suspicious content, such as virus, is an advanced detection technique. Another advanced detection technique is content integrity. That is, the disk system understands the format of a file and can verify its format. For example, consider a system password file in Ext2 file system (`/etc/passwd`), it consists of a set of well-defined records. Records are delimited by a line-break and each record has some colon-separated fields and each of them is given a specific meaning. By checking these field meanings, we can discover attacks. These complicated intrusion detection techniques cost more computing power and memory and we should evaluate its feasibility and efficiency. That is, we want to put advanced intrusion detection mechanisms into disk systems without decreasing system performance significantly.

Reference

- [1] [active disk] A. Acharya, M. Uysal, J. Saltz, , “Active disks: programming model, algorithms and evaluation”, In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 81-91, October 1998.
- [2] [research in IDS] S. Axelsson, “Research in intrusion-detection systems: a survey”, Technical report 98-17, Department of Computer Engineering, Chalmers University of Technology, December 1998.
- [3] [Firewalls and Internet] B. Cheswick and S. Bellovin, Firewalls and Internet security: repelling the wily hacker. Addison-Wesley, Reading, Mass. and London, 1994.
- [4] [a sense of self for UNIX processes] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, “A Sense of Self for UNIX Processes”, IEEE Symposium on Security and Privacy, pp. 120-128, May 1996.
- [5] [storage bricks have arrive] J. Gray, “Storage Bricks Have Arrived”, Invited Talk at the First USENIX Conference on File and Storage Technology (FAST’02), January 2002.
- [6] [SAN] InfraStor Technologies Corp., “Introduction to SAN”, available at <http://www.infrastor.com/tech/SANTechIntro.htm>, 2004.
- [7] [Jeffrey Katcher](#), “PostMark: A New File System Benchmark”, Technical Report TR3022, Network Appliance, October 1997.
- [8] [object based] M. Mesnier, G. R. Ganger, E. Riedel, “Object-Based Storage”, IEEE Communication Magazine, pp. 84-90, August 2003.
- [9] [Detecting Insider Threats] N. Nguyen, P. Reiher, G. H. Kuenning, “Detecting Insider Threats by Monitoring System Call Activity”, IEEE Workshop on

Information Assurance, pp.45-52, June 2003.

- [10] [process profiling] Y. Okazaki, I. Sato, “A New Intrusion Detection Method based on Process Profiling”, IEEE Symposium on Applications and the Internet, pp. 82-91, February 2002.
- [11] [Bro] V. Paxson, “Bro: A system for detecting network intruders in real-time”, In Proceedings of the Seventh USENIX Security Symposium, pp. 31-51. January 1998.
- [12] [EMERALD] P. A. Porras and P. G. Neumann, “EMERALD: event monitoring enabling responses to anomalous live disturbances”, National Information Systems Security Conference, pp. 353-365, October 1997.
- [13] [active storage for large-scale data mining and multimedia] E. Riedel, G. Gibson, and C. Faloutsos, “Active Storage for Large-Scale Data Mining and Multimedia”, In Proceedings of the 24th Conference on Very Large Databases, pp. 62-73, August 1998.
- [14] [TimeStamp] A. Rubini, “Making System Calls from Kernel Space”, Linux Magazine, available at http://www.linux-mag.com/2000-11/gear_01.html, November 2000.
- [15] [semantically-smart disk system] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, “Semantically-Smart Disk Systems”, In Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003), pp. 73-88, April 2003.
- [16] [self-securing storage system] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A.N. Soules, G. R. Ganger, “Self-Securing Storage: Protecting Data in Compromised Systems”, In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, pp. 165-180, October, 2000.