The Hippocratic File System: Protecting Privacy in Networked Storage

Abstract

Privacy protection is increasingly difficult in today's information society. In this paper, we look at an important link in the chain of information protection: the file system, and propose mechanisms to enhance the disclosure control of personal data. The scheme, called the Hippocratic File System, stores personal data's purpose and use limitation as the data's label, propagates the label as the information flows from one place to another, and enforces the label to prevent accidental disclosures. We describe the design, implementation and experience with the Hippocratic file system. In particular, we highlight a deployment obstacle: "cross-invocation contamination" of legacy applications, and describe techniques to alleviate this problem.

1 Introduction

The growth of the Internet and the trend to conduct business electronically have resulted in an explosive growth in the amount of personal information stored online. Medical and financial records are now almost all stored online. Web sites routinely harvest surfing and purchase patterns for better targeting and business analytics. [Ols] Amid the growth in online personal information came the increased risk to individual privacy. Over the past few years, there has been a series of well-publicized privacy breaches. Below are a few example among many similar instances [HIP]:

- About 400 pages of detailed psychological records concerning visits and diagnoses of at least 62 children and teenagers were accidentally posted on the University of Montana Web site for eight days. (C. Piller, "Web Mishap: Kids' Psychological Files Posted," Los Angeles Times, November 7, 2001, p. A1)

- Eli Lilly and Co. inadvertently revealed over 600 patient e-mail addresses when it sent a message to every individual registered to receive reminders about taking Prozac. (R. O'Harrow, "Prozac Maker Reveals Patient E-Mail Addresses," The Washington Post, July 4, 2001, p. E1)

- A psychiatrist from New Hampshire was fined $1,000 for repeatedly looking at the medical records of an acquaintance without permission. ("Psychiatrist Convicted of Snooping in Records," The Associated Press State & Local Wire, May 5, 1999)

- Confidential Medicaid records were disclosed during the sale of surplus equipment by the Arkansas Department of Human Services twice in six months. In October 2001, the state stopped the sale of the department's surplus computer storage drives when it was discovered that Medicaid records that were supposed to be erased were found on the computers. ("DHS Surplus Sales Again Reveal Confidential Information," Associated Press, April 3, 2002)

Almost all of the breaches are due to human error of the insiders, rather than attacks by hackers from the outside.

One can argue that the current structure of file systems makes it easy to accidentally disclose confidential information:

- The use of Discretionary Access Control (DAC) puts the responsibility of setting proper permissions on files entirely in the hands of users, who often just use the default permissions on all files.

- Even if a sensitive file has its permission set properly, if the data in the file are transformed into a new file, the permission on the derived file is again set via discretionary access control (i.e. set by users).

- Aggregate files containing multiple people's personal data should have access control lists that are intersections of the individuals' access control lists, so that the aggregate won't be sent to any of the individual. Unfortunately, the aggregate file is again protected by discretionary access control only.

Existing techniques that enhance file system security unfortunately are not adequate. Improving the access

control at the file server can make it harder for certain clients to retrieve sensitive files. However, the file server has no visibility about the information flow once the file leaves the server. The file might be transformed or aggregated into other files, and the file server has no information to protect the derived files. Hence, *client-side information-flow tracking mechanisms are needed to protect sensitive information.*

Using a military style multi-level security (MLS) system would address the information flow issue. However, privacy data do not conform to military-style information model. In particular, a person's confidential data are secret to certain people, but not others. Traditional MLS system captures this aspect via compartments; however existing research mostly assume a finite number of predefined compartments. If one uses compartments to model individual's privacy data, the number of compartments is infinite.

Encrypting files is another way to enhance access control. In essence, access control is achieved by judiciously handing out decryption keys. The difficulty thus lies in key management. A key management scheme that can reflect rich group/subgroup relationships is difficult. Furthermore, when external communication is involved (e.g. emails or web site submissions), setting up the key management involves Public Key Infrastructures that are even harder to manage.

In this paper, we describe a set of client-side mechanisms that track information flow and enhance protection of personal data. Sensitive, personal data carry *labels*, which are instantiated purposes of such data, specifying for what purpose the data are used, and who can access or receive such data. The label *propagates* to all descendant files containing the data, implemented through a lineage tracking module in the operating system of the hosts where the data reside. The label is then *enforced* upon storage of data in networked file system, and transmission of data to other organization and individuals.

We call the system the Hippocratic file system, as it aspires to uphold the privacy protection tenet of the Hippocratic Oath: "And about whatever I may see or hear in treatment, or even without treatment, in the life of human beings – things that should not ever be blurted out outside — I will remain silent, holding such things to be unutterable." — Hippocratic Oath, 8. [Sta66].

A prototype Hippocratic file system has been implemented in Linux, and experience with it shows that a number of common productivity applications on Linux suffer from *cross-invocation contamination.* Cross-invocation contamination means that once the application open one labeled file, all future files written by the application carry the label, even if the application is terminated and then restarted. This is usually a problem due to the logic of the application. Since rewriting applications is not a viable option in most environments, we describe techniques to contain contamination with as little risk as possible to enabling unauthorized information disclosure.

The Hippocratic file system focuses on preventing accidental disclosure of personal data, not dedicated attempts at obtaining the information. Preventing dedicated attempts involves secure operating systems and network infrastructure, and is a much bigger issue than we can address in this paper.

Despite its name, the Hippocratic file system is not focused on protecting medical records, nor does it solve all the complex issues involving aggregation and access control of medical information. Rather, it's a design that aims to implement the four OECD principles in file systems, and apply to all personal data including e-commerce transactions, web surfing logs, mortgage applications, etc.

2 A Privacy-Conscious Infrastructure

The Hippocratic file system does not operate by itself. It works in tandem with other system components that are aware of care needed for personal data. Below we first describe principles upon which the Hippocratic file system is designed, then describe a simple usage scenario and introduce the other system components involved.

2.1 OECD Principles on Privacy

If one were to build a file system that enhance privacy protection, where should one start? One source of inspiration are the laws and regulations that many governments around the world passed to govern the collection, usage and transmission of personal information. Underlying these laws are a set of international consensus for privacy protection, the most well known of which is the OECD (Organization for Economic Co-operation and Development) guidelines [Org]. The OECD guidelines stipulate eight principles for handling privacy data, of which the following four are particularly relevant:

- *Purpose Specification*: The purposes for which personal data are collected should be specified not

later than at the time of data collection and the subsequent use limited to the fulfillment of those purposes or such others as are not incompatible with those purposes and as are specified on each occasion of change of purpose.

- *Use Limitation*: Personal data should not be disclosed, made available or otherwise used for purposes other than those specified in accordance with Purpose Specification except: a) with the consent of the data subject; or b) by the authority of law.
- *Security Safeguards*: Personal data should be protected by reasonable security safeguards against such risks as loss or unauthorized access, destruction, use, modification or disclosure of data.
- *Accountability*: A data controller should be accountable for complying with measures which give effect to the principles stated above.

The principles are policy statements specifying the standard of care for personal data. However, privacy protection cannot be implemented solely by legislative means; it needs technical enforcement.

As designers and builders of computer storage systems, the file system community has a unique responsibility for making sure that personal data are well protected and highly resistant to accidental disclosure. Our paper serves as a first step in this direction.

2.2 A Simple Usage Scenario

To illustrate the issues facing protection and communication of privacy data, we use a medical clinic as an example. The example does not necessarily resemble reality. A more realistic design can be found in Anderson's work. [And96]

The clinic uses database servers to hold various records, for example, patient information, insurance provider billing addresses, doctor's notes, etc. There are six tables in the database, with schema shown in Table 1. We assume that insurance providers have electronic billing methods, such as Web (HTTPS) or email (SMTP), and associated billing addresses, such as web site addresses or email addresses. Similarly, pharmacies may also electronic methods to call in a prescription, and associated electronic call-in addresses.

The administrators of the clinic access the database to carry out the daily tasks. Whenever an admin accesses personal data stored in the database, there is a purpose as to why he or she needs the personal data. In this example, there are the following possible purposes:

- *billing* for patient x: bill the insurance provider for procedure performed;
- *prescription_reminder* for patient x: send email to the patient to remind him or her to take prescriptions;
- *medical_history_request* for patient x: send a patient's medical history file to the patient, as per the patient's request;
- *daily_statistics*: collect statistics of medical procedures performed each day;

Note that the first three purposes are associated with specific patients, while the last is not.

The admin's activities are carried out on desktop machines, which are also used for the administrators' personal tasks, such as emailing friends and colleagues, instant messaging, and browsing the web. The desktop machines store data both locally and on a file server.

The challenge is designing the system such that the administrator does not accidentally place the doctor's notes in the public web directory, or email a patient's prescription information to the insurance provider.

2.3 Purpose Binding

Among the OECD principles, *Purpose Specification* and *Use Limitation* stipulate that when a data item is collected, its purpose must be specified to the user, and when a data item is used, its usage must be consistent with the purpose. While the file system is not involved with data collection, it is involved with data storage and use. The *Security Safeguards* principle thus mandates that the file system built in mechanisms to enforce use limitation and prevent accidental disclosure.

From the computer system's perspective, a "purpose" carries three constraints:

- authorized issuers: those who are authorized to carry out activities for the purpose. For example, while any of the clinic admins can carry out billing and prescription filling, only one designated person can respond to patient request of the medical history file due to the sensitivity of the information.
- allowed data: the kinds of personal data that the purpose can access, expressed as the allowed tables and allowed attributes in the tables.
- allowed viewers and recipients: the list of internal users and external recipients who can view the

Table	Attributes
patient_info	patient-id, name, address, email-address, insurance-provider-id, insured-id, pharmacy-id
procedure_info	patient-id, doctor, procedure-code, date
prescriptions	patient-id, doctor, drug-name, dosage, refile-cycle, num-refills, date
doctor_notes	patient-id, doctor, date, notes
insurance_provider	id, name, address, electronic-billing-method, electronic-billing-address
pharmacy	id, name, address, call-in-method, call-in-address

Table 1: Schema of database relations that store patients' personal information in the example clinic.

Purpose	**billing for patient id** x
Authorized Issuers	anyone in the billing office (i.e. group *finance*)
Allowed Data	insurance_provider.*, procedure_info.*, patient_info.insurance-provider-id(x), patient_info.insured-id(x)
Allowed Recipients	group *finance*, insurance_provider.electronic-billing-address(patient_info.insurance-provider-id(x)) with method insurance_provider.electronic-billing-method(patient_info.insurance-provider-id(x))
Purpose	**prescription reminder for patient id** x
Authorized Issuers	anyone in the IT office (i.e. group *IT*)
Allowed Data	patient_info.*(x), prescriptions.*(x)
Allowed Recipients	group *IT*, group *doctors*, group *nurses* patient_info.email-address(x) via SMTP
Purpose	**medical history request for patient** x
Authorized Issuers	the designated *patient representative* in the clinic
Allowed Data	patient_info.*(x), prescriptions.*(x), doctor_notes.*(x)
Allowed Recipients	the designated "patient representative", patient_info.email-address(x) via SMTP
Purpose	**daily statistics**
Authorized Issuers	anyone in the IT office (i.e. group *IT*)
Allowed Data	procedure_info.doctor, procedure_info.procedure-code, procedure_info.date
Allowed Recipients	everyone in the medical office

Table 2: Semantics of the purposes in terms of access control. Notations such as insurance_provider.* means that all columns in the table insurance_provider can be accessed. Note that the allowed recipient is frequently a function of the subject (i.e. patient) of the purpose. For example, patient_info.email-address(x) means the email-address column of the patient_info table for the row where patient Id is x.

data. By internal users, we mean the users who have an account in the computer system. In general, the allowed viewers always include at least the issuer of the activity. External recipients are indicated by a means of communication and a communication address. Note that recipients often are functions of the subject (i.e. patient) for whom the activity is carried out.

The constraints on the four purposes in the clinic example are listed in Table 2.

To implement use limitations, each purpose can only access the data that it is allowed to, and once accessed, the data must carry the limitation of that purpose around. In other words, data (i.e. files) need to be associated with the purpose and the access control specified by the purpose.

2.4 Database Support

The Hippocratic file system cannot enforce use limitation by itself. Enforcing that only authorized issuers can issue a "purpose" and that a purpose can only access allowed data is done by the database system.

Fortunately, the database community has long noticed the need of privacy protection and designed the Hippocratic Databases architecture [AKSX02]. Hippocratic databases are also organized around purposes, and understand the constraints associated with each purpose. The Hippocratic Database requires that every query presented to the database state its purpose, and only data allowed for the purpose be retrieved.

We assume that the Hippocratic databases are installed at the clinic, and can consult an authentication and authorization server to verify that: a) the user issuing the query is the claimed user, and b) the user is authorized to issue queries for the specified purpose. The retrieved results from the database are stored in a file on the desktop machine. We further assume that the program that retrieve data for a purpose will also look up the database to obtain the list of allowed viewers and recipients, and store the list as an "read control list" (RCL) for the data.

However, the database protection mechanism stops the moment the data leave the database server and reside on the desktop machine. From there on, Hippocratic file system takes over the protection of the data.

3 Design Overview

In Hippocratic file system, files carry security labels with them, stored as extended attribute of the file. Two set of mechanisms, label propagation and label enforcements, manipulates and interprets the labels on files.

3.1 Labels

In the Hippocratic file system, every file containing personal information has a label on it. Initial labels are assigned by a trusted application, which retrieves personal data from the database and then assign labels to the file containing the data. The application is assumed to be examined and verified, as commonly done in the Clark&Wilson commercial security framework. [CW87]

A label is an *instantiated* purpose, consisting of the purpose and instantiated Read-Control-List (RCL) of the purpose. The trusted application that query the database to carry out a particular purpose for a subject (e.g. patient) also query the database to obtain the list of allowed viewers and external recipients. The RCL is *instantiated* because the list depends on the subject and the trusted application looks up the database to determine the list's contents. Example labels are *billing* + {*Read/finance-group, HTTPS/billing.blueshield.com:443*}, *prescription reminder* + {*Read/medical-staff, SMTP/john.doe@yahoo.com*}, and *daily_statistics* + {*Read/clinic-everyone*}. Note that the RCLs carry both file system access restrictions, but also communication restrictions.

Only a limited number of verified and trusted application can assign or change labels on a file. Besides the database query application, other applications that declassify or encrypt the data should be allowed to rewrite or delete labels on a file. All such applications must be verified and trusted, similar to how trusted programs are handled in Clark&Wilson, or trusted subjects in multi-level secure systems.

3.2 Label Propagation

Label propagation ensures any file whose content might be causally affected by sensitive files carries a label that reflects the sensitivity. In other words, the label should go where the information propagates to.

However, since label propagation is a kernel mechanism, its determination of causal relations is necessarily conservative, like all access control mechanisms that try to constrain information flows while treating

the application as a black box. In other words, if the kernel sees that a process reads data from file A and then writes data to file B, the kernel has to assume that file B contain information from file A. Of course, a finer level of observation, such as at the memory page level [KEF+05], can eliminate certain false correlations, as can knowledge of a program's properties, e.g. via Proof-Carrying Code.

The task performed by label propagation is thus the following: for each file that is written by a process, assign a label that is the combination of the labels of all files that have been read by the process.

3.2.1 Label Combination Function

The label combination function has two parts: generating a new purpose, and generating a new read-control-list.

Combining RCLs The rule for generating a new real-control-list is simple: the new RCL is the intersection of all the input label's RCLs. Note that this set cannot be empty since the user who runs the process has read access to all the files involved, thus the user is on the RCLs of all input labels.

Following this rule, label propagation thus has the following simple property.

Theorem. As the data propagate through the file system, the set of users and external recipients never expands.

Combining Labels The rule determining the new "purpose" is more complex. The semantics of combining data of multiple purposes depends on the semantics of the purposes themselves. In our view, a "purpose" has two aspects associated with it: a sensitivity level of the data, and the "conflict-of-interests" constraints.

One can have a completely-ordered "sensitivity levels" and place purposes on these levels. For example, in our clinic example, there are two sensitivity levels, *medical_history_request* at the higher sensitivity level while the rest are at the lower level. For each sensitivity level that has more than one purpose, we associate a synthetic purpose indicating that the file combines data from multiple purposes from the level. When purposes combine, the result should be of the highest sensitivity level among the purposes.

To reflect "conflict of interests" relationships, our current proposal is to use manually specified combination functions. If there are "conflict of interests" relationships between the purposes, then the output purpose ought to reflect the combined "conflict of interests" restrictions of the input purposes. If there doesn't exist a purpose that reflect such restrictions, then a new "synthetic" purpose should be generated to reflect such restrictions. Modeling "conflict of interest" relationships in an easy-to-use fashion is outside the scope of this paper, and is part of our future work.

Taking the above considerations into account, here are the rules that determine the combined purpose:

- If all the input labels have the same purpose, then the output purpose is the input purpose;
- If the security officer specified a combination function of the purposes, then that combination function is used to generate the new purpose.
- The purpose with the highest sensitivity level is chosen as the output purpose, if there is only one purpose at that level. If there are more than one purpose at the highest sensitive level, then the synthetic purpose for that level is used.

Difference from Multi-Level Security How is the above scheme different from traditional multi-level security such as Bell-LaPudula? The difference lies in the fact that in addition to purposes, the RCL is used for use limitation. Unlike multi-level security scheme such as Bell-LaPudula, purposes here are specific to individuals, and it may occur that data of purpose A for individual x are combined with data of purpose B for individual y. In this case, we rely on the intersection of RCL to make sure that neither x or y can view the combined data.

In a lattice model, one can view our scheme as being the cross-product of two lattices. One lattice is generated by sensitivity levels. The other lattice is the infinite lattice formed over sets, with one set dominating another if the former is a subset of the latter, and the common dominating set of two sets is the intersection of the two.

3.2.2 Special Domains of Trusted Applications

While the above rules specify the default behavior of applications, the combination function needs to change if the application is a special one, for example, an encryptor that allows the sensitive data to be transmitted over the network.

To accommodate the special applications, we borrow the idea of domains from Domain-Type en-

forcement (the security implementation framework in SELinux) [BK85, BSS+95], and place the special applications in a separate domain with its own label generation function. For example, a de-classifier domain holds the encryptor program and other programs that delete sensitive information, and all files generated from that domain have no label, i.e. its label generation function generates a NULL label.

In summary, the rules of label propagation are the following:

- Special domains: if the executable of the process belongs to a special domain, then the output file carries the label that is generated by the domain's label generation function;
- Regular domains: if the executable of the process does not belong to a special domain, then the rules described above are used to determine the label on the output file;

Note that all applications placed in the special domains need to be verified and trusted.

3.3 Label Enforcement

Labels need to be enforced in three areas: access to data residing in the machine's local file system, access to data residing in file servers, and communication of data to external world.

3.3.1 Local File System Enforcement

For a labeled file on local disk, the kernel needs to make sure that not only the file's permission bits or ACLs are enforced when the file is read, but also the RCL in the file's label are enforced. The check is performed upon every file read; the check should only incur overhead if the file's label is changed.

3.3.2 Network File System Enforcement

In network file systems, the labels are stored at the file server as extended attributes of files. We note that many commonly used file systems, such as CIFS and AFS, all support extended attributes.

When the file is stored at a file server via a distributed file system protocol, the kernel further changes the file's ACL such that it is an intersection of the ACL specified by the user and the restriction specified by the label. That is, when a file is newly created, instead of its ACL being the default access permissions in the user's profile, the file's ACL is the intersection of the default and the label's restriction list. Similarly, a request to change the file's ACL, if generated by the local machine that retrieved the data to begin with, will also be enforced by the machine's kernel so that the resulting ACL is an intersection of the request and the label's restrictions. With this mechanism in place, a file containing personal data cannot be accidentally put on the web, since the access permission on the file will never be set to world-readable.

Care needs to be taken when label propagation interacts with client-side caches, particularly write-back caches. If a file's label is changed when it is written and as a result, the file's ACL is changed, the change in file's ACL must be sent to the file server before the dirty data in the file. Even with this precaution, systems such as NFS V2/V3 have a vulnerability, since they allow a client to cache a stale file attribute for up to 60 seconds. Thus, information leakage may occur if the client writes back the dirty data to the server and the client caching the stale attribute happens to be missing those dirty data in its cache. This is a fundamental problem with NFS V2/V3 that we will not attempt to solve in the Hippocratic file system.

The approach described above relies on client-side kernel mechanisms, and is only effective in environments where the client machines all support the Hippocratic file system. This is the case in environments such as financial and medical institutions where IT personnel makes sure that all clients use the same kernel, are configured uniformly, and perform nightly checkup on the clients. The advantage of this approach is that it doesn't require file server modifications. For environments where there are clients that do not support the Hippocratic file system, the file server should not permit such clients receive labeled files.

3.3.3 Communication Enforcement

For files that need to sent out to an external party, for example, submitted via HTTP to an external web site, or sent via email (SMTP) to an external email address, label enforcement employs "proxies" to intercept the communication and verify that the recipient address is on the label's restriction list. If a socket becomes labeled, its content is automatically intercepted by a proxy process running on the desktop machine. The proxy process understand every high-level protocols used for communication of sensitive data, including HTTP, HTTPS and SMTP, and can interpret the commands to understand where on the Internet the data is going to, and whether that destination is permitted by

the label. In the case of HTTPS, the proxy can only determine if the destination hostname or IP address is allowed, not the actual URL.

This approach requires that the proxy understands the common protocols used to send privacy data to other third parties. An alternative would be to use encryption, which requires keys to be set up before hand among the parties so that only a party holding the key can read the data. While the encryption approach is definitely more secure, in practice the setup is too complicated for many types of communications, notably email messages to individual users. Since our focus is on preventing human error in breach of individual privacy, we feel that using a proxy is a more suitable approach.

Due to time limitations, we have not implemented the proxy based solution in this paper.

3.4 Preventing Human Errors

We argue that, with label propagation and label enforcement, many of the errors made in the privacy breaches mentioned in Section 1 can be avoided:

- Medical records can no longer be posted on the web accidentally. Both on the local machine and on the file server, a file containing medical records has a label restricting those who can read it. The label enforcement mechanisms described above will make sure that the file's access permission can never be changed to "readable by the world" accidentally.
- Any user who is not allowed to view a patient's file cannot "accidentally" gain read permission when the data in the file is copied around or included in other files. Label propagation makes sure that the original restriction can only be strengthened with derived data, not weakened.
- Group email messages containing many patient's email addresses cannot be sent out. The process generating the group email reads the prescription reminder data of many patients. Since the combined RCL is the intersection of all input RCLs, the email that this process generates cannot be sent to any patient. The constraint thus force the system to send email messages by reading one patient's information at a time and send an email message to that patient only.

Of course, the Hippocratic file system is not a panacea for privacy breaches. For example, the scheme does not prevent the sale of disk drives containing personal data. In this case, good physical security and data scrubbing processes are needed. However, by propagating labels, the Hippocratic file system helps other mechanisms to catch all places where sensitive data may reside in stable storage. [CPG+04]

4 Implementation Details

We have implemented a prototype of the Hippocratic file system as a security module in Linux 2.6. The prototype implementation uses ASCII to represent the purpose and the RCL in a label. A production system is likely to use a representation that can be interpreted by other components such as the Hippocratic database. After the module is loaded, a trusted user-level application specifies the combination function via `/proc` to the security module.

4.1 Implementing Label Propagation via LSM

We implemented label propagation on a prototype Linux system using the Linux Security Module (LSM) infrastructure [Ope]. LSM is a flexible mechanism for adding security and access control mechanisms to the kernel and is currently best known for its use by SELinux, an extension to Linux that implements mandatory access control. It allows different security modules to be implemented without kernel modifications. LSM is ideal for our purposes because it adds hooks in all places where information flow might occur. Using LSM, we implemented label propagation as a security module that can be loaded into any 2.6 Linux kernel.

Generally speaking, LSM support in the kernel works by adding an extra pointer field to data structures of kernel objects [SFV], including `struct task_struct` (for processes and threads), `struct inode` and `struct file` (for files, pipes and sockets), `struct kern_ipc_perm` (for System V IPC), etc. The pointer field is used to store module-specific security information. LSM also adds calls to module-provided authorization functions at access control decision points, for example, inode creation, file open/read/write, process creation, etc. The security module functions are called before operations are performed and return a value indicating whether the operation should be performed or denied. The contents of the security structures and implementation of the authorization functions are entirely up to the particular LSM module.

Storing Labels In the Hippocratic file system, the security module keeps labels for files, processes, System V IPC mechanisms, and other types of files (named pipes, UNIX domain sockets, sockets, etc.). The labels for processes are used to aid in the implementation of label propagation.

There are two versions of labels stored for on-disk files. The in-memory version is associated with the `struct inode`. We decide to use `struct inode` instead of `struct file`, because two programs opening the same on-disk file have two separate `struct file`, while every `struct inode` always uniquely corresponds to one particular on-disk file. To ensure that two programs opening the same file see label changes to that file caused by the other program, we store a pointer to the label in the `struct inode` for the file. The on-disk version of the label is stored as extended attributes, which are supported by all common Linux file systems and is also the choice of label storage in SELinux. The label is stored as an extended attribute called `security.Hippocratic`.

For processes, the pointer to the label is stored in `struct task` of the thread group leader. In Linux 2.6, tasks are threads that can potentially share address spaces with other threads. A POSIX process is represented by a group of tasks (often just one), one of which is the so-called task group leader. Since the shared address space is also an information conduit, label propagation needs to work at the granularity of processes, rather than threads. Thus, labels are tracked in the thread group leader, instead of every task/thread. In other words, for a `struct task_struct *task`, the pointer to the label is `task->group_leader->security`.

Propagating Labels On every file read, the module checks if the label of the file is different from the current label of the process. If so, the combination function is applied to the file's label and the process's label (which can be NULL if the process doesn't have a label) and the result is set to be the new label of the process. On every file write, the module checks if the label of the process is the same as the label of the file. If not, then the combination is applied to the process's label and the file's label (when can be NULL if the file is newly created) and the result is set to be the new label for the file. Thus, no action is taken when a file's label is changed. Instead, the propagation occurs next time the information flows out of the file (i.e. read) into a process, and then flown out of the process (i.e. written) into another file.

Since information flow must occur via actions of certain process running on the CPU (i.e. disks and memory do not perform information flow by themselves), the above method of implementation is correct.

For applications in the special domain, the label for their processes carries a special flag indicating that label propagation should stop, and also contains the result label for all files written by applications in the domain (for example, NULL if the domain consists of encryptors). If the kernel detects that the process carries the special flag, it takes the label stored in the process's security information and assign it as the new label of any file written by the process.

Currently, any change in a file's label is written to disk synchronously. An alternative higher-performance scheme would be to always flush the change in the file's extended attribute before the file's data. However, we have not yet implemented it.

Assigning Labels Trusted applications can assign or change the labels of files using the setxattr system call. LSM provides a function `inode_setxattr` that is called from the setxattr system call. It is generally used to restrict user programs from changing arbitrary file labels and thereby bypassing the kernel security mechanism. We use it to check if program is trying to change the `security.Hippocratic` label and if this is the case and the operation is granted, we set the label inside `struct inode` to the new label provided by the user program.

Label Contribution Log For usability of the system, there needs to be tools that let users query how a file acquires a label. To aid such tools, the module keeps a log of label contributions. The log has three types of entries:

- Label transition through inheritance: "Fork: <label of parent process>, <parent pid>, <child pid>".
- Label transition from input file to process: "Read: <process pid>, <label of input file>, <new label for the process>, <inode-number>, <file path name>".
- Label transition from process to output file: "Write: <process pid>, <label of the process>, <new label for the file>, <inode-number>, <file path name>".

The module outputs a log line upon the following cases: every fork, every file write that leads to a label change

on the file, every file read that leads to a label change on the process, and every file read that leads to a new file contributing to the process' label. For each file, the module keeps a buffer in the security structure associated with each file to store the file's pathname, which is filled when the file is opened. The buffer is deallocated with the rest of the security structure when the file is closed. The log is output by the security module through `/proc`; a user level daemon reads the data and write to log files stored in the local file system.

The label contribution log needs to be cleaned every time a labeled file is deleted from the storage. Due to time limitations, we have not implemented the garbage collection of the logs. Here is a brief description of the algorithm that we plan to use. The log file is traversed to build a tree, whose nodes are files or processes. An edge exist from u to v if v is the only source of label contribution to u. Then all lines associated with every node in the tree can be deleted from the log. The cleaning process ensures that the log only contains information related to existing labeled files.

4.2 Implementing Label Enforcements

Label enforcement on the local machine is implemented in "file_permission", which is called on every file read and write, with an argument indicating whether the user is attempting to read or write. If the call is for file read, the function checks if the effective uid or gid is listed in the RCL in the file's label, retrieved via pointer in struct inode of the file. If not, the read operation fails. No checks are done for file write, since we are only interested in protecting confidentiality, not integrity.

We have only implemented label enforcement for the NFS v3 file system. Though NFS v3 doesn't support extended attribute natively, we faked it by using a special dot-named file associated with the main file. The file permission change for NFS is performed when the file's label is changed. Whenever the label is changed, the kernel tries to write the label to the extended attribute of the file. At this point, the file_system_type structure associated with the file system of the file is consulted. If it is a local file system, then the label is stored as an extended attribute. If it is NFS, it is stored in a special dot-named file associated with the original file. The UNIX permission bits for the file is then checked. If there is a label associated with the file, then the read permission for `other` is turned off. If the gid is not in the RCL of the label, then the read permission for `group` is turned off. If the check results in permission bits changes, an RPC is issued to update the file server immediately.

Kernel compilation	Base	HFS	Overhead
real	92m20s	92m21	0%
user	78m45s	78m52	0%
sys	10m26s	10m51	3%

Table 4: Linux 2.6.11 kernel compilation.

4.3 Performance Experiments

Following the example of SELinux [LS01], we measure the performance overhead of the implementation via both microbenchmarks and macrobenchmarks. Due to time limitations, we use only UnixBench [Nie] as the microbenchmark, and kernel compilation as the macrobenchmark. We are continuing performance optimization of the implementation and performance measurement with more benchmarks. We ran all experiments on a Pentium 3 with 256MB of RAM and a 1Ghz Processor.

Microbenchmark Results We run UnixBench under three systems. The first is an unmodified Linux 2.6.11 kernel. The second is Linux 2.6.11 kernel with the Hippocratic File System (HFS) security module, but the module is instrumented such that labels change on every file read and write. The performance in this case serves as the worst case scenario of HFS's performance, since in practice there are few long chains of information flow and label propagation converges quickly. The third is instrumented HFS such that labels change every time a file is opened. The third system characterize cases when working with new sensitive data. In regular use, label change becomes less frequent and performance gets closer to the baseline case. The results are shown in Table 3.

Macrobenchmark Results We also timed our system on compilation of the 2.6.11 Linux Kernel. We ran our experiments in single user mode right after booting after having previously performed a `make clean`. We ran `time make` after putting different labels on about 10% of the source files which meant that every file written by `make` would have a label and that labels would change quite frequently. This is an example of a worse-than-average workload when working with sensitive data. The results are shown in Table 4.

Since we observed virtually no slow down we did not run any tests for files without labels. These results confirm the absence of noticeable slowdown that we have noticed during development even on slow hardware.

Microbenchmark	Base	HFS (worst case)	Overhead	HFS	Overhead
file copy 4KB	51668	34386	50%	38604	30%
file copy 1KB	67987	27645	250%	58838	15%
file copy 256B	34227	8746	400%	25412	35%
pipe throughput	404101	374748	10%	387534	4%
pipe switching	145014	138194	7%	139785	4%
shell scripts (8)	193	176	10%	185	4%

Table 3: UnixBench system microbenchmark results.

5 Handling Cross-Invocation Contamination

An obstacle to deploying the Hippocratic file system in a real environment is the issue of cross-invocation contamination. Applications may use helper files such as history files or log files that are written every time the application runs. Once the application is used on one sensitive file, the helper files acquire labels. In subsequent invocation of the application, the helper files then propagate the labels to other files, and "contaminates" them. The problem is also recognized in general Multi-Level Security (MLS) systems, phrased as "after a while, everything flows to the top (level of secrecy)".

While it's easy to dismiss the problem by demanding that applications be rewritten to eliminate the use of helper files, we do not deem such options practical. Instead, we take the view point that the behavior of common applications can be observed and contaminations through them can be eliminated by *white-listing* certain files. The drawback of this strategy is that bugs in those applications may compromise the disclosure protection. However, we note that: a) the approach does not make the situation worse than the current system, which has no label propagations at all; and b) without white-listing, we may end up with a system that is simply not usable.

5.1 Experience with Common Linux Applications

Since Linux is our implementation platform, we investigate the scope of the contamination problem with common Linux applications,

- *Firefox* has two files that act as sources of cross-invocation contamination: `downloads.rdf`, which keeps the download listing, and `localstore.rdf`, which is the configuration file for the GUI.
- *OpenOffice* contaminates through configuration files and caches. If the user changes the configuration of Openoffice while reading a file with label, then the label propagates to the configuration file.
- *Emacs* has no cross-invocation contamination.
- *vim* has cross-invocation contamination through its configuration file `$HOME/.viminfo`.
- *gedit*, the text editor for Gnome desktop environment, can contaminate through the history file `gedit-metadata.xml` and configuration file `gedit-2`.
- *gcc* has no cross-invocation contamination.
- *LaTeX* has no cross-invocation contamination.

The trend is that applications with more features, particular features involving "history", are at the most risk for cross-invocation contamination.

5.2 White-Listing Helper Files

To contain cross-invocation contamination, the Hippocratic file system allows files to be white-listed. White-listed files would not have labels propagate to them. However, for a file to be white-listed, it must have an owner application, and the system enforces that only the owner application can write to the file.

The owner application must belong to a special domain called *white-listed applications*. These applications are not trusted to assign labels to files. Rather, they are trusted that, under normal operations, they do not disclose information contained in input files to the white-listed file, and thus the labels do not propagate to the white-listed file. This trust can be established through code examination, manual observation of application behavior, or certification of application vendors. It is expected that the security officer of the organization should examine commonly used applications and white-list files used by these applications that cause cross-invocation contamination.

A white-listed file can have a reader restriction. If imposed, the white-listed file can only be read by its owner application and applications in special domains (i.e. the encryptor domain and the file system maintenance domain). The reader restriction will further limit the chances of disclosure via the white-listed file.

White-listed files with reader restrictions are intended for an application's own files, such as history files, log files and configuration files. These files are only intended to be used by the application. White-listed files without reader restriction can be used for system files such as `/var/run/utmp` and `/etc/ld.so.cache`, which might be read by many applications. The fact that only utmp can write to the file limits the chance of information disclosure to the file.

Only a trusted program, the *white-lister*, can specify white-listed files. The interface for white-listing a file takes as arguments the path to the file and the pathname of the owner application. If the white-listed file is reader restricted, the interface also include the list of special domains whose applications can read the file. The information is then stored as a special extended attribute, which is interpreted by the `security.Hippocratic` module. If an application has white-listed files, then the white-lister needs to be invoked everytime the application is installed.

5.3 The Decontaminator

Handling contamination also requires the existence of a *decontaminator*. That is, despite efforts to inspect applications and white-list helper files, there bound to be cases when contamination happens and a user want to request that files with labeled be reversed. Tools should exist such that the user only specifies one file for decontamination (the origin file), and the system takes care of eliminating labels on files who have acquired a label solely due to the origin file.

The Hippocratic file system provides a *decontaminator* tool. The tool uses the label contribution log to identify all files that need to be cleaned. Specifically, when the de-contaminator tool is invoked, it takes a snapshot of the label contribution log, then reads each log line to construct a tree whose nodes are files and whose root is the origin file. An edge from u to v exists in the tree if u's label is the only source of v's label. The decontaminate then resets the label of all nodes in the tree. We are in the process of implementing the de-contaminator tool.

6 Related Work

Comparison with Existing Security Frameworks The best known lattice-based access control framework is multi-level security (MLS), as first developed by Bell and LaPadula. [BL76] In the original BLP definition, mechanism and policy were tightly intertwined: the policy of "no downward information flow" was decomposed into two safety properties (no read up – the simple security property, and no write down – the *-property). In MLS systems, the lattice of security labels is the cross product of a totally ordered set of levels (e.g., Unclassified, Secret, Top Secret) and an unordered set of compartments (e.g., NUCLEAR, CRYPTO, etc.). Implementing MLS was the first major use of mandatory access control (MAC), where the system, via a security officer, enforces a policy regardless of the individual user's actions. MAC is the key concept for enforcing confidentiality in the MLS setting, and privacy in our setting: users simply aren't allowed to break certain rules. While Bell-LaPadula is focused on confidentiality, Biba [Bib75] showed that integrity can be seen as the dual (in a lattice theoretic sense) of confidentiality, albeit over a different lattice.

Type Enforcement [BK85], and its generalization, Domain and Type Enforcement [BSS+95] is a mechanism for enforcing MAC without baking a policy into the mechanism. Instead, several matrices are used to configure the policy in Type Enforcement, and Domain Type Enforcement adds a policy specification language. While MLS was the motivating example policy for type enforcement, the matrices themselves do not encode any fixed rules, a la "must go upwards in a lattice."

Stephen Weeks' paper, "Understanding Trust Management Systems," [Wee01] offers a general framework that explains why lattices are good models for access controls. Weeks has a hierarchy of privileges, much like this work, and uses properties of lattices to produce sensible answers to all access control queries. Weeks considers the distinguishing feature of trust management (vs. other forms of access control) to be the search for a solution to whether this access is allowed, rather than a simple yes-or-no answer.

The Clark-Wilson model [CW87] focused on integrity as the key security property for for commercial applications, rather than confidentiality. The model focuses on ensuring that only verified, well-formed transactions run against the production database, and enforcing a separation-of-duty policy between software development and deployment. The Chinese Wall security model [BN89] is focused on enforcing separation of duty policies. These policies are inherently history based

(working with company X prevents one from working with company Y in the future).

Anderson has worked out a realistic medical information privacy policy. [And96] However, his work is designed for a single-payer medical system, as exists in the UK. Our model more closely follows the American system, with doctors receiving payment from multiple insurance companies.

We unabashedly borrow from these frameworks. In our example, we model each patient as a separate compartment (in MLS terms), with the associated privileges and privacy rights somewhat corresponding to levels. However, we allow for (countably) infinitely many compartments, which may be created dynamically, as in the Chinese Wall policy. Our concept of verified programs essentially corresponds to well-formed transactions in the Clark-Wilson model. These programs, in a generalization of the MLS "trusted subject" notion, may place arbitrary security labels on their outputs. While every label is a lattice element, similar to all practical MLS systems, we allow seeming downward flows: for example, the verified program may have suitably anonymized or encrypted the data.

Comparison with Floating Level MLS Systems The Trusted Solaris system [Suna] also associates labels with files and processes, has an option for "information label floating", which propagates labels along the path of information flow. There are three types of labels in Trusted Solaris: sensitivity label, information label, and CMW label (which is simply a combination of sensitivity label and information label). The labels there, however, are used to implement multi-level security systems in a dynamic fashion. The label combination function is a simple MLS "write-up" rule.

The manual for Trusted Solaris [Sunb] did note the problem of cross-invocation contamination. The proposed solutions are changing applications, asking the user to manually reset the labels, or allowing the application to stop label propagation all together. None of these are desirable. In contrast, the technique of white-listing helper files avoids contamination while making sure that labels still propagate through the applications.

Comparison with Encrypted Storage Encryption is another means of access control. There are two ways to encrypt storage: encrypting disks and encrypting files. Encrypting disks (and tapes) mostly improves the physical security of the storage medium, and is completely complementary to the scheme described here. Encrypting files offers a method of access control that does not weaken as the information is copied or stored into other files. However, it is difficult to design an encryption framework that accommodate the complexity and richness of privacy protection, where both internal and external recipients are involved.

7 Conclusions and Open Issues

We present a file system that prevents accidental information leakage. The Hippocratic file system attempts to balance security and usability. With modest effort, it can be circumvented, but it will force the perpetrator into the position of knowingly doing wrong. We describe a prototype implementation and experience with existing applications.

Two main open issues remain.

Backup The system backup procedure must be able to read all files during backup and assign labels during restore. One solution is to use Domain and Type Enforcement assured pipelines, so that the backup must be encrypted for confidentiality and signed for integrity before being written to removable media. The restore program must be trusted to verify the signature and properly label all files it restores.

Time limitation One of the other OECD principles requires keeping personal data for only a limited time period. In the presence of aggregate data and backups, this can be difficult to enforce. While the challenge with backups can be addressed with encryption techniques, more effort is needed to develop suitable policies for time-limits on aggregated data.

We plan to address these in our future work.

References

[AKSX02] Rakesh Agrawal, Jerry Kiernan, Ramakrishman Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the 28th VLDB Conference*, 2002.

[And96] Ross Anderson. A security policy model for clinical information systems. In *Proceedings of the 15th IEEE Symposium on Security and Privacy*, pages 30–43, Oakland, CA, May 1996.

[Bib75] K. J. Biba. Integrity considerations for secure computer systems. Technical Report

MTR 3153, Mitre Corporation, Bedford, MA, June 1975.

[BK85] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

[BL76] D. Elliot Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997 Rev. 1, MITRE Corporation, March 1976.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[BSS+95] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*, pages 321–336, San Diego, CA, August 2004.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security poli cies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[HIP] HIPAAps Inc. Hipaa privacy & security: Examples of privacy violations. `http://www.hipaaps.com/main/examples.html`.

[KEF+05] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop of Hot Topics in Operating Systems*, June 2005.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, 2001. `http://www.nsa.gov/selinux/papers/freenix01-abs.cfm`.

[Nie] D. C. Niemi. Unixbench 4.1.0. `http://www.tux.org/pub/tux/niemi/unixbench`.

[Ols] Stefanie Olsen. Top web sites compromise consumer privacy. December 17, 1999, `http://news.com.com/2100-1017-234631.html`.

[Ope] Open Source Community. Linux security modules. `http://lsm.immunix.org/`.

[Org] Organisation for Economic Co-operation and Development. Oecd guidelines on the protection of privacy and transborder flows of personal data. `http://www.oecd.org/document/18/0,2340,en_2649_201185_1815186_1_1_1_1,0%0.html`.

[SFV] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux security moduels: General security hooks for linux. `http://lsm.immunix.org/docs/overview/linuxsecuritymodule.html`.

[Sta66] Heinrich Von Staden. In a pure and holy way: Personal and professional conduct in the Hippocratic oath. In *Journal of the History of Medicine and Applied Sciences*, volume 51, pages 406–408, 1966.

[Suna] Sun Microsystems, Inc. Trusted solaris 2.5.1 answer book. `http://docs.sun.com/app/docs/coll/175.2`.

[Sunb] Sun Microsystems, Inc. Trusted solaris 2.5.1 answer book: Label guidelines. `http://docs.sun.com/app/docs/doc/805-8031/6j7i5o2fm?a=view`.

[Wee01] Stephen Weeks. Understanding trust management systems. In *Proceedings of the 15th IEEE Symposium on Security and Privacy*, pages 94–105, Oakland, CA, May 2001.