# Practical Data-Leak Prevention for Legacy Applications in Enterprise Networks

Yogesh Mundada, Anirudh Ramachandran, Mukarram Bin Tariq, and Nick Feamster
School of Computer Science, Georgia Institute of Technology
{yhm,avr,mtariq,feamster}@cc.gatech.edu

## ABSTRACT

Organizations must control where private information spreads; this problem is referred to in the industry as *data leak prevention*. Commercial solutions for DLP are based on scanning content; these impose high overhead and are easily evaded. Research solutions for this problem, *information flow control*, require rewriting applications or running a custom operating system, which makes these approaches difficult to deploy. They also typically enforce information flow control on a single host, *not* across a network, making it difficult to implement an information flow control policy for a network of machines. This paper presents *Pedigree*, which enforces information flow control *across a network* for legacy applications. Pedigree allows enterprise administrators and users to associate a label with each file and process; a small, trusted module on the host uses these labels to determine whether two processes on the same host can communicate. When a process attempts to communicate across the network, Pedigree tracks these information flows and enforces information flow control either at end-hosts or at a network switch. Pedigree allows users and operators to specify *network-wide* information flow policies rather than having to specify and implement policies for each host. Enforcing information flow policies in the network allows Pedigree to operate in networks with heterogeneous devices and operating systems. We present the design and implementation of Pedigree, show that it can prevent data leaks, and investigate its feasibility and usability in common environments.

## 1. Introduction

Enterprise networks continually face the threats of data "leaks", where sensitive information travels to some part of the network where it should not have gone. The insider-initiated leak at Goldman Sachs in July 2009 [11] and the leaks of classified military videos [46] are recent prominent examples, but such mishaps are all-too frequent. Verizon's recent data breach investigation report found that 43.2% of data breaches occurred through deliberate actions of insiders and 38% were due to malware [45]. Ponemon Research estimates the average total cost of each data breach incident to be $6.75 million in 2009 in the United States [32]. Existing commercial data leak prevention (DLP) systems focus on regulatory compliance and are not equipped to defend against malware and insider threats. For example, an attacker may be able to evade these DLP systems simply by encrypting the sensitive document. Our challenge is to design a breach-prevention system that provides more comprehensive control of information flow within and outside the enterprise, *without requiring the deployment of new applications or operating systems*.

*Information flow control*—controlling information exchange between principals according to some security policy—is a well-established field: "lattices" were introduced nearly 40 years ago [7], and many systems have attempted to implement some type of support for information flow in programming languages, applications, or even the operating system itself. Despite a long history and deep theory, however, today's systems for implementing information flow control fail to stop even simple data leak scenarios, not because they lack sound theory or implementation, but rather because they are too difficult to use and deploy. Existing information flow control systems often require an application to be rewritten [19], often in a specialized programming language [26]; sometimes, these systems require deploying an entirely new operating system [44, 50]. These systems are useful for new applications or controlled settings, but, unfortunately, not every application people need (or want) to use can be rewritten, and, in reality, users may run their own software on a heterogeneous set of devices that run a variety of operating systems. Commercial information-flow control products must resort to enforcing information-flow control policies by inspecting the contents of network traffic [24, 36, 41], but many of these systems can actually be circumvented with encryption or other obfuscation techniques.

We present Pedigree, a practical information-flow control system for enterprise networks that applies to legacy applications and operating systems. In designing Pedigree, we recognize that information-flow control has two functions—propagating labels and enforcing policy. Previous systems have not been able to decouple these two functions, resulting in systems that are somewhat difficult to deploy or manage in practice. In contrast, Pedigree decouples how and where these

two functions are implemented: a small, trusted labeler propagates and manages labels on each host, and a central controller manages policy and enforces information flow. Pedigree separates the "data plane" (the functions that forward data and associated labels) from the "control plane" (the elements that decide whether the information should be forwarded in the first place). Network designers have recognized the benefits of this decoupling for simplifying various network management tasks [29]. Pedigree leverages these emerging functions and technologies for separating the network data plane and control plane, making it much easier to build an information-flow control system that both operates across the network (rather than only on a single host) and allows users to specify and control policy independently of how it is enforced. Recent developments in this area, such as the realization of the OpenFlow specification in commodity switches, have made it possible to implement designs that enforce this separation and deploy them on production networks.

Pedigree has many advantages over existing information-flow control systems. First, it works for *unmodified legacy applications in heterogeneous environments*. To our knowledge, Pedigree is the first lattice-based information-flow control system that can work with legacy applications (*i.e.*, without requiring them to be rewritten or recompiled).

Second, Pedigree performs information-flow control across a *network* of end-hosts running commodity operating systems and unmodified applications; earlier research on networked information-flow control requires all hosts to run a specialized operating system [51]. Finally, it is *simple and manageable*. Having policy at a central controller allows *users and operators* to specify information flow policies for their own data and applications that can then be centrally managed, instead of having applications themselves manage policy as in other information-flow control systems [19, 50].

Pedigree's design goals present several challenges. First, centralized policy and label management implies that hosts must continually communicate with a central controller, which poses scalability concerns. Second, instrumenting legacy applications and operating systems implies that label propagation interposes on system calls; this approach not only imposes system-call overhead, but it also means that entire processes, rather than individual variables or regions of memory, acquire security labels. Practically speaking, an application may become unusable if it acquires a high security label (*e.g.*, if it reads a sensitive file), making it unable to communicate with any other resource. We evaluate scalability and overhead of Pedigree in Section 7 and show that it is acceptable for practical usage scenarios.

Looking forward, we believe that Pedigree's approach to data-leak prevention allows centralized policy specification with distributed enforcement across heterogeneous network devices can lead to practical information-flow control systems in many emerging settings. In this paper, we focus on Pedigree's applicability for data-leak prevention in an enterprise network setting, but a similar approach might also help control data propagation in other settings with heterogeneous software and applications that the infrastructure administrator cannot fully control (*e.g.*, "cloud" infrastructure).

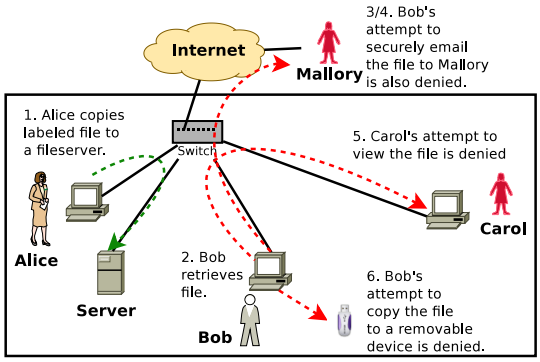



**Figure 1: Example data-leak scenario. Bob can attempt to leak data in a number of ways, including copying the file over the network (either inside or outside the network), or copying the file to a removable drive. Data-leak prevention software should prevent leaks in both cases.**

The rest of the paper proceeds as follows. Section 2 describes an motivating data-leak scenario that we will appeal to throughout the paper. Section 3 describes Pedigree's design, including a comparison to existing systems. Section 4 explains how Pedigree implements information flow control policies both on a single host and across a network. Section 5 describes the details of label management. Section 6 describes our implementation of Pedigree, and Section 7 evaluates Pedigree's overhead and usability. Section 8 explains the extent to which Pedigree can defend against a range of attacks. Section 9 compares Pedigree to related work, and Section 10 concludes.

## 2. Example and State of the Art

Figure 1 shows a concrete use case that we will use to discuss the capabilities of Pedigree in comparison to existing systems. Suppose that Alice wishes to share a secret idea with her colleague, Bob, who is authorized to see it. Alice copies the file to the company's network file server (Step 1), which Bob then directly accesses (Step 2). At this point, secret data can traverse the network freely, since both users are allowed to view the secret data.

Now suppose that Bob wishes to send the secret to an outsider, Mallory, against company policy. Bob's first attempt is to attach the file to a message in his email client and send it; a network device recognizes that the traffic contains secret data, but this time prevents it from leaving the network. Frustrated, Bob then tries to down-

load the file to his local machine and send it as an encrypted email attachment. Again, the network switch recognizes that the email message contains secret data and thwarts his attempt (Steps 3 and 4). Bob then tries to send the file to his colleague Carol to see if she can help him transport the file out of the network (Step 5); however, Carol is not authorized to see the file, so a trusted module on Carol's host blocks the transfer. As a last resort, Bob tries to copy the encrypted version of the secret file off his machine using an auxiliary device, such as a USB drive or over Bluetooth (Step 6); at this point, the trusted module on Bob's machine recognizes that Bob does not have the privilege and prevents the transfer from happening.

Table 1 compares Pedigree to existing information flow control systems. *Granularity* refers to the granularity at which the system tracks information flow. Pedigree tracks information flow at the granularity of files and processes, which is coarse relative to TaintTracker [27], Panorama [49], and the recent mobile OS taint-tracking system, TaintDroid [9], which track information flow at byte-level granularity (albeit within an emulator or a virtual machine). RSA DLP [36] attempts to control information by inspecting the contents of a resource or message; this approach can be evaded using encryption (*e.g.*, Bob would be able to send his file outside of the enterprise network by first encrypting it). Like Pedigree, both Flume [19] and Dstar [51] track resources at the granularity of processes, but neither Flume nor Dstar work with unmodified applications. TaintEraser [53] allows users to track sensitive input (*e.g.*, passwords entered using the keyboard), but it is limited to a single host, and its taints are not as expressive as with Pedigree; it does not allow users to define arbitrary labels of their own (*e.g.*, Alice could not create a "secret" label for a file that she creates). Because Pedigree can run on a commodity operating system, is more vulnerable to covert channels than Dstar; we discuss these vulnerabilities in Section 8.

# 3. Pedigree Design

This section presents the design of Pedigree, after briefly discussing the threat model.

**Threat Model.** We make some assumptions regarding the enterprise and user behavior. We assume, as in other threat models [1, 38], that the typical enterprise user will not take extreme measures to subvert Pedigree: specifically, we assume that the user will not attempt to remove internal storage media or bypass the operating system to read the physical storage (if such attacks are feasible, we assume that techniques such as full-disk encryption can be applied in defense). Pedigree also runs a trusted module—the labeler—in the OS kernel on end-hosts in the enterprise; we assume that this module cannot be compromised by enterprise users or malicious programs. Finally, we assume that routers and switches in the enterprise forward traffic with new IP options (*e.g.*, for Pedigree-related protocol messages). We discuss potential attacks against Pedigree in the context of this threat model in Section 8.

## 3.1 Overview

Pedigree tracks and controls the propagation of *labels*, associated with every *resource*, which may be a file or a process. Either a user or an administrator can specify an initial label for a resource. Pedigree has two components: a labeler and an enforcer. The *labeler* that resides on each host manages, tracks, and updates labels as resources interact with one another. Based on these labels, an *enforcer* decides whether a particular information flow between two resources can take place. We first provide an overview of these functions. We then describe how Pedigree's labels help enforce information flow control ("IFC") using our running example.

**Label management and maintenance.** Every resource (*i.e.*, process or file) has a label, and the user who owns the resource can modify a label to specify information-flow policy. *Labelers* reside on end-hosts and mediate all interactions between resources in the enterprise, whether within a single host or between hosts. If information flows from a resource $p$ to a resource $q$ on a single host (*e.g.*, a process writes to a file), the labeler on that host retrieves the labels for both $p$ and $q$ before initiating IFC checks. If $p$ and $q$ are on different hosts in the enterprise (*e.g.*, a process sends data to a remote server), the remote host's labeler initiates the IFC check: only the remote host can determine which process accepts data sent by $p$ and hence its label. Because the remote host also needs $p$'s label to perform checks, the labeler on $p$'s machine pushes $p$'s current label to a central location that the remote labeler can access; we call this central repository the *global label store*.

**Policy enforcement.** *Enforcers* perform information-flow checks; they may be located in the network ("network enforcer"), or as part of the Pedigree module on host OSes ("host enforcer"). Host enforcers prevent information leaks that occur at a host, for example, a malicious process attempts to read a confidential file, or it tries to write the contents of a confidential file to a removable drive. For each information flow where the labels of the sender resource $p$ and receiver $q$ differ, the labeler queries the host enforcer to decide whether an information flow is permitted. Network enforcers control the propagation of information based only on the sender's label and network flow attributes. For example, the network enforcer could prevent traffic flows that may contain secret information from reaching insecure networks (*e.g.*, an open wireless network, or the outside

| System | Granularity | Legacy Apps | Execution Environment | Expressive Labels | Network-wide | User-role | Covert Channels |
|---|---|---|---|---|---|---|---|
| RSA DLP [36] | Coarse[1] | ✓ | in commodity OS | × | ✓ | none | many |
| Panorama [49] | Fine | ✓ | within an emulator | × | × | none | some |
| TaintEraser [49] | Fine | ✓ | in commodity OS | × | × | some control | few |
| Flume [19] | Coarse | × | in commodity OS | ✓ | × | none | few |
| Dstar [51] | Coarse | × | in new OS | ✓ | ✓[2] | none | very few |
| **Pedigree** | Coarse | ✓ | in commodity OS | ✓ | ✓ | direct control | few |

**Table 1: Comparison of related information-flow control systems. Pedigree can enforce information-flow control for legacy applications that run on a commodity operating system. It also enforced information-flow control across the network (rather than on a single host), and allows individual users to define security classes (rather than only a network or system administrator.**

Internet).

**Challenges.** Pedigree's design entails several challenges. The first challenge concerns *how principals can manage and manipulate labels*. Earlier approaches [19, 50, 51] expect processes to manage their own labels and associated policies, which requires rewriting applications. Because Pedigree is designed to work with legacy applications on commodity OSes, we separate policy management (*i.e.*, who is allowed to read what) and from propagation and maintenance of labels. Pedigree's principals are enterprise users; Pedigree delegates management of policies to users. The actual tracking of labels as information flows between resources is performed automatically by in-kernel labelers, as is policy enforcement when an information flow violation occurs A second challenge entails addressing the trade-off between centralized and decentralized policy management: decentralized policy (*e.g.*, capabilities with Flume [19] or "clearance" in messages with DStar [51]) has the advantage of limited compromise, but requires individual hosts to manage their own policy. Pedigree has a centralized policy management infrastructure that is easier to deploy, maintain, and audit, because most enterprises are already used to enterprise-wide centralized services (*e.g.*, email servers, authentication portals).

## 3.2 Labels

Pedigree associates *labels* with processes and files (specifically, inodes). A label is a set of *taints*, each of which is a 64-bit integer. The first two bits of a taint denote the *secrecy* and *integrity* status of the resource with respect to that taint. If secrecy or integrity bits are set for one or more taints in a label, the labeled resource will require information flow checks when communicating with other resources. If a resource's label acquires one or more taints with the secrecy bit set, we say that the resource's secrecy has been "raised"; similarly, removal of one or more secret taints is referred to as "lowering secrecy" or "declassification". We will use $S_p$ to denote the set of taints in the label of a resource $p$ for which the secrecy bit is set, and $I_p$ to denote the set of taints that have the integrity bit set. According to the access control model proposed by the Bell-LaPadula model [4]—which most information-flow control systems adopt [19, 50]—information can flow from resource $p$ to resource $q$ if

$$S_p \subseteq S_q \text{ (\textit{i.e.}, \textit{no read ``up'', no write ``down''})} \quad (1)$$

$$I_q \subseteq I_p \text{ (\textit{i.e.}, \textit{no read ``down'', no write ``up''})} \quad (2)$$

The $\subseteq$ relation represents "can flow to", *i.e.*, $S_p \subseteq S_q \Rightarrow p$ can write to $q$. Labels form an information flow lattice under the partial order of the $\subseteq$ relation [7]. Pedigree also follows these rules for information-flow control, with exceptions as described in Section 4. Because the relations imply that data can only flow towards higher secrecy levels, processes may ultimately be unable to export secret data without a mechanism to declassify it. Thus, Pedigree also allows users and their processes to declassify information if they have the corresponding capabilities; we briefly explain label management in the next section. Section 5 describes label management processes in more detail.

## 3.3 Label Management

Enterprise *users* assume the responsibility of creating taints, and initially adding taints to the label of a resource they own. Users can create new taints or modify the labels of processes and files based on the user's *capabilities* with respect to each taint. We introduce three components to enable information flow tracking and control within the enterprise: an *authentication service*, the *label stores*, and a policy database called the *capability database*.

Before a user can create taints or modify labels, he must be authenticated. The authentication service (*authserv*) is centrally managed by a trusted enterprise administrator. Once authenticated, the authserv allows users to (1) create new taints; (2) manage a taint that they own; and (3) modify labels of a resource that they own by adding or removing taints from the label.

The *label store* is a repository that stores the labels associated with resources. Each host has a local label store—usually an encrypted partition accessible only to the host's in-kernel labeler—that stores the labels of resources on the host. In addition, if a process with a non-null label communicates over the network, its host labeler pushes the process's label to the centrally-

managed global label store to enable network-wide information flow control. The global label store exposes the labels for networked applications to network enforcers, as well as any host enforcer that might want to enforce an IFC policy at to the receiving host. The sending host's labeler ensures that, before a process is allowed to send data to the network, its latest label is written back to the global label store.

The *capability database* manages the capabilities that users have for manipulating taints. Each record in the capability database pertains to one taint, and is stored as a list of (user, capability) tuples. Each user can have six capabilities for manipulating a taint:

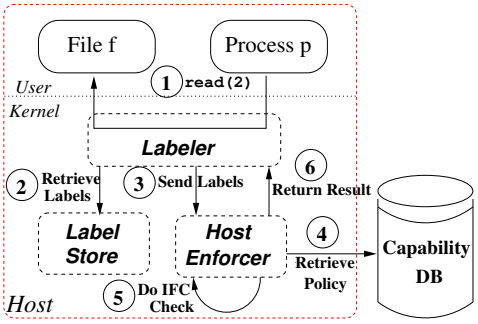
- set the *secrecy* bit ($s^+$), or unset it ($s^-$)
- set the *integrity* bit ($i^+$), or unset it ($i^-$)
- add/remove users who can manage capabilities of the taint
  (*i.e.*, the "capability set") ($o^+$ and $o^-$).

A user who first creates a taint has all six capabilities. Taints may also have an entry with the username set to a "wildcard" entry, to specify capabilities for users not explicitly listed in the capability set; this wildcard capability is how Pedigree implements provenance, which helps prevent confidential enterprise-wide information (*e.g.*, projected earnings, internal memos) from being leaked. For example, the user that creates a file may set a wildcard entry with only $s^+$ rights; this implies that while any other user will be able to open and read the file using their reader application, the application will not be able to *remove* the taint, essentially restricting the file to dissemination only within the enterprise network boundary.

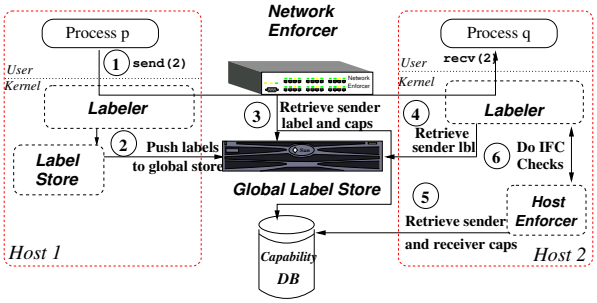
In this paper, we focus on examples using secrecy bits alone, because these are most useful in a data loss prevention scenario. Integrity bits, however, can be used as illustrated in previous work on information-flow control, such as for preventing untrusted processes or applications from editing critical system files and directories (*e.g.*, `/etc/rc` [19]).

## 3.4 Policy Enforcement

Figure 2(a) shows how Pedigree enforces information flow control on a single host. Suppose $p$ is a process that wishes to read a secret file $f$; the user that owns $f$ can mark $f$ as secret to all other users. When $p$ attempts to read $f$ (Step 1), the labeler retrieves the labels for $f$ and $p$ from the local label store (Step 2). The local label store is an in-memory structure that is buffered on disk and maintains labels of all active and persistent resources, such as running processes and files. The local label store is also stored persistently on disk. The labeler then queries the host enforcer with the labels of $p$ and $f$ to check if the information flow from $f$ to $p$ is permissible (Step 3). The enforcer first retrieves the capabilities




(a) Information flow control *within a host*.




(b)

Information flow *between two hosts*.

**Figure 2: Pedigree can control information flow either between processes on a single host, or between processes across a network.**

of the users that own $p$ and $f$ for *all* taints in the labels of $p$ and $f$ (Step 4) from the capability database, and then checks whether $p$ reading $f$ would cause a violation of Equations (1) or (2) (Step 5). The enforcer returns the result of the check in Step 6.

Figure 2(b) shows how Pedigree enforces information flow control policies across a network. Suppose the malicious process $p$ on Host 1 is attempting to exploit a trusted server process $q$ on Host 2. Because $p$'s data is sent over the network, the labeler on $p$'s machine pushes $p$'s latest label to the global label store at a location accessible to the network enforcer and the labeler on Host 2 (Step 2). The network enforcer, if it wishes, can check whether the attributes of the flow and $p$'s labels permit the send to take place (Step 3). If the network enforcer allows the flow to reach Host 2, the labeler on Host 2 first retrieves $p$'s label from the global label store (Step 4). After this point, the information flow is equivalent to a *local* inter-process communication on Host 2, and Host 2's labeler subsequently invokes its enforcer to perform the IFC check using $p$'s and $q$'s labels, which the enforcer completes after retrieving appropriate capabilities from the capability database (Steps 5 and 6).

The *authserv* (not shown in Figure 2) allows users to: (1) create and add a new taint to a file or process with the secrecy or integrity bit set; (2) change the capabilities of users on any given taint; or (3) remove a taint

from a file's or process's label. Section 5 details these operations.

## 4. Pedigree in Practice

We explain how Pedigree provides information flow control both on a single host and across the network.

### 4.1 Single-Host Information Flow Control

Enforcing information flow policies at a host involves propagating labels for resources using the labeler, and enforcing information flow using the host enforcer. To track information flow between resources, the host labeler intercepts system calls where information flows between resources (*e.g.*, `read(2)`, `send(2)`, `execve(2)`, etc.). After intercepting a system call, the labeler first retrieves labels of the two resources, say $A$ and $B$, from the label store. It then queries the host enforcer, supplying the two labels and the user IDs of the owners of $A$ and $B$ as arguments. The host enforcer retrieves the capability sets of all taints in each resource's label from the capability database, and checks to see whether the information flow from $A$ to $B$ is allowed.

To reduce user intervention and to allow new processes to read secret files, Pedigree slightly modifies the conventional rules for information flow control. Specifically, the labeler tries to automatically allow information flows that only *raise* a resource's secrecy (or integrity); lowering secrecy (*i.e.*, declassification), however, must always require user intervention. For an information flow from resource $A$ to $B$, the labeler tries the following options: (1) setting $S_B$ to $S_B \cup S_A$, *i.e.*, automatically "raising" the secrecy of $B$ to match $A$'s secrecy label; (2) setting $I_A$ to $I_A \cup (I_B - I_A)$, *i.e.*, automatically raising the integrity of $A$. The process's secrecy label is raised to the file's own set, *provided the user who owns the process has the $s^+$ capability* with respect to every secrecy taint in the file's label. Without this modification to IFC rules, a user would have to explicitly raise the secrecy of her editor application before reading a sensitive document.

If the above conditions are not satisfied, the enforcer denies the information flow, returning the decision "not allowed" to the labeler, which the labeler then returns to the process through an appropriate return code. This automated re-labeling function represents a significant departure from existing DIFC systems, which do not face the same problem because they typically apply to only a few specific processes with a limited set of capabilities.

**Declassification.** Resources may automatically acquire many secrecy taints and be unable to communicate to less-secret resources or the outside network. To allow users with appropriate capabilities to "declassify" their processes and files, the labeler exposes the `declassify` system call, which sets a bit in the process's label indicating that the process *can have* a lower secrecy level for subsequent resource interactions. For example, consider a `write(2)` call by process A on file B invoked after first invoking `declassify`. Instead of the default behavior of raising $S_B$ to $S_B \cup S_A$, the labeler would *lower* $S_A$ to $S_A - (S_A - S_B)$, (*i.e.*, secrecy taints in A that are not present in B are declassified). All information flow from the process will be performed under declassify until the bit is explicitly unset with another syscall.

**Preventing Host-based Information Leaks.** The labeler marks all potential avenues for information leaks, including network interfaces, secondary or removable hard drives, and other communication mechanisms such as Bluetooth, Infrared, or Firewire, as "dummy" resources with *immutable* taint sets $S = \phi$ and $I = \phi$. When a process or user that does not have $s^-$ for one of its taints tries to write to a removable drive, the host enforcer prevents the action. Exceptions to this rule are the primary network interface (*i.e.*, the network interface connected to the network enforcer and the rest of the enterprise) and the primary hard disk (*i.e.*, on which the operating system is installed); for writes to these devices, the labeler does not perform the usual IFC checks. Our threat model assumes that the primary hard disk is not an avenue for leaks. Data sent to the network over the primary interface is checked by a host enforcer on the remote host or by the network enforcer in the network. The labeler discovers the primary hard disk at OS install time; it discovers the primary network interface at boot time using a two-way bootstrap protocol between the labeler and a *labeler authentication service*, which is described in Section 5.1.

At boot time, the host enforcer builds a list of all output devices (except the display device) as potential avenues for information leaks, and sets the immutable secrecy taint sets for these devices as $\phi$ to prevent tainted resources from writing to these devices. It excludes from this list the primary NIC and the primary hard disk; these devices receive a special "master" taint, *i.e.*, $S_{NIC} = \{\text{master}\}$. The master taint indicates that `send(2)` and `recv(2)` through this interface do not involve IFC checks, nor do they change the taint sets of the calling process, irrespective of a process's secrecy set $S$.

**Example.** Suppose Alice writes a confidential report $f$ on a multi-user machine. She wants to get feedback on the report but does not want other users to copy it off the machine. After creating the document, she authenticates herself to the authserv, creates a new taint $t$, sets the secrecy bit, and applies it to $f$'s label. In addition, she changes the taint's capability set to include a wildcard entry that gives any user the capability $s^+$ on the taint (we call this a *provenance* taint).

When Bob then uses his untainted editor process $e$ to open $f$, his host's labeler sees that $S_e(= \phi) \not\subseteq S_f$.

It first checks whether the capabilities of all taints in $S_f - S_e (= \{t\})$ allow the capability $s^+$ for Bob. Because Alice added an entry with $s^+$ for any user on taint $t$, the host enforcer allows Bob's editor to read $f$ after raising $S_e$ to $\{t\}$. Bob's editor may fork other programs and write copies of files (all of which retain the taint $t$), but because he does not have $s^-$, he cannot export any portion of the file through a removable drive or a secondary network device. After receiving comments from Bob, if Alice wishes to take a copy of the file home using a removable drive, a command such as "`cp f /usbdrive`", will fail because the secrecy taint set of the USB drive is empty and immutable. However, if she executes "`declassify cp f /usbdrive`", the labeler first sets the declassify bit for the `cp` process, and subsequently drops $S_{cp}$ to $S_{usb}$ because Alice possesses $s^-$ for all taints in $S_{cp} - S_{usb}$, *i.e.*, $\{t\}$.

## 4.2 Network-Wide Information Flow Control

The challenge in enforcing IFC at the network or a remote host is that the receiver's enforcer needs to make IFC decisions but it does not have the labels associated with the sending process. To uniquely associate a sender process with a label, the sender's labeler annotates each packet with a resource ID and a version number. The network enforcer or the enforcer on the receiving host can retrieve the sending process's label using the sender's IP address, resource ID, and version number from the global label store. The resource ID is unique during a process's lifetime; we use the process ID as the resource ID. Version numbers increment from zero and indicate the version of the sending process's label; if the sending process's label changes, the sender's labeler annotates subsequent packets with the incremented version number to indicate to the receiving labeler that the sender's label has changed.

**IFC checks at the Remote Host.** The remote labeler first extracts the resource ID and version number from the packet header for incoming flows and retrieves the sender's label from the global label store. Because it also knows the label for the *receiving* process, it can perform IFC checks using the remote host enforcer, similar to the single-host IFC scenario. Specifically, operations such as `declassify` can also work across the network: if a user invokes a `send(2)` call in a declassified process $p$, the receiving process $q$ will be able to read the data if the sending process's owner possesses $s^-$ for all taints in $S_p - S_q$. Section 5 sketches a protocol that allows the sending process to be declassified after the enforcer at the remote host verifies that the declassification would successfully allow information flow.

**IFC checks at the Network Enforcer.** The network enforcer typically resides at boundaries between networks of different trust levels, such as at the edge of the enterprise leading to the Internet, or between wired and wireless network boundaries within the enterprise. The enforcer has policies that designate immutable secrecy and integrity taint sets to certain destination prefixes, or perhaps to specific ports of the network device. For example, the enforcer can designate traffic destined to the Internet as $S_{\text{Internet}} = \phi$. When a network enforcer sees a new data flow, it extracts the resource ID and version number from the packet header and retrieves the sender's label from the global label store. It then performs IFC checks to ensure that the data flow is permitted. For example, the policy of setting $S_{\text{Internet}} = \phi$ would indicate that an outgoing flow generated by a process with any secrecy taints at all will be dropped. Once a flow passes IFC checks, the network enforcer typically installs a rule that allows future packets with the same resource ID and version number through without undergoing any checking. This functionality is provided by the popular Openflow switch platform [29] in combination with a NOX controller.

**Example.** Let us revisit the previous example in a networked setting (Figure 3). Alice creates her confidential file $f$, applies the taint $t$, and deposits the file on a publicly accessible directory on the enterprise file server. The labeler on the fileserver ensures that the label of the file contains $S = \{t\}$.[3] Now suppose Alice wishes to collaborate with Bob in editing the file, but wishes to keep the file secret from everyone else. Using the authentication service and the capability database, she modifies $t$'s capability set to include Bob's user ID with the associated capabilities $\{s^+\}$, as shown below.

| *username* | *capability* |
|---|---|
| `alice` | $s^+, s^-, i^+, i^-, o^+, o^-$ |
| `bob` | $s^+$ |

Now, Alice and Bob can read the file within the enterprise, but Bob cannot export the file's data, and a malicious user, Carol, cannot read the file, since $t$'s capability set for $\{s^+\}$ does not contain Carol.

Now suppose that after some collaboration, Bob forgets that the file is secret and tries to send it to his friend Mallory using encrypted email through his machine's primary network interface card. His email client acquires the taint $\{t\}$ as soon as it reads the file. Although no IFC checks are performed on his host, the network enforcer, after comparing the sending process's label ($\{t\}$) with $S_{\text{Internet}} = \phi$ discovers the information flow violation and blocks the traffic; the same violation occurs irrespective of the method Bob uses for exfiltration over the primary interface. Frustrated, Bob tries to

[3]We assume that the fileserver is either an untainted process that forks new processes to service each incoming connection (*e.g.*, sshfs), or a kernel-resident server (*e.g.*, NFS). This assumption is needed because the label of a single-process fileserver may accumulate a large number of taints as different processes write to it such that IFC rules are violated.

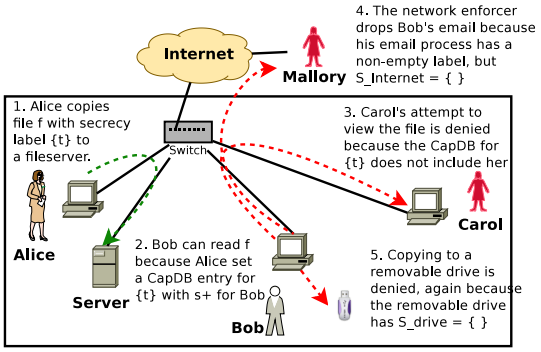



**Figure 3: Example data-leak scenario, with explanations of the IFC violations that prevent the data leak attempt.**

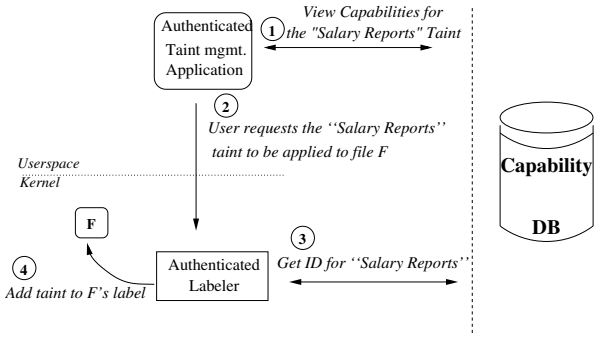



**Figure 4: Steps involved in applying a taint to a resource.**

run `declassify` before attempting to leak the file. Although the email client's process will then have its declassify bit set, it does not have any effect because Bob does not have the $s^-$ capability for the taint $t$.

Pedigree can also protect against leaks of data by malware in the same fashion: if the malware is running as Bob's user, it is limited by the same capabilities as Bob. Even if the malware is running on Alice's machine *as Alice*, Pedigree can prevent it from leaking $f$: unless a process performs the data leak under `declassify`, the taint $t$ will stay in the process's label, preventing it from leaving the enterprise boundary. Thus, the malware has to not only run as Alice, but also have Alice invoke `declassify` on it. Because we expect that `declassify` will be implemented similar to a `sudo` operation, requiring a password (or a proof-of-human test such as a CAPTCHA), a malicious program will not be able to stealthily leak $f$ without Alice's explicit approval.

## 5. Secure Label Management

This section describes secure protocols for associating the labeler with a host (Section 5.1) and allowing users to manage the capabilities of taints for resources that they own (Section 5.2).

### 5.1 Associating a Labeler with a Host

Although the global label store may contain labels from various hosts, a particular labeler must be able to manipulate labels in the global label store only for resources that are associated with its host. Labelers must be uniquely identifiable so that Pedigree can associate each labeler with some host, even as hosts enter and leave the network or change IP addresses. Thus, an administrator assigns a unique private/public key pair to each labeler when the OS is installed. The operating system and labeler also have a unique *Host ID*, which is usually its public key (or a hash of the public key); the Host ID can thus be used for self-certification.

When a host enters the network, its labeler registers with a *labeler authentication service*. The labeler authentication service knows each Host ID and corresponding public key; this public key is used to prove the labeler's identity and also to securely establish an expirable session key for use between the labeler and the enterprise services (label store, capability database, and the user authentication service, *authserv*). The labeler also interacts securely with the global label store in order to register storage for itself, and to push and retrieve labels for network communication.

In some cases, a host may have multiple network interfaces, with only one is connected to the enterprise network (the "primary" interface). The labeler must determine the primary interface so that it can denote all other interfaces as potential avenues for data leaks (*i.e.*, assign them immutable taint sets of $S = \phi$). To discover the primary interface, the host labeler broadcasts the message to all configured interfaces, and designates the interface on which it receives a signed response from the labeler authentication service as the primary NIC.

### 5.2 Allowing Users to Manipulate Labels

To allow users to create and manipulate labels for their files and declassify resources, we provide a userspace library and application. For all operations, the user first authenticates himself to the authserv using his enterprise credentials and potentially a proof-of-human test. Once the user is authenticated, the application contacts the capability database and presents the user a list of operations, including (1) create a taint; (2) manage capabilities for taints that the user has the $o^{\pm}$ capability; (3) add a taint to the label a resource running on his local host, such as a file or a process. The label management application replaces taint IDs with mnemonics (*e.g.*, "Salary Reports"). Users can manage capabilities on taints directly through the application, but to apply a taint to a local resource, the application requires help from the labeler.

Figure 4 shows the sequence of messages exchanged when a user adds a taint he owns to the label of a file on his machine; we require this mediated because we do not wish to give any malicious users or malware direct visibility or control over the labels of their resources. Once the user has chosen a taint to be applied to a spe-

| Syscall Type | Example syscalls |
|---|---|
| Inter-process Communication | send(2), shmat(2), msgsnd(2), kill(2) |
| File/device operations | read(2), unlink(2), mknod(2) |
| Process creation | fork(2), execve(2), clone(2) |
| Memory operations | mmap(2), mprotect(2) |
| Kernel configuration | sysctl(2), init_module |

**Table 2: Types of system calls that are monitored by the labeler, with examples of each system call type.**

cific resource on his host (Step 1), the taint management application invokes a system call to its local enforcer, providing as arguments the user ID, the type of request that the user wishes to perform, and any extra arguments necessary to complete the request. For example, to add a secret taint with the mnemonic "Salary Reports" to a local file, the system call will be a `TAINT_MANAGE` request that includes the mnemonic of the taint, and the full path to the local file (Step 2). The labeler communicates with the capability database using its own encrypted channel to ensure that the user is authorized to perform the requested operation. If so, the labeler receives the actual taint ID for "Salary Reports" (Step 3), which it adds to the local file (Step 4). If all steps succeed, a "success" code is returned to the taint management application. Other functions of the taint management application include a `TAINT_CREATE` request to create a new taint (and a name for the taint), and `TAINT_MODIFY`, which modifies the capability sets of a taint.

# 6. Implementation

We describe the implementation of Pedigree's label management services: the labeler, the label store, and the capability database. We then describe the implementation of the host and network enforcers.

## 6.1 Label Management and Maintenance

The labeler is a Linux Security Module (LSM) [48] in Linux kernel version 2.6.22. LSM is a framework within the Linux kernel that allows various security models to be added-on without changing core kernel code. LSM provides hooks within system call handlers that can be implemented by a security module; thus, a third-party module can implement mandatory access control for a system call (*e.g.*, `read(2)`) without changing the core implementation of the system call handler (*e.g.*, `sys_read`). Using LSM hooks, the labeler intercepts all system calls that transfer information between resources on a host. We have implemented hooks to track information flow for the system calls listed in Table 2. The labeler enforces IFC checks using the host enforcer—which, although currently implemented as a kernel module, could also be implemented "below" the kernel (*e.g.*, in a hypervisor or on a trusted platform module). An enterprise administrator first installs the labeler module on the OS when the system is in some known "clean" state

```
struct label {
    /* Version number of this label */
    int version;
    /* Is resource a process or an inode? */
    int is_process;
    /* The identifier for the resource */
    union {
        pid_t    pid;
        unsigned long inode_num;
    };
    /* Taints associated with this label */
    taintset_t *taintset;
    /* Enterprise-wide uid of the owner */
    uid_t uid;
    /* Called under 'declassify'? */
    int can_declassify;
};
```

**Figure 5: Structure of a label.**

(*e.g.*, as might be determined by an audit, virus scanner, or simply by using a Linux distribution with the preloaded labeler for installation). On reboots, the labeler is loaded shortly after `init` during the boot sequence.

The hooks for the system calls are 1,500 lines of C code, and the logic for the labeler and the host enforcer is another 6,500 lines, so the total module implementation is about approximately 8,000 lines of code. We have not yet implemented hooks for certain system calls that transfer information such as semaphore operations, mounting or unmounting of superblocks, and ptrace; we plan to implement these hooks for future work.

**Label Implementation.** The labeler maintains a label for every resource. Figure 5 shows the structure of the label, which has structure for a set of taints (called `taintset`) that is implemented as a sorted set allowing taint addition, deletion, and lookup in $O(log\ n)$ time, and unions in linear time. A taint is described by the taint ID and its current setting of secrecy and integrity bits. The local labeler does not maintain *capabilities* of the user that owns the resource, since those are maintained in the capability database. For efficiency, the labeler does not create or maintain labels for files and directories that have no taints in their label, assigning such resources a "null" label.

**Label Store.** The local label store is a partition that is encrypted using a key embedded in the kernel image, and is stored in a partition not readable to user space processes (enforced using LSM checks). On disk, labels are indexed by the inode numbers or process IDs of the resources to which they map. Label reads and writes are buffered using an in-memory cache of approximately 1,000 entries. At shutdown, labels for files and directories are written to disk; process labels are discarded. To prevent loss of sensitive labels in the event of a machine crash, we use a journaling filesystem on the label store partition, and write back a file's labels to the label store before the file's inodes themselves are written to disk. Pedigree kernel modules implement additional security features to ensure that user-space processes (even

| I/P Port | Src MAC | Dst MAC | Src IP | Dst IP | L4 Src Port | L4 Dst Port | **Pedigree Version** | **Pedigree PID** | O/P Port | Action |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 00:1a:a0:3b:7a:2d | 00:50:56:c0:00:01 | 10.1.1.17 | 10.1.1.37 | 5000 | 443 | **5** | **882** | 12 | *Permit* |

**Table 3: An example of a flow table rule for the modified OpenFlow Switch with two extra fields: label version and process ID.**

superuser processes) cannot interfere with the operation of the labeler, which we describe in more detail in Section 8.

**Capability Database and Global Label Store.** The capability database and global label store are hash tables that allow clients to look up values of keys. We implement both services using Redis [34], a high-performance key-value store. Redis supports about 110,000 `SETs` per second, about 81,000 `GETs` per second [35], outperforming many relational databases (*e.g.*, MySQL) and key-value stores (*e.g.*, memcached). Redis supports only string keys, but values can be of any type. Keys for the capability database are the taint IDs; each value is a structure that contains the name for the taint, and a list of users and their capabilities over the taint.

## 6.2 Policy Enforcement

**Host enforcer.** The host enforcer is a kernel module with the same security and privileges as the labeler. The labeler invokes the host enforcer when it detects information flow between two resources that have incompatible labels. The enforcer communicates with the capability database to retrieve the appropriate capabilities, but it can sometimes make a decision without querying the capability database (*e.g.*, if the sender's label has one or more secrecy taints, but the `can_declassify` bit is not set and the destination is a removable drive, the enforcer can deny the information flow).

**Network enforcer.** The network enforcer uses a slightly-modified OpenFlow [29] switch implementation, and a custom NOX [13] controller that communicates with the switch over a secure channel. The controller queries the capability database and global label store to make information flow decisions and installs rules on the switch to forward or block flows based on the result. We modified OpenFlow switches to augment flow table entries with label version numbers. When a new flow arrives, the switch forwards the traffic to the controller. The network enforcer at the controller performs the checks described in Section 4.2. If the flow is permitted, the controller inserts a flow table entry in the switch as shown in Table 3, and data packets that match this entry are forwarded. If the version number embedded in data packets change mid-flow, the flow table entry no longer matches; at this point, the controller performs a new IFC check on the new version of the sender's label.

# 7. Evaluation

We evaluate the overhead and usability of Pedigree in various scenarios. We used the Emulab [8] testbed for all experiments; our test machines were dual-core Intel Xeon 3 GHz machines with 2 GB of physical memory. We set up a network of hosts connected through a software OpenFlow switch, and verified that Pedigree satisfied information flow policies according to Equation 1. We evaluated the overhead that Pedigree imposes on both local system calls and on network connections. We are running Pedigree on several machines in our lab and have found the implementation to be stable and usable.

## 7.1 Host Overhead

Because the labeler creates a new label for each new resource, Pedigree introduces overhead for workloads that create many resources. We perform two benchmarks to evaluate this overhead: First, we perform a controlled experiment with a series of `read(2)` and `write(2)` to evaluate the individual overheads on the system calls alone with and without Pedigree. Second, we compile several source packages and compare the total times taken with and without Pedigree.

**Microbenchmark: System Call Overhead.** We measure the overhead of Pedigree using two benchmarks: a file read/write benchmark that measures `read(2)` and `write(2)` overhead, and a networking benchmark that measures `send(2)` and `recv(2)` overhead. For the file benchmark, we create 1,000 8 MB files with random data and apply a label to each file. We then use `read(2)` to read the contents of each file and `write(2)` to write the data to a new file. We measure the time for each system call in user space, as well as the primary sources of Pedigree's overhead in the kernel. We average these measurements over the total number of system calls, for all files. To benchmark network socket calls, we set up a TCP client and server, send 10,000 small packets from the client to the server, and measure the average time for a `send(2)` at the client and `recv(2)` at the server.

Table 4 shows the average time taken for each system call, with and without Pedigree. The `read(2)` and `write(2)` system calls incur only minimal overhead: 15% and 6%, respectively. The additional time is spent mostly in fetching the label of a file being read or written from the *local* label store on disk. The write times are smaller than read times because the operating system does not write inodes to disk immediately; however, the

| *System call* | *Time w/o Pedigree (ms)* | *Time with Pedigree (ms)* | |
|---|---|---|---|
| `read(2)` | 73.632 | Total<br>Retrieve labels | 84.707 *(Overhead: 15.04%)*<br>↪ 99.73% |
| `write(2)` | 6.471 | Total<br>Retrieve labels | 6.8655 *(Overhead: 6.09%)*<br>↪ 93.17% |
| `send(2)` | 0.0048 | Total<br>Marshall labels & Push to label store | 0.3034 *(Overhead: 6220.83%)*<br>↪ 99.42% |
| `recv(2)` | 0.0016 | Total<br>Fetch labels from label store and Unmarshall | 0.2995 *(Overhead: 18618.75%)*<br>↪ 99.43% |

**Table 4: Average time taken for system calls with and without Pedigree. We also show the time overhead of Pedigree, and the percent time spent in the most time-consuming activity when using Pedigree.**

measurement represents the time for the `write(2)` system call to return both with and without Pedigree.

The overhead for individual `send(2)` and `recv(2)` calls are understandably larger with Pedigree because `send(2)` and `recv(2)` involve one round-trip time to the label store: `send(2)` experiences about a 63× slowdown, and `recv(2)` has about an 187× slowdown. Over 99% of this time—approximately 0.29 milliseconds—is spent in from pushing the sending process's label to the label store in the case of `send(2)`, or retrieving the sending process's label from the label store in the case of `recv(2)`. This measurement represents a worst-case scenario where every `send(2)` or `recv(2)` requires contacting the global label store; as we show in Section 7.2, this overhead is negligible for large flows with many sends and receives.

**Macrobenchmark: Compilation.** To compare the overhead for read- and write-intensive applications, we compare the time to compile three packages—GNU wget, GNU bash, and Apache. httpd. These packages have 69, 112, and 254 C header files and 65, 241, and 564 C source files respectively; after compilation, they generate 180, 227, and 912 new files. We add a label to each C header file so that the label propagates to all compiled object files and binaries. Figure 6 presents the average system, user, and wall-clock times taken for compilation on our test machine with and without the Pedigree labeler. The user and system times do not increase appreciably with Pedigree, implying that most of the overhead is in I/O, such as retrieving labels for files from the on-disk local label store as opposed to CPU-bound tasks. The relative increase in compilation times is smaller for the larger packages: httpd compilation with Pedigree takes 172.4 ms instead of 165.57 ms (overhead: 4.1%), whereas wget compilation with Pedigree takes 19.55 ms instead of 16.39 ms (overhead: 19.2%). The lower percentage overhead for compiling httpd results from amortization of disk access times: for httpd, the label of a header file fetched from disk remains in the in-memory label cache and can be accessed quickly for compiling many other source files; on the other hand, wget has fewer C source files than header files.

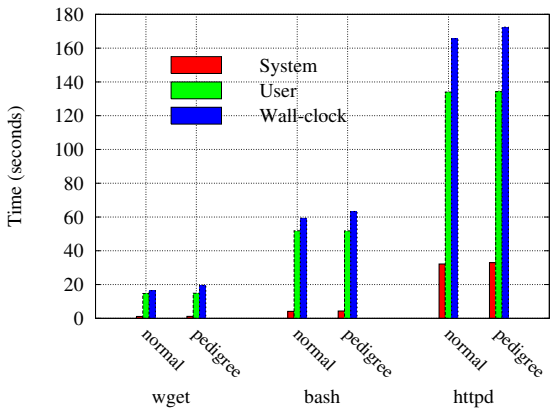



**Figure 6: Average time to compile three software packages with and without Pedigree.**

## 7.2 Network Enforcer Overhead

In this section, we compare the overhead of Pedigree's network enforcer with other setups, as shown in Figure 7. Our baseline is the total time taken to forward a flow across a switch without any extra logic ("Direct forwarding"). Next, we measure the overhead with a software OpenFlow switch in conjunction with a simple NOX controller that reads the first packet of a flow and installs a *permit* flow table entry for the flow without any processing ("OpenFlow/Controller"). Third, we measure the overhead of Pedigree's network enforcer, implemented as a custom controller; this scenario is similar to the previous setting, except that the network enforcer retrieves the sending process's labels and capabilities from the global label store and capability database before permitting it to proceed. We compare these approaches to a traditional DLP approach of applying a regular expression on the content of every packet; in this case, the switch forwards each packet of the flow to the controller before proceeding.

Figure 7 shows the worst-case transfer time with each setup over 100 runs. Pedigree adds measurable overhead to the transfer time of small flows due to the time to fetch the sending process's label and capabilities; as the flow size increases, this overhead is amortized. For a large flow of 128 MB, the transfer time with the network enforcer is 2.052 seconds, compared to 1.587 seconds for

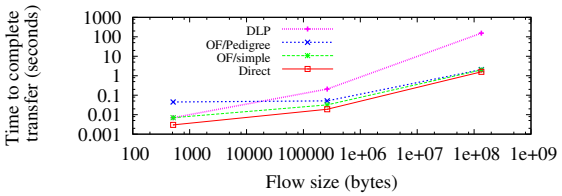



**Figure 7: Transfer times for three flow sizes (512 bytes, 2 MB, and 128 MB) using direct forwarding, OpenFlow with a simple controller, OpenFlow with a Pedigree network enforcer, and a deep-packet inspector that inspects the content of each packet for DLP.**

direct forwarding, 1.995 seconds for a simple controller, and 154.24 seconds with deep packet inspection.

## 7.3 Lived-in Experience

A few members of our lab have used Pedigree on their GNU/Linux desktops and laptops to evaluate usability issues including: (1) Is the system-call overhead of Pedigree noticeable in day-to-day activities; and (2) Does dealing with sensitive-labeled files using common applications work as expected?

System-call overhead is noticeable only for read- or write intensive applications operating on files with non-empty labels, such as copying a large file or compiling a package. Interactive activities such as opening or editing documents introduces no noticeable delay. Users did not observe overhead with networked applications such as opening a Web page or sending an email. Certain applications, however, tend to "leak" labels to resources that should not typically be reading or accessing sensitive data. For example, a Gnome editor (`gedit`) may communicate the title of an open file with the system-wide applet `gnome-panel` *after it reads the file contents*. If the file is labeled sensitive, `gnome-panel` will automatically acquire this label. If `gnome-panel` subsequently communicates to another application, that process may also unintentionally acquire the label. Although we cannot prevent such label "leakage", we find that few applications (*e.g.*, `gnome-panel`, the `D-Bus` protocol, etc.) are responsible for most leaks. If the in-kernel labeler is aware of these applications, it can ensure that they are not allowed to read data from sensitive processes.

# 8. Attacks and Defenses

We describe how Pedigree handles attacks against the local label store, the labeler, and the OS kernel.

**Attacks on the label store and kernel binaries.** Although standard OS APIs protect the labeler and label store from malicious applications, physical attacks, such as removing the primary hard disk, or booting off secondary media to read the primary disk, are possible. To prevent unauthorized access to labels, we store the local label store on an encrypted partition that is not accessible from user space. Before shutdown, the labeler flushes all persistent resource labels to the local store. It then negotiates an encryption key with the labeler authentication service for encrypting the local store. This key and checksum of the label store are sent to the authentication service and are not stored in plaintext on the host. When the labeler authenticates itself at bootup, it retrieves this key and the label store's checksum; it can detect tampering if the checksums do not match.

An additional stage of encryption can prevent malicious users from reading or modifying the kernel binaries (including the labeler and enforcer code): After the label store is encrypted, the kernel encrypts the boot partition (*e.g.*, /boot) using another per-boot key that is either pushed to a central location, or stored in sealed storage on a local Trusted Platform Module (TPM) [3]. A custom bootloader can retrieve the key to decrypt the boot partition at boot time.

**Attacks on the running OS or the Pedigree module.** Although rare, OS vulnerabilities may allow attackers to remove or disable protections provided by the labeler or the OS. The attacker can disable the labeler or cause it to not properly label resources that read secret information. Fortunately, an attacker who compromises only a single host cannot access or modify labels on other hosts. A *privilege-escalation vulnerability* may occur either due to kernel bugs, or bugs in applications running as a privileged user. To defend against such attacks, we separate superuser privileges from those required to modify the kernel. Superusers can perform privileged tasks such as installation of system-wide programs and raw socket I/O, but only enterprise administrators retain full control over the kernel (in contrast to current systems, where the superuser also fully controls the kernel). The Pedigree kernel prevents operations that attempt to remove core kernel modules (including Pedigree) or reinstrument certain functionality (*e.g.*, inserting a module without a valid signature from an enterprise administrator [39]). In a typical enterprise, most users do not require privileges for kernel modifications, so even an exploit that allows a malicious program to attain superuser privileges cannot disable IFC checks.

Pedigree cannot defend against a vulnerability that allows the attacker to insert code into the kernel through non-standard channels. This attack is serious in monolithic kernels such as Linux, because every kernel component is equally "trusted". Fortunately, serious kernel vulnerabilities have at most resulted in privilege escalation in user-space [20, 21], so the possibility for kernel compromise itself is low. To stem the impact of these types of vulnerabilities, groups have been working to harden the Linux kernel to better contain covert channels, such as grsecurity [12]; Pedigree could be implemented on these kernels.

**Covert Channels.** A malicious process could attempt to

covertly transfer information to another, for example, by modulating its memory usage. Covert channels are difficult to disable without OS re-design (*e.g.*, HiStar [50]). Pedigree (and the DIFC OS Flume [19]) runs on Linux, which has no inherent support to detect or prevent covert channels, but Pedigree could also be run on a secure OS that better defends against covert channels such as HiStar.

## 9. Related Work

Distributed information flow control has been applied to various systems since the 1970s [7]. Static tainting approaches [15, 25, 30, 43] can taint certain variables or portions of memory with security classes, but these methods require the programmer to assign security labels at programming time. Many operating systems have attempted to secure a *single* host against exploits or to prevent security breaches (*e.g.*, exfiltration), starting as early as 1975 with the Hydra operating system [6] to more recent work (*e.g.*, Taos [47]). Also in the operating systems layer are Tripwire [42] to discover changes to critical files, and mandatory access control models such as SELinux [37] and AppArmor [2]. While these frameworks also leverage system call hooks to enforce access control, they implement static policies based on type- or role-based access control.

Dynamic taint analysis techniques have seen a resurgence recently, with TaintTracker [27], a mechanism to monitor information flows at the instruction level to detect potential exploits on a host; and process coloring [16], which focuses on tracking interactions between resources ("color diffusion") used for early detection of resources on a host that possess "colors" of a vulnerable process. More recent efforts include Panorama [49], Privacy Oracle [17], TaintDroid [9], and Neon [52]. Although these systems work to track information leakage from legacy applications, they, unlike Pedigree, (1) track information flow using a fixed set of non-expressive taints, (2) all concentrate on leakage prevention from a single host, and (3) have static policies (*e.g.*, data input from sensors is automatically tainted), which limits user control and applicability. These systems also run with some form of instruction-level instrumentation which can cause more overhead than Pedigree's system call interposition; on the other hand, these systems also track taints at a finer granularity than Pedigree. LIFT [33] reduces the overhead of propagating taint information; others have targeting reducing tainting overhead for Web applications [14, 28, 31]. Hardware defenses include preventing tainted data from being used at a jump destination address in a branch or jump instruction [18, 40], and tracking instances when tainted pointers are dereferenced [5]. Tainting has also been applied to other areas, including program understanding, software testing, and debugging (*e.g.*, [10, 22, 23]).

Pedigree bridges the gap between dynamic taint analysis on legacy applications, and systems that are inspired by the use of labels for information flow control [25]. Operating systems such as Asbestos [44] and HiStar [50] are designed to allow containment of applications that can be separated into trusted and untrusted processes. Flume [19] modifies Linux to allow similar policies, at the price of a large trusted computing base and any covert channels present in the kernel. Dstar [51] is an attempt to extend HiStar's decentralized information flow control to the network, using signed messages to transfer labels and capabilities between hosts. Although Dstar is similar in concept to our work, it requires HiStar on both machines to understand and allow label transfers. Pedigree is more general, because (1) it allows unmodified applications to acquire labels, (2) policy is centralized, which allows for easy auditing and enforcement, and (3) users and administrators manage capabilities instead of processes.

## 10. Conclusion

Enterprises remain vulnerable to data leaks that cost network administrators countless hours and billions of dollars. It has been difficult to achieve network-wide information-flow control, primarily because most existing solutions require modifications to operating systems, applications, or both. Unfortunately, most enterprise networks have legacy applications, operating systems, and infrastructure, making many existing solutions untenable. We presented Pedigree, a system that tracks and controls information flow as data travels between processes, both on a single host and across the network. Pedigree can enforce information flow policy either on the hosts themselves or in the network.

Pedigree makes information flow control more practical for legacy systems and across networks; its design and mechanisms may also apply in other settings where administrators cannot constrain the applications or operating systems that users run, such as cloud environments. Perhaps the most significant remaining challenge is the issue of granularity: assigning labels at the granularity of files and processes may sometimes be too coarse-grained, but byte-level tainting or labeling in memory is unlikely to scale. One possible avenue for future work would be to create a hybrid approach that combines Pedigree with byte-level tainting.

## REFERENCES


[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[2] AppArmor Application Security for Linux. `http://www.novell.com/linux/security/apparmor/`.

[3] S. Bajikar. Trusted Platform Module (TPM) based Security for Notebook PCs. `http://www.intel.com/design/mobile/platform/downloads/trusted_platform_module_white_paper.pdf`.

[4] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report 2547, MITRE Corporation, 1976.

[5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of The International COnference on Dependable Systems and Networks*, 2005.

[6] E. S. Cohen and D. Jefferson. Protection in the hydra operating system. pages 141–160, 1975.

[7] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[8] Emulab. `http://www.emulab.net/`.

[9] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2010.

[10] V. Ganesh, T. Leek, and M. C. Rinard. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 474–484, 2009.

[11] Goldman's Secret Sauce Could be Loose Online. `http://is.gd/dmahZ`.

[12] grsecurity: Enhancing Security for the Linux Kernel. `http://grsecurity.net`.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[14] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 13th International World Wide Web Conference (WWW04)*, pages 40–52, 2005.

[15] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From Trusted to Secure: Building Applications that Enforce System Security. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[16] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, and E. H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *ICDCS*, June 2006.

[17] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, and T. K. Gabriel Maganis. Privacy Oracle: a System for Finding Application Leaks with Black Box Differential Testing. In *ACM Conference on Computer and Commnications Security*, 2008.

[18] J. Kong, C. C. Zou, and H. Zhou. Improving Software Security via Runtime Instruction-level Taint Checking. In *ASID '06: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.

[19] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[20] Linux kernel 32 bit compatibility mode vulnerability. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3081`.

[21] Linux kernel RDS protocol vulnerability. `http://www.kb.cert.org/vuls/id/362983`.

[22] W. Masri. Exploiting the empirical characteristics of program dependences for improved forward computation of dynamic slices. *Empirical Softw. Eng.*, 13(4):369–399, 2008.

[23] W. Masri, D. Leon, and A. Podgurski. An empirical study of test case filtering techniques based on exercising information flows. In *IEEE Transactions on Software Engineering*, pages 454–477, Piscataway, NJ, USA, 2007. IEEE Press.

[24] McAfee Data Loss Prevention. `http://www.mcafee.com/us/enterprise/products/data_protection/data_loss_prevention/data_loss_prevention.html`.

[25] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[26] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, Saint-Malô, France, Oct. 1997.

[27] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2005.

[28] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[29] OpenFlow Switch Consortium. `http://www.openflowswitch.org/`, 2008.

[30] P.Broadwell, M.Harren, N.Sastry. Scrash: A system for generating security crash information. In *Proc. 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[31] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[32] Ponemon Institute. Fifth Annual US Cost of Data Breach Study. `http://www.ponemon.org/local/upload/fckjail/generalcontent/18/file/US_Ponemon_CODB_09_012209_sec.pdf`.

[33] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Redis. `http://redis.io/`.

[35] Redis Benchmarks. `https://code.google.com/p/redis/wiki/Benchmarks`.

[36] RSA Data Loss Prevention. `http://www.rsa.com/node.aspx?id=3426`.

[37] Security-Enhanced Linux. `http://www.nsa.gov/research/selinux/`.

[38] M. Shaw. Leveraging good intentions to reduce unwanted network traffic. In *Proc. USENIX Steps to Reduce Unwanted Traffic on the Internet workshop*, San Jose, CA, July 2006.

[39] Signed Kernel Modules. `http://www.linuxjournal.com/article/7130`.

[40] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS 04)*, pages 85–96, New York, NY, USA, 2004. ACM.

[41] Symantec Data Loss Prevention. `http://www.symantec.com/business/products/family.jsp?familyid=data-loss-prevention`.

[42] Tripwire Configuration Audit. `http://www.tripwire.com/`, 2009.

[43] U.Shankar, K.Talwar, J.Foster, D.Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.

[44] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazieres. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems*, 25(4):1–43, Dec. 2007.

[45] Verizon 2010 Data Breach Investigations Report. `http://www.verizonbusiness.com/go/2010databreachreport/`.

[46] Army Broadens Inquiry Into WikiLeaks Disclosure. `http://www.nytimes.com/2010/07/31/world/31wiki.html`.

[47] E. Wobber, M. Abadi, and M. Burrows. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[48] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[49] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2007.

[50] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *Proc. 7th USENIX OSDI*, Seattle, WA, Nov. 2006.

[51] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing Distributed Systems with Information Flow Control. In *Proc. 5th USENIX NSDI*, San Francisco, CA, Apr. 2008.

[52] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. C. Snoeren, G. M. Voelker, S. Savage, and A. Vahdat. Neon: System Support for Derived Data Management. In *ACM Conference on Virtual Execution Environments*, Mar. 2010.

[53] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. In *ACM Conference on Computer and Communications Security*, Nov. 2010.