Information Flow Control for Standard OS Abstractions

Maxwell Krohn Alexander Yip Micah Brodsky Natan Cliffer
M. Frans Kaashoek Eddie Kohler† Robert Morris
MIT CSAIL †UCLA
http://flume.csail.mit.edu/

ABSTRACT

Decentralized Information Flow Control (DIFC) [24] is an approach to security that allows application writers to control how data flows between the pieces of an application and the outside world. As applied to privacy, DIFC allows untrusted software to compute with private data while trusted security code controls the release of that data. As applied to integrity, DIFC allows trusted code to protect untrusted software from unexpected malicious inputs. In either case, only bugs in the trusted code, which tends to be small and isolated, can lead to security violations.

We present *Flume*, a new DIFC model and system that applies at the granularity of operating system processes and standard OS abstractions (e.g., pipes and file descriptors). Flume eases DIFC's use in existing applications and allows safe interaction between conventional and DIFC-aware processes. Flume runs as a user-level reference monitor on Linux. A process confined by Flume cannot perform most system calls directly; instead, an interposition layer replaces system calls with IPC to the reference monitor, which enforces data flow policies and performs safe operations on the process's behalf. We ported a complex Web application (MoinMoin wiki) to Flume, changing only 2% of the original code. The Flume version is roughly 30–40% slower due to overheads in our current implementation but supports additional security policies impossible without DIFC.

Categories and Subject Descriptors:
D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls, Access controls*; D.4.7 [**Operating Systems**]: Organization and Design; C.5.5 [**Computer System Implementation**]: Servers

General Terms: Security, Design, Performance

Keywords: distributed information flow control, DIFC, endpoints, reference monitor, system call interposition, Web services

1 INTRODUCTION

As modern applications grow in size, complexity and dependence on third-party software, they become more susceptible to security flaws. Decentralized information flow control (DIFC) [24], a variant of classic information flow control [1, 2, 6], can improve the security of complex applications, even in the presence of potential exploits. Existing DIFC systems operate as programming language abstractions [24] or are integrated into communication primitives in new operating systems [8, 38]. These approaches have advantages, such as fine-grained control of information flow and high performance, but require a shift in how applications are developed. Flume instead provides process-level DIFC as a minimal extension to the communication primitives in *existing* operating systems, making DIFC work with the languages, tools, and operating system abstractions already familiar to programmers.

The Flume system provides DIFC at the granularity of processes, and integrates DIFC controls with standard communication abstractions such as pipes, sockets, and file descriptors, via a user-level reference monitor. Its interface helps programmers secure existing applications and write new ones with existing tools and libraries. Flume enforces the DIFC policy as the application runs.

A typical Flume application consists of processes of two types. *Untrusted* processes do most of the computation. They are constrained by, but possibly unaware of, DIFC controls. *Trusted* processes, in contrast, are aware of DIFC and set up the privacy and integrity controls that constrain untrusted processes. Trusted processes also have the *privilege* to selectively violate classical information flow control—for instance, by *declassifying* private data (perhaps to export it from the system), or by *endorsing* data as high-integrity. This privilege is distributed among the trusted processes according to application policy, making it decentralized (the "D" in DIFC). Though bugs in the trusted code can lead to compromise, bugs elsewhere in the application cannot, and trusted code can stay relatively isolated and concise even as the application expands.

A central challenge for Flume is to accommodate processes that use existing communication interfaces such as sockets and pipes but also need to specify how and when they use their privileges. It would be awkward to, for example, modify each call to `read` or `write` to take arguments indicating whether privilege should be applied. Worse, the conventional process interface is rife with channels that "leak" information, like network sockets. A system could simply mark these channels off-limits, restricting the process interface to those system calls with obvious and controllable information flow, but this approach would make many libraries unusable. Flume instead seeks to restrict access to these uncontrolled channels only when necessary.

Our solution is an *endpoint* abstraction. Flume represents each resource a process uses to communicate as an endpoint, including pipes, sockets, files, and network connections. A process can specify what subset of its privileges should be exercised when communicating through each endpoint. Uncontrolled channels are modeled as endpoints that exit the DIFC system; Flume ensures that no process can have both an uncontrolled channel and access to private data it cannot declassify.

We built Flume in user-space (with a few small kernel patches) for implementation convenience and portability: the implementation runs on Linux and OpenBSD. Unlike prior systems that provide DIFC as part of a new kernel design (e.g., Asbestos [8] and HiStar [38]), Flume takes advantage of large existing efforts to maintain and improve the kernel support for hardware, NFS, RAID, SMP, etc. The disadvantage is that Flume's trusted computing base is many times larger than those of dedicated DIFC kernels, leaving

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

it vulnerable to security flaws in the underlying operating system. Also, Flume's user space implementation incurs some performance penalties and may expose covert channels that deeper kernel integration would close.

To evaluate Flume's programmability, we ported a complex and popular application, MoinMoin wiki [22], to the Flume system. MoinMoin is a feature-rich Web document sharing system (91,000 lines of Python code), with support for access control lists, indexing, Web-based editing, versioning, syntax highlighting for source code, downloadable "skins", etc. We captured Moin's access control policies with DIFC-based equivalents, thereby moving the security logic out of the main application and into a small, isolated security module about a thousand lines long. Only bugs in the security module, as opposed to the large tangle of Moin code and its plug-ins, can compromise end-to-end security. We also implemented a Moin security policy that could not exist without Flume: *end-to-end integrity protection*. Moin can pull third-party plug-ins into its address space, but with end-to-end integrity protection, users can enforce that selected plug-ins never touch (and potentially corrupt) their sensitive data, either on input or output.

FlumeWiki achieves these security goals with only a thousand lines of modification to the original MoinMoin system (in addition to the new security module). Though prior DIFC work has succeeded in sandboxing legacy applications [38] or rewriting them anew [8, 15], the "drop-in" replacement of an existing access control policy with a DIFC-based one is a new result. In at least three cases, FlumeWiki closes security holes in the original Moin. Experiments with FlumeWiki on Linux show that the new system performs 43% slower than the original in read workloads, and 34% slower on write workloads. Slow-downs are due primarily to Flume's user-space implementation. We expect that for many Web sites the prototype's performance is adequate.

This paper's contributions include:

- New DIFC rules that fit standard operating system abstractions well and that are simpler than those of Asbestos and HiStar. Flume's DIFC rules are close to rules for classic "centralized" information flow control [1, 2, 6], with small extensions for decentralization and communication abstractions found in widely-used operating systems.
- The first design and implementation of process-level DIFC for stock operating systems (OpenBSD and Linux).
- Refinements to Flume DIFC required to build real systems, such as machine cluster support, and DIFC primitives that scale to large numbers of users.
- A full-featured DIFC Web site (FlumeWiki) with novel end-to-end integrity guarantees, composed largely of existing code.

All security claims made for Flume rely on two important assumptions. First, machines running Flume must not have security bugs that result in super-user privileges. Second, though the Flume design makes a concerted effort to close all known covert *storage* channels (and we indicate where it falls short), Flume assumes processes running on the machine do not leak data via covert *timing* channels. For example, an exploit might transmit sensitive information by carefully modulating its CPU use in a way observable by other processes. Covert timing channels are present in all existing DIFC systems; they can be reduced but not eliminated, particularly for systems connected to the network [38].

The rest of this paper proceeds as follows: Section 2 describes related work. Section 3 considers an abstract definition for DIFC, while Section 4 presents endpoints and the instantiation of DIFC on Unix. Section 5 and 6 describe the design of Flume and its file system, and Section 7 describes the FlumeWiki. Section 8 provides a performance evaluation, and Section 9 concludes.

2 Related Work

There has been much work on improving security on stock operating systems, including buffer overrun protection (e.g., [5, 18]), system call interposition (e.g., [11, 12, 16, 27, 32]), isolation techniques (e.g., [13, 17]), virtual machines (e.g., [14, 33, 35]), and recovering from compromises (e.g., [7]). Flume uses some of these techniques for its implementation (e.g., LSMs [36] and systrace [27]), but Flume is more closely related to mandatory access control and specifically decentralized information flow control.

Mandatory access control (MAC) [28] refers to a system security plan in which security policies are *mandatory* and not enforced at the discretion of the application writers. In many such systems, software components may be allowed to read private data but are forbidden from revealing it. Traditional MAC systems intend that an administrator set a single system-wide policy. When servers run multiple third-party applications, however, administrators cannot understand every application's detailed security logic. DIFC promises to support such situations better than most MAC mechanisms, because it partially delegates the setting of policy to the individual applications.

SELinux [20] and TrustedBSD [34] are recent examples of stock operating systems modified to support many MAC policies. They include interfaces for a security officer to dynamically insert security policies into the kernel, which then limit the behavior of kernel abstractions like inodes and tasks [30]. Flume, like SELinux, uses the Linux security module (LSM) framework in its implementation [36]. However, SELinux and TrustedBSD do not allow untrusted applications to define and update security policies (as in DIFC). If SELinux and TrustedBSD were to provide such an API, they would need to address the challenges considered in this paper.

TightLip [37] implements a specialized form of IFC that prevents privacy leaks in legacy applications. TightLip users tag their private data and TightLip prevents that private data from leaving the system via untrusted processes. Unlike TightLip, Flume and other DIFC systems (e.g. Asbestos and HiStar) support multiple security classes, which enable safe commingling of private data and security policies other than privacy protection.

IX [21] and LOMAC [10] add information flow control to Unix, but again with support for only centralized policy decisions. Flume faces some of the same Unix-related problems as these systems, such as shared file descriptors that become storage channels.

Myers and Liskov introduced a decentralized information model [23], thereby relaxing the restriction in previous information flow control systems that only a security officer could declassify. JFlow and its successor Jif are Java-based programming languages that enforce DIFC within a program, providing finer-grained control than Flume [24]. One benefit is that Jif can limit declassification privileges to specific function(s) within a process, rather than (as with Flume) to the entire process. On the other hand, Jif requires applications (such as Web services [4]) to be rewritten while Flume provides better support for applying DIFC to existing software. Flume's DIFC rules (Section 3) are inspired by Jif's but split readership and ownership into separate per-process labels, and allow ownership to be transfered as capabilities. Flume's endpoints (Section 4) provide the glue between software written for an existing API and DIFC.

Asbestos [3, 8] and HiStar [38] incorporate DIFC into new operating systems, applying labels at the granularity of unreliable messages between processes (Asbestos) or threads, gates, and segments (HiStar). Flume's labels are influenced by Asbestos's and

incorporate HiStar's improvement that threads must explicitly request label changes (since implicit label changes are covert channels). Asbestos and HiStar labels combine mechanisms for privacy, integrity, authentication, declassification privilege, and port send rights. Flume separates (or eliminates) these mechanisms in a way that is intended to be easier to understand and use.

HiStar implements an untrusted, user-level Unix emulation layer using DIFC-controlled low-level kernel primitives. A process that uses the Unix emulation layer but needs control over the DIFC policy would have to understand and manipulate the mapping between Unix abstractions and HiStar objects. Such controls could be factored into a new library, and this paper answers the question of what such a library might look like.

As new operating systems, Asbestos and HiStar have smaller TCBs than Flume, and can tailor their APIs to work well with DIFC. However, they don't automatically benefit from mainstream operating systems' frequent updates, such as new hardware support and kernel improvements.

3 Information Flow In Flume

This section describes Flume's approach to decentralized information flow control. The *Flume model* closely follows traditional IFC [6], adding a simple new representation of how processes maintain and use decentralized privilege.

3.1 Tags and Labels

Flume uses *tags* and *labels* to track data as it flows through a system. Let $\mathscr{T}$ be a very large set of opaque tokens called *tags*. A tag t carries no inherent meaning, but processes generally associate each tag with some category of secrecy or integrity. Tag b, for example, might label Bob's private data.

Labels are subsets of $\mathscr{T}$. Labels form a lattice under the partial order of the subset relation [6]. Each Flume process p has two labels, S_p for secrecy and I_p for integrity. If tag $t \in S_p$, then the system conservatively assumes that p has seen some private data tagged with t. A process whose secrecy label contains one or more tags requires independent consent for each tag to reveal data publicly. For integrity, if $t \in I_p$, then every input to p has been endorsed as having integrity for t. Files (and other objects) also have secrecy and integrity labels.

Although any tag can appear in any type of label, in practice secrecy and integrity usage patterns are so different that a tag is used *either* in secrecy labels *or* in integrity labels, not both. We therefore sometimes refer to a "secrecy tag" or an "integrity tag".

We will illustrate Flume's information flow with several running examples.

Example: Secrecy Alice and Bob share access to a server, but wish to keep some files (but not all) secret from one another. Misbehaving software can complicate even this basic policy; for example, Bob might download a text editor that, as a side effect, posts his secret files to a public Web site, or writes them to a public file in `/tmp`. Under the typical OS security plan, Bob can only convince himself that the text editor won't reveal his data if he (or someone he trusts) audits the software and all of its libraries.

With information flow control, Bob can reason about the editor's (mis)behavior without auditing its code. Say that tag b represents Bob's secret data. As described below, Bob explicitly trusts some processes to export his data out of the system. For now, consider all other (i.e. *untrusted*) processes, like the text editor and its minions. Bob seeks four guarantees for each such process p: (1) if p reads his secret files, then $b \in S_p$; (2) p with $b \in S_p$ can only write to other processes (and files) q with $b \in S_q$; (3) p cannot remove b from S_p; (4) p with $b \in S_p$ cannot transmit information over an uncontrolled channel (like the network). If all four conditions hold, then a simple inductive argument shows that the editor cannot leak Bob's data from the system.

Example: Integrity A complementary policy involves integrity. Say Charlie has administrator privilege on his machine, allowing him to edit sensitive files (e.g., `/etc/rc`, the script that controls which processes run with superuser privileges when a machine boots up). However, other users constantly update libraries and download new software, so Charlie lacks confidence that all editors on the system will faithfully execute his intentions when he edits `/etc/rc`. A path misconfiguration might lead Charlie to access a malicious editor that shares a name with a responsible editor, or a good editor that links at runtime against phony libraries.

Secrecy protection won't help Charlie; rather, he needs an end-to-end guarantee that *all* files read when editing `/etc/rc` are uncorrupted. Only under these integrity constraints should the system allow modifications to the file. Say that an integrity tag v represents data that is "vendor-certified." As described below, some processes on the system can *endorse* files and processes, giving them integrity v. For now, consider all other processes, like the text editor. Charlie seeks four guarantees for each such process p: (1) if p modifies `/etc/rc` then $v \in I_p$; (2) a process p with $v \in I_p$ cannot read from files or processes that lack v integrity, and only uncorrupted files (like binaries and libraries) have v integrity; (3) a process p cannot add v to I_p; and (4) p with $v \in I_p$ cannot accept input from uncontrolled channels (like the network). If all four conditions hold, Charlie knows that changes to `/etc/rc` were mediated by an uncorrupted editor.

3.2 Decentralized Privilege

In centralized IFC, only a trusted "security officer" can create new tags, subtract tags from secrecy labels (*declassify* information), or add tags to integrity labels (*endorse* information). In Flume DIFC, any process can create new tags, which gives that process the privilege to declassify and/or endorse information for those tags.

Flume represents privilege using two *capabilities* per tag. For tag t, the capabilities are t^+ and t^-. Each process *owns* a set of capabilities O_p. A process with $t^+ \in O_p$ *owns* the t^+ capability, giving it the privilege to add t to its labels; and a process with $t^- \in O_p$ can remove t from its labels. In terms of secrecy, t^+ lets a process add t to its secrecy label, granting itself the privilege to receive secret t data, while t^- lets it remove t from its secrecy label, effectively declassifying any secret t data it has seen. In terms of integrity, t^- lets a process remove t from its integrity label, allowing it to receive low-t-integrity data, while t^+ lets it add t to its integrity label, endorsing the process's current state as high-t-integrity. A process that owns both t^+ and t^- has *dual privilege* for t and can completely control how t appears in its labels. The set $D_p = \{t \mid t^+ \in O_p \text{ and } t^- \in O_p\}$ represents all tags for which p has dual privilege.

Any process can allocate a tag. Tag allocation yields a randomly-selected tag t and sets $O_p \leftarrow O_p \cup \{t^+, t^-\}$, granting p dual privilege for t.

Flume also supports a *global* capability set $\mathbf{O}$. Every process owns every capability in $\mathbf{O}$: the system enforces that $\mathbf{O} \subseteq O_p$ for all p. Only tag allocation can change $\mathbf{O}$; an allocation parameter determines whether the new tag's t^+, t^-, or neither is added to $\mathbf{O}$ (and thus to every current and future process's O_p). A process can test whether a given capability is in $\mathbf{O}$, but to prevent data leaks, processes cannot enumerate the contents of $\mathbf{O}$. A process p can, however, enumerate its non-global capabilities (those in $O_p - \mathbf{O}$).

Two processes can transfer capabilities so long as they can communicate. A process can freely drop non-global capabilities (though we add a restriction in Section 4.2).

For a set of tags T, we define the capability set $\{T\}^+$ as $\{t^+ \mid t \in T\}$, and similarly for $\{T\}^-$.

Example: Secrecy Bob can maintain the secrecy of his private data with a policy called *export protection*. One of Bob's processes allocates the secrecy tag b used to mark his private data; during the allocation, b^+ is added to $\mathbf{O}$, but only the allocating trusted process gets b^-. Thus, any process p can add b to S_p and therefore read b-secret data, but only processes that own b^- (i.e., Bob's trusted process and its delegates) can declassify this data and export it out of the system. (We describe how to create b-secret data below.)

A related but more stringent policy is called *read protection*. A process allocates a secrecy tag t, but neither t^+ nor t^- is added to $\mathbf{O}$. By controlling t^+, the allocating process can limit which other processes can *view* t-secret data, as well as limiting which other processes can declassify t-secret data. Read protection can prevent secret data from leaking through covert channels, including timing channels [38].

Example: Integrity Another policy, *integrity protection*, is suitable for our integrity example. A "certifier" process allocates integrity tag v, and during the allocation, v^- is added to $\mathbf{O}$. Now, any p process can *remove* v from I_p, but only the certifier has v^+. The ability to add v to an integrity label—and thus to endorse information as high-v-integrity—is tightly controlled by the certifier. Charlie requests of the certifier to edit `/etc/rc` using an editor of his choice. The certifier forks, creating a new process with v integrity; the child drops the v^+ capability and attempts to execute Charlie's chosen editor. With $v \in I_p$ and $v^+ \notin O_p$, the editor process can only read high-integrity files (be they binaries, libraries, or configuration files) and therefore cannot come under corrupting influences.

These three policies—export protection, read protection, and integrity protection—enumerate the common uses for tags, although others are possible.

3.3 Security

The Flume model assumes many processes running on the same machine and communicating via messages, or "flows". The model's goal is to track data flow by regulating both process communication and process label changes.

Definition 1. A system is secure in the Flume model if and only if all allowed process label changes are "safe" (Definition 2) and all allowed messages are "safe" (Definition 3).

We define "safe" label changes and messages below. Though many systems might fit this general model, we focus on the Flume system in particular in Section 4.

Safe Label Changes In the Flume model (as in HiStar), only process p itself can change S_p and I_p, and must request such a change explicitly. Other models allow a process's label to change as the result of receiving a message [8, 10, 21], but implicit label changes turn the labels themselves into covert channels [6, 38]. When a process requests a change, only those label changes permitted by a process's capabilities are safe:

Definition 2. For a process p, let L be S_p or I_p, and let L' be the new value of the label. The change from L to L' is *safe* if and only if:

$$\{L' - L\}^+ \cup \{L - L'\}^- \subseteq O_p.$$

For example, say process p wishes to subtract tag t from S_p, to achieve a new secrecy label S'_p. In set notation, $t \in S_p - S'_p$, and such a transition is only safe if p owns the subtraction capability for t (i.e. $t^- \in O_p$). The same logic holds for addition, yielding the above formula.

Safe Messages Information flow control restricts process communication to prevent data leaks. The Flume model restricts communication among unprivileged processes as in classical IFC: p can send a message to q only if $S_p \subseteq S_q$ ("no read up, no write down" [1]) and $I_q \subseteq I_p$ ("no read down, no write up" [2]).

Processes that hold some privilege are less restricted, and we relax our rules accordingly. Specifically, if two processes *could* communicate by changing their labels, sending a message using the centralized rules, and then restoring their original labels, then the model can safely allow the processes to communicate without label changes. A process can make such a temporary label change only for tags in D_p, for which it has dual privilege. A process p with labels S_p, I_p would get maximum latitude in sending messages if it were to lower its secrecy to $S_p - D_p$ and raise its integrity to $I_p \cup D_p$. It could receive the most messages if it were to raise secrecy to $S_p \cup D_p$ and lower integrity to $I_p - D_p$. The following definition captures these *hypothetical* label changes to determine what messages are safe:

Definition 3. A message from p to q is *safe* iff

$$S_p - D_p \subseteq S_q \cup D_q \quad \text{and} \quad I_q - D_q \subseteq I_p \cup D_p.$$

For processes with no dual privilege ($D_p = D_q = \{\}$), Definition 3 gives the centralized IFC definition for safe flows. On the other hand, if p must send with a hypothetical secrecy label of $S_p - D_p$, then p is declassifying the data it sends to q. If q must receive with secrecy $S_q \cup D_q$, then it is declassifying the data it received from p. In terms of integrity, if p must use an integrity label $I_p \cup D_p$, then it is endorsing the data sent, and similarly, q is endorsing the data received with integrity label $I_q - D_q$. Declassification or endorsement can also occur when a process p makes actual (rather than hypothetical) label changes to S_p or I_p, respectively.

External Sinks and Sources Any data sink or source outside of Flume's control, such as a remote host, the user's terminal, a printer, and so forth, is modeled as an unprivileged process x with permanently empty secrecy and integrity labels: $S_x = I_x = \{\}$ and also $O_x = \mathbf{O}$. As a result, a process p can only write to the network or console if it could reduce its secrecy label to $\{\}$ (the only label with $S_p \subseteq S_x$), and a process can only read from the network or keyboard if it could reduce its integrity label to $\{\}$ (the only label with $I_x \subseteq I_p$).

Objects Objects such as files and directories are modeled as processes with *immutable* secrecy and integrity labels, fixed at object creation. A process p's write to an object o then becomes a flow from p to o; reading is a flow sent from o to p. When a process p creates an object o, p specifies o's labels, subject to the restriction that p must be able to write to o. In many cases, p must also update some referring object (e.g., a process writes a directory when creating a file), and writes to the referrer must obey the normal rules.

Example: Secrecy We now can see how the Flume model enforces our examples' security requirements. In the editor example, Bob requires that all untrusted processes like his editor (i.e., those p for which $b^- \notin O_p$) meet the four stated requirements. We first note that since b is an export-protect tag, $b^- \notin O_p$ implies that $b \notin D_p$.

1. *If process p reads Bob's secret files, then $b \in S_p$:* Bob's secret files are modeled as objects f with $b \in S_f$. Since $b^+ \in \mathbf{O}$, any process can write such files. Reading an object is modeled as an information flow from f to p, which requires that $S_f \subseteq S_p \cup D_p$ by Definition 3. Since $b \in S_f$, and $b \notin D_p$, it follows that $b \in S_p$.
2. *Process p with $b \in S_p$ can only write to other processes (or files) q with $b \in S_q$*: If a process p with $b \in S_p$ successfully sends a message to a process q, then by Definition 3, $S_p - D_p \subseteq S_q \cup D_q$. Since b is in neither D_p nor D_q, then $b \in S_q$.
3. *Processes cannot drop b from S_p*: The process that allocated b kept b^- private, so by Definition 2, only those processes that own b^- can drop b from their secrecy labels.
4. *Process p with $b \in S_p$ cannot transmit information over uncontrolled channels*: An uncontrolled channel x has secrecy label $\{\}$, so by Definition 3, process p can only transmit information to x if it owns b^-, which it does not.

Note that since $b^+ \in \mathbf{O}$, any process (like the editor) can add b to its secrecy label. Such a process p can read Bob's files, compute arbitrarily, and write the resulting data to files or processes that also have b in their secrecy labels. But it cannot export Bob's secrets from the system. Of course if p owned b^- or could coerce a process that did, Bob's security could be compromised. Similar arguments hold for the integrity example.

Example: Shared Secrets The power of decentralized IFC lets Flume users combine their private data in interesting ways without leaking information. Imagine a simple calendar application where all system users keep private data files describing their schedules. A user such as Bob can schedule a meeting with Alice by running a program that examines his calendar file and hers, and then writes a message to Alice with possible meeting times. When Alice gets the message, she responds with her selection. Such an exchange should reveal only what Bob and Alice chose to reveal (candidate times, and the final time, respectively) and nothing more about their calendars. Alice and Bob both export-protect their calendar files with a and b respectively. To reveal to Alice a portion of his calendar, Bob launches a process p with labels $S_p = \{a, b\}$ and $O_p = \{b^-\} \cup \mathbf{O}$. This process can read both calendar files, find possible meeting times, and then lower its S_p label to $\{a\}$ and write these times to a file f labeled $S_f = \{a\}$. Though f contains information about both Alice and Bob's calendars, only Alice can export it—and in particular, Bob himself cannot export it (since it contains Alice's private data). Thus, other users "eavesdropping" on this exchange learn nothing. When Alice logs on, she can use a similar protocol to read Bob's suggestions, choose one, and export that choice to Bob in a file g labeled $S_g = \{b\}$. Bob and Alice have agreed on a meeting time while exposing none of their private information to other users, and even while controlling the information they expose to each other.

4 Endpoints In Flume

This section describes the *Flume system*, a refinement of the Flume *model* from Section 3. The Flume model gives general guidelines for what properties a system ought to uphold to be considered "secure" but does not dictate system specifics such as what API processes use to communicate. Some DIFC kernels like Asbestos expose only unreliable messages (as in Definition 3) to applications, making reliable user-level semantics difficult to achieve. A goal of the Flume system is to better fit existing (i.e. reliable) APIs for process communication—that of Unix in particular—while upholding security in the Flume model.

The Flume system applies DIFC controls to the Unix primitive for communication, the *file descriptor*. Flume assigns an *endpoint* to each Unix file descriptor. A process can potentially adjust the labels on an endpoint, so that all future information flow on the file descriptor, either sent or received, is controlled by its endpoint's label settings.

Relative to raw message-based communication, endpoints simplify application programming. When message delivery fails according to Definition 3, it does so *silently* to avoid data leaks. Such silent failures can complicate application development and debugging. However, when a process attempts and fails to adjust the labels on its endpoints, the system can safely report errors, helping the programmer debug the error. In many cases, once processes properly configure their endpoints, reliable IPC naturally follows.

Endpoints also make many declassification (and endorsement) decisions *explicit*. According to Definition 3, every message a privileged process sends and receives is implicitly declassified (or endorsed), potentially resulting in accidental data disclosure (or endorsement). The Flume system requires processes to explicitly mark those file descriptors that serve as avenues for declassification (or endorsement); others do not allow it.

4.1 Endpoints

When a process p acquires a new file descriptor, it gets a new corresponding *endpoint*. Each endpoint e has its own secrecy and integrity labels, S_e and I_e. By default, $S_e = S_p$ and $I_e = I_p$. A process owns readable endpoints for each of its readable resources, writable endpoints for writable resources, and read/write endpoints for those that are bidirectional. Endpoints meet safety constraints as follows:

Definition 4. A readable endpoint e is *safe* iff

$$(S_e - S_p) \cup (I_p - I_e) \subseteq D_p.$$

A writable endpoint e is safe iff

$$(S_p - S_e) \cup (I_e - I_p) \subseteq D_p.$$

A read/write endpoint is safe iff it meets both requirements.

All IPC now happens between two *endpoints*, not two processes, requiring a new version of Definition 3.

Definition 5. A message from endpoint e to endpoint f is *safe* iff e is writable, f is readable, $S_e \subseteq S_f$, and $I_f \subseteq I_e$.

We can now prove that any safe message between two safe endpoints is also a safe message between the corresponding processes. Take process p with safe endpoint e, process q with safe endpoint f, and a safe message from e to f. In terms of secrecy, that the message between the endpoints is safe implies by Definition 5 that e is writable, f is readable, and $S_e \subseteq S_f$. Since e and f are safe, Definition 4 implies that $S_p - D_p \subseteq S_e$ and $S_f \subseteq S_q \cup D_q$. Thus, $S_p - D_p \subseteq S_q \cup D_q$, and the message between processes is safe for secrecy by Definition 3. A similar argument holds for integrity. ∎

4.2 Enforcing Safe Communication

For the Flume system to be secure in the model defined in Section 3, all messages must be safe. Thus, the Flume system enforces message safety by controlling a process's endpoint configurations (which must *always* be safe), and by limiting the messages sent between endpoints. The exact strategy depends on the type of communication and how well Flume can control it.

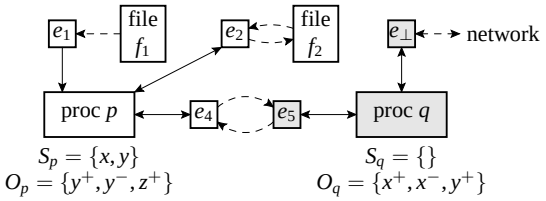

Figure 1: Processes p and q. Assume $\mathbf{O} = \{\}$.

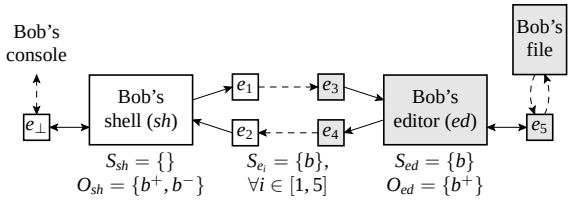

Figure 2: A configuration for Bob's shell and editor. Here, $\mathbf{O} = \{b^+\}$.

IPC First is communication that the Flume reference monitor can completely control, where both ends of the communication are Flume processes and all channels involving the communication are understood: for example, two Flume processes p and q communicating over a pipe or socketpair. Flume can proxy these channels message-by-message, dropping messages as appropriate. When p sends data to q, or vice-versa, Flume checks the corresponding endpoint labels, silently dropping the data if it is unsafe according to Definition 5. A receiving processes cannot distinguish between a message unsent, and a message dropped because it is unsafe; therefore, dropped messages do not leak information.

The endpoints of such a pipe or socketpair are *mutable*: p and q can change the labels on their endpoints so long as they maintain endpoint safety (Definition 4), even if the new configuration results in dropped messages. Verifying that a process p has safe endpoints requires information about p's labels, but not information about q's. Thus, if a process attempts to change a mutable endpoint's label in an unsafe way, the system can safely notify the process of the failure and its specific cause. Similarly, endpoint safety may prevent a process from dropping one or more of its non-global capabilities, or from making certain label changes, until either the endpoint label is changed or the endpoint itself is dropped.

Two processes with different process-wide labels can use endpoints to set up bidirectional (i.e., reliable) communication if they have the appropriate capabilities. For example, in Figure 1, p can set $S_{e_4} = \{x\}$, and q can set $S_{e_5} = \{x\}$, thus data can flow in both directions across these endpoints. In this configuration, p is prohibited from dropping y^- or y^+, since so doing would make e_4 unsafe; similarly, q cannot drop x^- or x^+. Note that reliable two-way communication is needed even in the case of a one-way Unix pipe, since pipes convey flow control information from the receiver back to the sender. Flume can safely allow one-way communication over a pipe by hiding this flow control information and rendering the pipe unreliable; see Section 5.3.

File I/O Second is communication that the Flume reference monitor chooses not to completely control. For example, Flume controls a process's file I/O with coarse granularity: once Flume allows a process to open a file for reading or writing, it allows all future reads or writes to the file (see Section 6.1). Since the reference monitor does not interpose on file I/O to drop messages, it enforces safe communication solely through endpoint labels.

When a process p opens a file f, p can specify which labels to apply the corresponding endpoint e_f. If no labels for e_f are specified, they default to p's. When opening f for reading, p succeeds if e_f is a safe readable endpoint, $S_f \subseteq S_{e_f}$ and $I_{e_f} \subseteq I_f$. When opening f for writing, p succeeds if e_f is a safe writable endpoint, $S_{e_f} \subseteq S_f$ and $I_f \subseteq I_{e_f}$. When p opens f for both reading and writing, e_f must be safe, read/write, and must have labels equal to the file's. It is easy to show that p's file I/O to f is safe under these initial conditions (Definition 3).

A process p must hold such an endpoint e_f at least until it closes the corresponding file. Moreover, the labels on e_f are *immutable*: p cannot change them under any circumstances. Because the labels on e_f and f are immutable, and the initial conditions at file open enforced safety, all subsequent reads and writes to f across e_f are safe. This immutable endpoint preserves safety by restricting how the process can change its labels and capabilities. In Figure 1, say that file f_2 is open read/write and $S_{e_2} = S_{f_2} = \{x\}$. Then p cannot drop the y^- capability, since doing so would make e_2 unsafe. Similarly, p cannot add z to S_p despite its z^+ capability; it could only do so if it also owned z^-, which would preserve e_2's safety. Again, Flume can safely report any of these errors to p without inappropriately exposing information, since the error depends only on p's local state.

External Sources and Sinks Immutable endpoints also allow Flume to manage data sent into and out of the Flume system via network connections, user terminals and the like. If the system knows a process p to have access to resources that allow transmission or receipt of external messages (such as a network socket), it assigns p an immutable read/write endpoint $e_\perp$, with $S_{e_\perp} = I_{e_\perp} = \{\}$. Since $e_\perp$ must always be safe, it must always be the case that $S_p - D_p = I_p - D_p = \{\}$. That is, p has the privileges required import and export all of its data.

Similarly, if a process has communication channels not yet understood by the Flume reference monitor (e.g. System V IPC objects), then Flume simply assumes the process can expose information at any time and gives it an $e_\perp$ endpoint that cannot be removed until the resources are closed. This blunt restriction can be loosened as Flume's understanding of Unix resources improves.

Endpoints in Practice Endpoints help fill in the details of our earlier examples. For our secrecy example, Figure 2 shows how Bob uses a shell, *sh*, to launch his new (potentially evil) editor. Because *sh* can write data to Bob's terminal, it must have an $e_\perp$ endpoint, signifying its ability to export data out of the Flume system. Bob trusts this shell to export his data to the terminal and nowhere else, so he launches the shell with $b^- \in O_{sh}$. Now the shell can interact with the editor, even if the editor is viewing secret files. *sh* launches the editor process *ed* with secrecy $S_{ed} = \{b\}$ and without the b^- capability. The shell communicates with the editor via two pipes, one for reading and one for writing. Both endpoints in both processes have secrecy labels $\{b\}$, allowing reliable communication between the two processes. These endpoints are safe for the shell because $b^- \in O_{sh}$ and therefore $b \in D_{sh}$. *ed*'s endpoint labels match S_{ed} and are therefore also safe. Once the editor has launched, it opens Bob's secret file for reading and writing, acquiring an immutable endpoint e_5 with $S_{e_5} = \{b\}$. The file open does not change *ed*'s existing endpoints and therefore does not interrupt communication with the shell.

In the shared-secrecy calendar example, Bob launches a process q that is disconnected from his shell, and therefore has no $e_\perp$ endpoints. q can then set $S_q = \{a\}$ without affecting the safety of existing endpoints. Another implementation of the calendar service might involve a server process r that Alice and Bob both trust to work on their behalf. That is, r runs with a^- and b^- in its ownership set, and with secrecy $S_r = \{a, b\}$. By default, r can only write

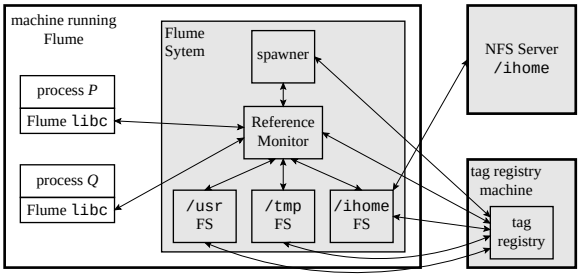

Figure 3: High-level design of the Flume implementation. The shaded boxes represent Flume's trusted computing base.

to processes or files that have both export protections. r can carve out an exception for communicating with Alice's or Bob's shell by creating endpoints with secrecy $\{a\}$ or $\{b\}$, respectively.

Similar examples hold for integrity protection and for processes that read from low-integrity sources.

5 The Flume Implementation

We present a user-space implementation of Flume for Unix, with some extensions for managing data for large numbers of users (as in Web sites). Flume's user space design is influenced by other Unix systems that build confinement in user space, such as Ostia [12] and Plash [29]. The advantages of a user space design are portability, ease of implementation, and in some sense correctness: Flume does not destabilize the kernel. The disadvantages are decreased performance and less access to kernel data structures, which in some cases makes the user-exposed semantics more restrictive than the DIFC rules require (e.g., immutable endpoints on files).

Flume's Linux implementation, like Ostia's, runs a small component in the kernel: a Linux Security Module (LSM) [36] implements Flume's system call interposition (see Section 5.2). The OpenBSD implementation of Flume uses the *systrace* system call [27] instead, but we focus on the Linux implementation in this description.

Figure 3 shows the major components of the Flume implementation. The *reference monitor* (RM) keeps track of each process's labels, authorizes or denies its requests to change labels and handles system calls on its behalf. The reference monitor relies on a suite of helpers: a dedicated spawner process (see Section 5.2), a remote tag registry (see Section 6.3), and user space file servers (see Section 6.7). The Flume-aware C library redirects Unix system calls to the RM and also supports the new Flume calls shown in Figure 4. Other machines running Flume can connect to the same tag registry and therefore can share the same underlying file systems (e.g., `ihome`) over NFS.

5.1 Confined and Unconfined Processes

To the reference monitor, all processes other than the helpers are potential actors in the DIFC system. A process can use the Flume system by communicating with the reference monitor via RPCs sent over a *control socket*. For convenience, a C library, which can be linked either statically or dynamically, translates many system calls into the relevant RPCs. The system calls that return file descriptors (e.g., `open`) use file-descriptor passing over the control socket. A process can have multiple control sockets to help with multi-threading.

Processes on a system running Flume are either *confined* or *unconfined*. By default, processes are unconfined and have empty labels and empty non-global ownership (i.e., $O_p - \mathbf{O} = \{\}$). The RM assigns an unconfined process an immutable endpoint $e_\perp$ with labels $I_{e_\perp} = S_{e_\perp} = \{\}$, reflecting a conservative assumption that the process may have network connections to remote hosts, open writable files, or an open user terminal (see Section 4.2).

- `label get_label({S,I})`
  Return the current process's S or I label.
- `capset get_ownership()`
  For the current process p, return capability set $O_p - \mathbf{O}$.
- `int change_label({S,I}, label l)`
  Set current process's S or I label to l, so long as the change is safe (Definition 2) and the change keeps all endpoints safe (Definition 4). Return an error code on failure.
- `int reduce_ownership(capset` O'`)`
  Reduce the calling process's ownership to $\mathbf{O} \cup O'$. Succeed if the new ownership keeps all endpoints safe and is a subset of the old.
- `label get_fd_label({S,I}, int fd)`
  Get the S or I label on file descriptor `fd`'s endpoint.
- `int change_fd_label({S,I}, int fd, label l)`
  Set the S or I label on `fd`'s endpoint to the given label. Return an error code if the change would violate the endpoint (Definition 4), or if the endpoint is immutable. Still succeed even if the change stops endpoint flows (in the sense of Definition 5).
- `tag create_tag({EP,IP,RP})`
  Create a new tag t for the specified security policy (export, integrity or read protection). In the first case add t^+ to $\mathbf{O}$; in the second add t^- to $\mathbf{O}$; and in the third add neither.
- `int flume_pipe(int *fd, token *t)`
  Make a new Flume pipe, returning a file descriptor and a pipe token.
- `int claim_fd_by_token(token t)`
  Exchange the specified token for its corresponding file descriptor.
- `pid spawn(char *argv[], char *env[], token pipes[], [label` S`, label` I`, capset` O`])`
  Spawn a new process with the given command line and environment. Collect given pipes. By default, set secrecy, integrity and ownership to that of the caller. If S, I and O are supplied and represent a permissible setting, set labels to S, I, and ownership set to O.

Figure 4: A partial list of new API calls in Flume.

An unconfined process conforms to regular Unix access control checks. If an unconfined process so desires, it can issue standard system calls (like `open`) that circumvent the Flume RM. In other words, the underlying Linux system dictates the security plan for unconfined processes.

5.2 Confinement and `spawn`

Confined processes are those for which the reference monitor carefully controls starting conditions and system calls. For any confined process p, the reference monitor installs a system call interposition policy (via LSM) that prevents p from directly issuing most system calls, especially those that yield resources outside of Flume's purview. In this context, system calls fit three categories: (1) *direct*, those that p can issue directly as if it were running outside of Flume; (2) *forwarded*, those that the LSM forbids p from making directly, but the RM performs on p's on behalf; and (3) *forbidden*, which are denied via LSM and not handled by the RM. Figure 5 provides a detailed list of which calls fall into which categories. The goal here is for the RM to maintain a complete understanding of p's resources. A confined process like p trades the restrictions implied by $e_\perp$ for a more restrictive system call interface.

Flume's `spawn` operation is the only way that new confined processes come into existence. Confined and unconfined processes can call `spawn` to make a new confined process, but confined processes may not fork. `spawn` combines the Unix operations of `fork` and `exec`, to create a new process running the supplied command. When a process p spawns a new confined process q, q's labels default to p's, but q starts without any file descriptors or endpoints. q

accumulates endpoints as a result of making new pipes and sockets or opening files (see Section 6.1). System call interposition blocks other resource-granting system calls.

Without `fork`, confined processes cannot use the Unix convention of sharing pipes or socketpairs with new children. Instead, Flume offers `flume_pipe` and `flume_socketpair`, which take the same arguments as their Unix equivalents, but both return a single file descriptor and a random 64-bit "pipe token." Once a process p receives this pair, it typically communicates the pipe token to another process q (perhaps across a call to `spawn`). q then makes a call to the reference monitor, supplying the pipe token as an argument, and getting back a file descriptor in return, which is the other logical end of the pipe (or socketpair) that the reference monitor gave to p.

The `spawn` operation takes up to six arguments: the command line to execute, an initial environment setting, an array of pipe tokens, and optional labels. The new process's labels are copied from the process that called `spawn`, unless S, I, O are specified. If the creator could change to the specified S, I, O labels, then those labels are applied instead. The only file descriptors initially available to the new process are a control socket and file descriptors obtained by claiming the array of pipe tokens. The new process is not the Unix child of the creating process, but the creator receives a random, unguessable token that uniquely identifies the new process (see below for a rationale). Labels permitting, the creator can wait for the new process or send it a signal, via forwarded versions of `wait` and `kill`.

The reference monitor forwards `spawn` requests to a dedicated spawner process. The spawner first calls `fork`. In the child process, the spawner (1) enables the Flume LSM policy; (2) performs any setlabel label manipulations if the file to execute is setlabel (see Section 6.5); (3) opens the requested executable (e.g. `foo.sh`), interpreter (e.g. `/bin/sh`) and dynamic linker (e.g., `/lib/ld.so`) via standard Flume `open` calls, invoking all of Flume's permission checks; (4) closes all open file descriptors except for its control socket and those opened in the previous step; (5) claims any file descriptors by token; and (6) calls `exec`.

The Flume LSM policy disallows all *direct* access to file systems by confined processes with a notable exception. When the child calls `exec` in Step (6), the LSM allows access to directories (used during path lookups in the kernel) and access to the binaries and scripts needed by `exec`, so long as they were opened during Step (3). Once the `exec` operation completes, the LSM closes the loophole, and rejects all future file system accesses.

The Flume LSM policy also disallows `getpid`, `getppid`, and friends. Because Linux allocates PIDs sequentially, two confined processes could alternatively exhaust and query the Linux PID space to leak information. Thus, Flume issues its own PIDs (chosen randomly from a sparse space) and hides Linux PIDs from confined processes. The standard LSM framework distributed with Linux does not interpose on `getpid` and friends, but Flume's small kernel patch adds LSM hooks that can disable those calls. Flume still works without the patch but allows confined processes to leak data through PIDs.

Confined processes are always run as an unprivileged user (e.g. `nobody`). If an adversary were to take over a confined process, it could issue only those system calls allowed by the Flume LSM policy. All other system interaction happens through the reference monitor and is subject to Flume's restrictions.

5.3 IPC In Flume

When p and q establish communication as a result of pipe token exchange, the file descriptors held by p and q actually lead to the reference monitor, which passes data back and forth between the two processes. The reference monitor proxies so it can interrupt communication if either process changes its labels in a way that would make endpoint information flow unsafe.

Direct	Forwarded
`clock_gettime`, `close`(file), `dup`, `dup2`, `exit`, `fchmod`, `fstat`, `getgid`, `getuid`, `getsockopt`, `lseek`, `mmap`, `pipe`, `poll`, `read`, `readv`, `recvmsg`, `select`, `sendmsg`, `setsockopt`, `setgid`, `sigprocmask`, `socketpair`, `write`, `writev` ...	`access`, `bind`(Unix-domain socket), `chdir`, `close`(socket), `getcwd`, `getpid`, `kill`, `link`, `lstat`, `mkdir`, `open`, `symlink`, `readlink`, `rmdir`, `spawn`†, `stat`, `unlink`, `utimes`, `wait` ...
	Forbidden
	`bind`(network socket), `execve`, `fork`, `getsid`*, `getpgrp`*, `getpgid`*, `getppid`*, `ptrace`, `setuid` ...

Figure 5: System calls available to confined processes in Flume. Those marked with "*" could be forwarded with better reference monitor support. Those marked with "†" are specific to Flume.

Consider two processes p and q connected by a pipe or socket where the relevant endpoint labels are the same as the process labels. If $S_p = S_q$ and $I_p = I_q$, data is free to flow in both directions, and communication is reliable as in standard Unix. That is, if p is writing faster than q can read, then the reference monitor will buffer up to a fixed number of bytes, but then will stop reading from p, eventually blocking p's ability to write. If $S_q \subsetneq S_p$ or $I_p \subsetneq I_q$, data cannot flow from q to p. Communication becomes one-way in the IFC sense and is no longer reliable in the Unix sense. The reference monitor will deliver messages from p to q, as before, but will always be willing to read from p, regardless of whether q exited or stopped reading. As the reference monitor reads from p without the ability to write to q (perhaps because q stopped reading), it buffers the data in a fixed-size queue but silently drops all overflow. Conversely, all data flowing from q to p (including an EOF marker) is hidden from p. The reference monitor buffers this data at first, then drops it once its queue overflows. If p or q changes its labels so that $S_p = S_q$ and $I_p = I_q$, then the reference monitor flushes all buffered data and EOF markers.

Spawned Flume processes can also establish and connect to Unix domain sockets. Creating a socket file is akin to creating a file and keeping it open for writing and follows the same rules (see the next section). Connecting to a Unix domain socket is akin to opening that file for reading. Assuming a client and server are permitted to connect, they receive new file descriptors and communicate with the proxy mechanism described above.

5.4 Implementation Complexity and TCB

The RM, spawner, file servers, and tag registry are all part of Flume's trusted computing base. We implemented them in C++ using the Tame event system [19]. Not counting comments and blank lines, the RM is approximately 14,000 LOC, the spawner about 1,000 LOC, the file server 2,500 LOC, and the tag registry about 3,500 LOC. The Flume LSM is about 500 LOC; the patch to the LSM framework for `getpid` and the like is less than 100 lines. Totaling these counts, we see Flume's total TCB (incremental to Linux kernel and user space) is about 21,500 LOC.

Flume's version of `libc`, the dynamic linker and various client libraries (like those for Python) are not part of the trusted computing base and can have bugs without compromising security guarantees. These libraries number about 6,000 lines of C code and 1,000 lines of Python, again not counting comments and empty lines.

6 PERSISTENCE IN FLUME

The Flume system aims to provide file system semantics that approximate those of Unix, while obeying DIFC constraints. Flume must apply endpoints to opened files to prevent data flows through the file system that are against DIFC rules. It also must enforce a naming scheme for files in a standard directory hierarchy that does not allow inappropriate release of information. Additionally, Flume must solve problems specific to DIFC, such as persistent storage and management of capabilities.

6.1 Files and Endpoints

To get Unix-like semantics, a process under Flume (whether confined or not) must have direct access to the Unix file descriptor for any file it opens, in case it needs to call `mmap` on that descriptor. Thus, the RM performs `open` on a process's behalf and sends it the resulting file descriptor. The process receives an immutable endpoint along with the descriptor.

File opens work as described in Section 4.2, with two additional restrictions in the case of writing. First, Flume assigns read/write endpoints to all writable file descriptors. A writer can learn information about a file's size by observing `write`'s or `lseek`'s return codes, and hence can "read" the file. The read/write endpoint captures the conservative assumption (as in HiStar) that writing always implies reading. Second, a file f has a *write-protect set* W_f in addition to its immutable labels S_f and I_f. A process p can only write to object f if it owns at least one capability in W_f (i.e., $O_p \cap W_f \neq \{\}$). This mechanism allows write protection of files in a manner similar to Unix's; only programs with the correct credentials (capabilities) can write files with non-empty W_f sets. By convention, a *write-protect tag* is the same as an integrity-protect tag: $t^- \in \mathbf{O}$, and t^+ is closely guarded. But t does not appear in I or S labels; only the capability t^+ has any use. The presence of t^+ in W_f yields the policy that processes must own t^+ to write f.

File closes use the standard Linux `close`. The reference monitor does not "understand" a process's internals well enough to know if a file is closed with certainty. Better LSM support can fix this shortcoming, but for now, Flume makes the conservative assumption that once a process has opened a file, it remains open until the process exits.

6.2 File Metadata

While Section 3.3 explains how file contents fit into Flume's DIFC, information can also flow through meta-data: file names, file attributes, and file labels. Flume does not maintain explicit labels for these items. Instead, Flume uses a directory's label to control access to the names and labels of files inside the directory, and a file's label to control access to the file's other attributes (such as length and modification time). Flume considers that a path lookup involves the process reading the contents of the directories in the path. Flume applies its information flow rules to this implicitly labeled data, with the following implications for applications.

A directory can contain secret files and yet still be readable, since the directory's label can be less restrictive than the labels of the files it contains. Typically the root directory has an empty S label and directories become more secret as one goes down. Integrity labels typically start out at $\mathcal{T}$ at the root directory and are non-increasing as one descends, so that the path name to a high-integrity file has at least as high integrity as the file.

The file system's increasing secrecy with depth means a process commonly stores secret files under a directory that is less secret. The Flume label rules prevent a process from creating a file in a directory that is less secret than the process, since that would leak information through the file's name and existence. Instead, the process can "pre-create" the files and subdirectories it needs early in its life, before it has raised its S label and read any private data. First, the process creates empty files with restrictive file labels. The process can then raise its S label, read private data, and write output to its files.

If a process p with labels S_p and I_p wants to spontaneously create a file f with the same labels, without pre-creating them, Flume offers a special leak-proof, high-integrity file namespace. p can create a directory of the form `/ihome/`$\mathsf{srl}(I_p)$`.`$\mathsf{srl}(S_p)$, where $\mathsf{srl}(L)$ is a serialized representation of label L. This directory has integrity level I_p and secrecy level S_p. Within that directory, the regular file system rules apply. Processes cannot directly open or read the `/ihome` directory, though they can traverse it on the way to opening files contained therein.

6.3 Persistent Privileges

In addition to supporting legacy Unix-like semantics, Flume provides persistence for capabilities and file labels. A process acquires capabilities when it creates new tags but loses those capabilities when it exits. In some cases, this loss of capabilities renders data permanently unreadable or unwritable (in the case of integrity). Consider a user u storing export-protected data on the server. A process acting on u's behalf can create export-protect tag t_u and write a file f with $S_f = \{t_u\}$, but if ${t_u}^-$ evaporates when the process exits, the file becomes inaccessible to all processes on the system, including those who speak for u.

Flume has a simple mechanism for reusing capabilities like ${t_u}^-$ across processes, reboots, and multiple machines in a server cluster. First, Flume includes a "central tag registry" that helps applications give long-term meaning to tags and capabilities. It can act as a cluster-wide service for large installations, and is trusted by all machines in the cluster. The tag registry maintains three persistent databases: one that maps "login tokens" to capabilities, one that remembers the meanings of capability groups, and a third database for extended file attributes (see Section 6.7).

A login token is an opaque byte string, possession of which entitles the holding process to a particular capability. A process that owns a capability c can ask its RM to give it a login token for c. On such a request, the RM asks the tag registry to create the token; the tag registry records the token and c in a persistent database. A process that knows a token can ask its RM to give it ownership of the corresponding capability. The operation succeeds if the RM can find the token and corresponding capability in the registry.

When creating new tokens, the tag registry chooses tokens randomly from a large space so that they are difficult to forge. It also can attach a timeout to each token, useful when making browser cookies good for one Web session only.

6.4 Groups

Some trusted servers keep many persistent capabilities and could benefit from a simpler management mechanism than keeping a separate login token for each capability. For example, consider a "finger server" that users trust to declassify and make public portions of their otherwise private data. Each user u protecting data with export-protect tag t_u must grant ${t_u}^-$ to the finger server.

Instead of directly collecting these capabilities (every time it starts up), the finger server owns a group G containing the capabilities it uses for declassification. Owning a capability for G implies owning all capabilities contained in G. When a new user v is added to the system, v can add ${t_v}^-$ to G, instantly allowing the finger server to declassify v's files. Groups can also contain group capabilities, meaning the group structure forms a directed graph.

Like any other capability, group capabilities are transferable, and can be made persistent with the scheme described in Section 6.3.

Capability groups are a scalability and programmability advance over previous DIFC proposals. In practice, secrecy and integrity labels stay small (less than 5 tags), and capability groups allow ownership sets to stay small, too. All group information is stored in the central tag registry, so that multiple machines in a cluster can agree on which capabilities a group contains. Reference monitors contact the tag registry when performing label changes. Since groups could grow to contain many capabilities, a reference monitor does not need to download the entire group membership when checking label change validity. Instead, it performs queries of the form "is capability *c* a member of group *g*," and the registry can reply "yes," "no" or "maybe, check these subgroups." In our experience, groups graphs form squat, bushy trees, and the described protocol is efficient and amenable to caching.

Finally, so that the groups themselves do not leak information, Flume models groups as objects, like files on the file system. When created, a group takes on immutable labels for secrecy and integrity, and also (at the creator's discretion) a write-protect capability set. Processes modifying a group's membership must be able to write to the group object (currently, only addition is supported). Processes using groups in their label change operations are effectively reading the groups; therefore, processes can only use a group capability in their ownership sets if they can observe the group object.

6.5 Setlabel

Flume provides a *setlabel* facility, analogous to Unix's *setuid* or HiStar's gates, that is the best way to launch a declassifier. Setlabel tightly couples a persistent capability with a program that is allowed to exercise it. A setlabel file contains a login token and a command to execute. Flume never allows a setlabel file to be read, to prevent release of the login token. Instead, the file's S and I labels limit which processes can execute the file. Any process whose S and I would allow it to read the setlabel file may ask the reference monitor to spawn the file. The reference monitor executes the command given in the file and grants the process the capability referred to by the login token. Typically a setlabel program will exercise the capability to read data that the parent is not allowed to export, and then declassify it to the parent.

Setlabel files can also specify a minimum integrity label and a maximum secrecy label, which executing processes must conform to. The minimum integrity label helps defend the setlabel process from surprises in its environment. The maximum secrecy label helps a setlabel program limit the types of secrets it declassifies.

6.6 Privileged Filters

Finally, in the application we've built, we have found a need for automatic endorsement and declassification of files; see Section 7.6 for a detailed motivation. A process can create a *filter* to replace "find label" (L_{find}) with a "replace label" (L_{repl}) if it owns the privileges to add all tags in $L_{\text{repl}} - L_{\text{find}}$ and to subtract all tags in $L_{\text{find}} - L_{\text{repl}}$. The filter appears as a file in the file system, similar to a setlabel file. Any other process p that can read this file can activate this filter. After activation, whenever p tries to open a file for reading whose file label contains all the tags in L_{find}, Flume replaces those tags with L_{repl} before it decides whether to allow the process to open the file. A process can activate multiple filters, composing their effects.

6.7 File System Implementation

The reference monitor runs a suite of user-space file server processes, each responsible for file system operations on a partition of the namespace. The reference monitor forwards requests such as `open` and `mkdir` to the appropriate file server. To reduce the damage in case the file server code has bugs, each server runs as a distinct non-root user and is `chroot`ed into the part of the underlying file system that it is using. The usual Unix access-control policies hide the underlying file system from unprivileged processes outside of Flume.

Each file server process store files and directories one-for-one in an underlying conventional file system. It stores labels in the extended attributes of each underlying file and directory. To help larger labels fit into small extended attributes, the tag registry provides a service that generates small persistent nicknames for labels. Flume file servers can also present entire underlying read-only file systems (such as `/usr`) as-is to Flume-confined software, applying a single label to all files contained therein. The Flume system administrator determines this configuration.

Since Linux's NFS client implementation does not support extended attributes, Flume supports an alternate plan when running over an NFS-mounted file system. In this case, Flume stores persistent label nicknames as 60-bit integers, split across the user and group ID fields of a file's metadata. The fake UID/GID pairs written to the file system are in the range $[2^{30}, 2^{31})$, avoiding UIDs and GIDs already in use. This approach unfortunately requires the file server to run as root, for access to the `fchown` call.

Simultaneous use of the same underlying file system by multiple Flume file server processes might result in lack of atomicity for label checks and dependent operations. For example, checking that file creation is allowed in a directory and actually creating the file should be atomic. Race conditions might arise when a cluster of hosts share an NFS file system. Flume ensures the necessary atomicity by operating on file descriptors rather than full path names, using system calls such as Linux's `openat`.

The DIFC rules require that a process must read all directories in any path name it uses. One approach is to laboriously check each directory in a given path name. In practice, however, applications arrange their directory hierarchies so that secrecy increases and integrity decreases as one descends. The Flume implementation enforces this ordering, with no practical loss of generality. Flume can thus optimize the path check: if a process can read a file f, it must also be able to read all of f's ancestors, so there is no need to check. If the file does not exist or the process cannot read it, Flume reverts to checking each path component, returning an error when it first encounters a component that does not exist or cannot be read.

At present, Flume supports most but not all of Unix's semantics. The current implementation allows renames and creation of hard links only within the same directory as the original file. And Flume implements the per-process working directory by remembering a path name per process, which will deviate from Unix behavior if directories are renamed.

Flume's file system has shortcomings in terms of security. An unconfined process with Unix super-user privileges can use the underlying file system directly, circumventing all of Flume's protections. This freedom can be a valuable aid for system administrators, as well as an opportunity for attackers. Also, Flume does not avoid covert channels related to storage exhaustion and disk quotas. A solution would require deeper kernel integration (as in HiStar).

7 APPLICATION

This section explores Flume's ability to enhance the security of off-the-shelf software. We first describe MoinMoin [22], a popular Web publishing system with its own security policies. We then describe FlumeWiki, a system that is derived from Moin but enforces the Moin's policies with Flume's DIFC mechanisms. FlumeWiki goes

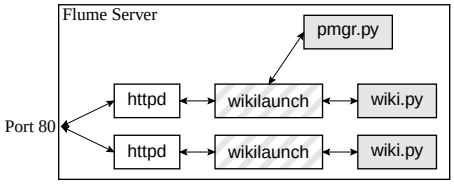

Figure 6: FlumeWiki application overview, showing two of many process pipelines. The top request is during a session login; the bottom request is for a subsequent logged-in request. Flume-oblivious processes are unshaded, unconfined processes are striped, and confined processes are shaded.

further, adding a new security policy that offers end-to-end integrity protection against buggy MoinMoin plug-ins. The resulting system substantially reduces the amount of trusted application code.

7.1 MoinMoin Wiki

MoinMoin is a popular Python-based Web publishing system (i.e., "wiki") that allows Web clients to read and modify server-hosted pages. Moin is designed to share documents between users, but each page can have an access control list (ACL) that governs which users and groups can access or modify it. For example, if a company's engineering document is only meant to be read by the engineers and their program manager Alice, the document would have the read ACL (alice, engineers), where "alice" is an individual and "engineers" is a group containing all the engineers.

Unfortunately, Moin's ACL mechanism has been a source of security problems. Moin comprises over 91,000 lines of code in 349 modules. It checks read ACLs in 41 places across 22 different modules and write ACLs in 19 places across 12 different modules. The danger is that an ACL check could have easily been omitted. Indeed, a public vulnerability database [26] and MoinMoin's internal bug tracker [25] show at least five recent ACL-bypass vulnerabilities. (We do not address cross-site scripting attacks, also mentioned in both forums.) In addition to ACL bugs, any bug in Moin's large codebase that exposes a remote exploit could be used to leak private data or tamper with the site's data.

Moin also supports plug-ins, for instance "skins" that change the way it renders pages in HTML. Site administrators download plug-ins and install them site-wide, but buggy or malicious plug-ins can introduce further security problems. Plug-ins can violate Moin's ACL policies. They also can wittingly or unwittingly misrender a page, confusing users with incorrect output.

7.2 Fluming MoinMoin

Flume's approach for enhancing Moin's read and write protection is to factor out security code into a small, isolated security module, and leave the rest of Moin largely unchanged. The security module needs to configure only a Flume DIFC policy and then run Moin according to that policy. This division of labor substantially reduces the amount of trusted code and the potential for security-violating bugs. In addition, the security module can impose end-to-end integrity by forcing the untrusted portion to run with a non-empty integrity label, yielding guarantees of the form: "no plug-ins touched the data on this page at any time" or "vendor v's plug-in touched this data but no other plug-ins did."

7.3 FlumeWiki Overview

Figure 6 illustrates the four main components of the FlumeWiki system. FlumeWiki uses an unmodified Apache Web server (httpd) for the front-end request handling. wiki.py is the bulk of the application code, consisting of mostly unmodified MoinMoin code.

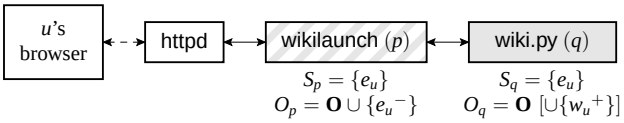

Figure 7: Label setup for a *read* or *write* request in FlumeWiki. wiki.py only gets capability w_u^+ if writing. The target page is export- and write-protected by user u.

pmgr.py is a small trusted program that manages usernames and passwords; it runs as a setlabel program so that it may compare submitted passwords against read-protected hashes on the server. wikilaunch is the small trusted security module; it is responsible for interpreting the Web request, launching wiki.py with the correct DIFC policy and proxying wikilaunch's response back to Apache. Because it communicates with resources outside of Flume (i.e., httpd), it is unconfined and has an $e_\perp$ endpoint.

When a typical HTTP request enters the system it contains the client's username u and an authentication token. httpd receives the request and launches wikilaunch as a CGI process. wikilaunch requests u's capabilities from the RM using the authentication token. It then sets up a DIFC policy by spawning wiki.py with appropriate S, I and O. wiki.py renders the page's HTML, sends it to wikilaunch over a pipe and exits. wikilaunch forwards the HTML back to httpd which finally sends it back to u's browser. wiki.py's S label prevents it from exporting data without the help of wikilaunch.

7.4 Principals, Tags and Capabilities

FlumeWiki enforces security at the level of principals, which may be users or ACL-groups (which are groups of users). Each principal x has an export-protect tag e_x and a write-protect tag w_x. Principal x also has a capability group $G_x = \{e_x^-, w_x^+\}$.

If user u is a member of ACL-group g with read-write privileges, her capability group G_u also contains G_g which allows her to read and modify g's private and write-protected data. If user u is a member of g with read-only privileges, her capability group G_u instead contains $G_g^{ro} = \{e_g^-\}$ which provides enough capabilities to read and export g's private data but not modify it.

Each Web page on a FlumeWiki site may be export-protected and/or write-protected. Export-protected pages have the secrecy label $S = e_x$ where x is the principal allowed to read and export it. x's write-protected pages have the write-protect capability set $W = \{w_x^+\}$.

7.5 Export- and Write-Protection Policies

wikilaunch handles requests that read pages differently from those that write. If u's request is for a read, and u has at least read access for groups $g_1, \ldots, g_n$, then wikilaunch spawns a new wiki.py process q with $S_q = \{e_u, e_{g_1}, \ldots, e_{g_n}\}$ and $O_q = \mathbf{O}$, allowing the standard MoinMoin code in FlumeWiki transparent read access to files the user is allowed to read (see Figure 7). For a request that involves creating or modifying a page, wikilaunch looks at the directory d in which the page resides. If d is protected by an export-protect tag e_x, wikilaunch sets wiki.py's $S = \{e_x\}$. If d is also protected by a write-protect tag w_x, wikilaunch sets wiki.py's $W = \{w_x^+\}$ (also shown in Figure 7). If the user u is not authorized to perform the requested action, wikilaunch will fail when trying to spawn wiki.py and notify the user of their transgression. Finally, wikilaunch sets its secrecy label equal to that of wiki.py so that they may share bidirectional pipe communication.

This DIFC policy provides three security properties. First, wikilaunch's S label ensures that only data the logged-in user is allowed to see can flow from wiki.py to the browser. Second, any other form

of output produced by wiki.py (for example a file) will also have a label containing e_u or some e_g so that other users' wikilaunch or wiki.py processes cannot reveal that output (since they lack e_u^- or e_g^-). Third, it provides discretionary write control: only processes that own w_x^+ can overwrite x's files.

7.6 End-to-End Integrity

In addition to read and write protection policies, FlumeWiki can optionally use Flume's integrity mechanisms to guard against accidental execution of untrusted dynamically-linked libraries or Python libraries like Moin plug-ins. The code that a Python program will execute is difficult to predict and thus difficult to inspect statically, since it depends on settings such as `LD_LIBRARY_PATH`, Python's class search path, and other run-time decisions.

FlumeWiki enforces an integrity constraint on the code that produced each page and then makes that integrity value visible to users. By default, only code in the base FlumeWiki distribution is allowed to be involved in displaying a page. However, if a page has a name like $v.f$, where v is the name of a third party vendor, then FlumeWiki also allows vendor v's software to participate in generating the page.

The default integrity policy operates as follows. During installation, all files in the distribution get $I = \{i_w\}$, where i_w represents the integrity of the base distribution. wikilaunch starts wiki.py with $I = \{i_w\}$, which guarantees that the program will never read any file (including dynamically-loaded program text) with an integrity label that doesn't contain i_w. wikilaunch sets its own label to $I = \{i_w\}$. Then, if wiki.py drops its integrity to $I = \{\}$, wikilaunch will be unable to receive its responses. This arrangement means that all properly created wiki documents have $I = \{i_w\}$, which indicates that they were created with the base distribution alone. In this manner, a user u gets an end-to-end integrity guarantee: all code involved with collecting u's input, writing u's data to disk, retrieving the data, formatting the data, and outputting the data had i_w in its label and therefore involved only the base FlumeWiki software.

For pages that allow the use of plug-in code, wikilaunch launches wiki.py with $I = \{i_v\}$ to allow v's plug-in code to participate in the page's rendering. However, the plug-in relies on FlumeWiki code during processing, which it cannot read off the disk: FlumeWiki's code does not have i_v in its integrity label. For wiki.py to read FlumeWiki's code, it would need to reduce its integrity label to $I = \{\}$, ruling out all future hopes of regaining non-empty integrity and outputting to wikilaunch. Filters (see Section 6.6) provide the solution.

The site administrator who installs v's plug-in owns the capability i_v^+, and thus can create an integrity filter that replaces labels of the form $I = \{i_w\}$ with $\{i_w, i_v\}$. This filter implements the idea that vendor v's code trusts FlumeWiki code. With this filter in place, wikilaunch can set wiki.py's and its own integrity labels to $I = \{i_v\}$, thus gaining assurance that any data returned was only touched by vendor v's and FlumeWiki's code.

7.7 Discussion

Adapting Moin to Flume required roughly 1,000 lines of new C++ code for wikilaunch, and modifications to about 1,000 out of Moin's 91,000 lines of Python. We did not modify or even recompile Apache or the Python interpreter, even though Python is spawned by Flume. The changes to Moin were in its login procedure, access control lists, and file handling, which we modified to observe and manipulate DIFC controls (like process labels and endpoint labels). Most of these changes are not user-visible. Though wrapper programs like wikilaunch could be expressed in other DIFC systems like Asbestos or HiStar, the integration within Moin would be difficult without an application-level API like the one presented here.

An advantage of the DIFC approach is that we did not need to understand all of Moin's code. Because wiki.py always runs within Flume's confines, we need only understand wikilaunch to grasp FlumeWiki's security policy. wikilaunch is small, and auditing it gave us confidence in the overall security of FlumeWiki, despite any bugs that may exist in the original Moin code or that we may have introduced while adapting the code.

Time did not permit the adaptation of all MoinMoin's features, such as internationalization, indexing, and hit counters. To Flume, these features attempt to leak data through shared files, so they fail with Flume permission errors. FlumeWiki could reenable them with specialized declassifiers.

8 EVALUATION

In evaluating Flume and FlumeWiki we consider whether they improve system security, how much of a performance penalty they impose and whether Flume's scaling mechanisms are effective.

For security, we find that Flume prevents ACL vulnerabilities and even helps discover new vulnerabilities. For performance, we find that Flume adds from 35–286μs of overhead to interposed system calls, which is significant. However, at the system level, the throughput and latency of FlumeWiki is within 45% and 35% of the unmodified MoinMoin wiki, respectively, and Flume's clustering ability enables FlumeWiki to scale beyond a single machine as Web applications commonly do.

8.1 Security

The most important evaluation criterion for Flume is whether it improves the security of existing systems. Of the five recent ACL bypass vulnerabilities [25, 26], three are present in the MoinMoin version (1.5.6) we forked to create FlumeWiki. One of these vulnerabilities is in a feature disabled in FlumeWiki. The other two were discovered in code FlumeWiki indeed inherits from Moin. We verified that FlumeWiki still "implements" Moin's original buggy behavior and that the Flume security architecture prevents these bugs from revealing private data.

To make FlumeWiki function in the first place, we had to identify and solve a previously undocumented vulnerability in Moin. The original Moin leaks data through its global namespace. For instance, a user Bob can prove that the secret document `ReasonsToFireBob` exists by trying and failing to create the document himself. By contrast, Flume's IFC rules forced FlumeWiki to be built in a way that doesn't leak information through its namespace.

8.2 Interposition Overhead

To evaluate the performance overhead when Flume interposes on system calls, we measured the system call latencies shown in Figure 8. In all of these experiments, the server running Linux version 2.6.17 with and without Flume is a dual CPU, dual-core 2.3GHz Xeon 5140 with 4GB of memory. The Web server is Apache 1.3.34 running MoinMoin and FlumeWiki as frozen Python CGI programs. The Web load generator is a 3GHz Xeon with 2GB of memory running FreeBSD 5.4.

For most system calls, Flume adds 35–286μs per system call which results in latency overhead of a factor of 4–35. The Flume overhead includes additional IPC, RPC marshalling, additional system calls for extended attributes and extra computation for security checks. The additional cost of IPC and RPC marshalling is shown by the `flumenull` latency, which reports the latency for a no-op RPC call into the reference monitor (RM). Most Flume system calls

Operation	Linux	Flume	diff.	mult.
`mkdir`	86.0	371.1	285.2	4.3
`rmdir`	13.8	106.8	93.0	7.7
`open`				
— create	12.5	200.2	187.7	16.0
— exists	3.2	110.3	107.1	34.5
— exists, inlined	3.3	41.0	37.7	12.5
— does not exist	4.3	101.4	97.1	23.6
— does not exist, inlined	4.2	39.8	35.6	9.5
`stat`	2.8	98.1	95.3	34.5
— inlined	2.8	38.7	35.9	13.7
`close`	0.6	0.9	0.2	1.3
`unlink`	15.4	110.0	94.6	7.2
`symlink`	9.5	106.8	97.3	11.2
`readlink`	2.7	90.2	87.5	33.0
`create_tag`		22.6		
`change_label`		55.0		
`flumenull`		20.1		
IPC round trip latency	4.1	33.8	29.8	8.2
IPC bandwidth	2945	937	2008	3.1

Figure 8: System call and IPC microbenchmarks, and Flume overhead as a multiplier. Latencies are in μs and bandwidth is in MB/sec. System calls were repeated 10,000 times, IPC round trips were repeated one million times, and IPC bandwidth was measured over a 20GB transfer; these results are averages.

consist of two RPCs, one from the client application into the reference monitor and one from the reference monitor to a file server, so the RPC overhead accounts for approximately 40μs of Flume's additional latency. As an optimization on public file systems, the RM handles `open` and `stat` calls inline rather than querying a file server and thus avoids a second RPC. Calls like `create_tag` and `change_label` also use a single RPC into the RM and `close` for files does not contact the RM at all. For non-public file systems, `open` on a non-existent file requires the RM to walk down the file system to determine what error message to return to the client, so this operation is particularly expensive. This check is faster in a public file system (where all files are readable to everyone), because the RM need not walk the parent directories.

Flume also adds overhead to IPC communication because it proxies IPC between processes. The base case in our measurements is an IPC round trip: *p* writes to *q*, *q* reads, *q* writes to *p*, and then *p* reads. This exchange amounts to four system calls in total on standard Linux. The RM's proxying of IPC adds eight system calls to this exchange: four calls to `select`, two `read`s and two `write`s. Thus, an IPC round trip takes 12 system calls on Flume, incurring the three-fold performance penalty for additional system calls seen in IPC bandwidth. As with `flumenull` computation and context switching in Flume add additional latency overhead, summing to the eight-fold latency degradation seen in Figure 8.

8.3 Flume Overhead

To evaluate the system level performance overhead of Flume, we compare the throughput and latency of pages served by an unmodified MoinMoin wiki and by FlumeWiki.

In the read experiments, a load generator randomly requests pages from a pool of 200 wiki pages; the pages are approximately 9 KB each. In the write experiments, each write request contains a 40 byte modification to one of the pages for which the server responds with an 9 KB page. In all experiments, the request is from a wiki user, who is logged in using an HTTP cookie. For the latency results, we report the latency with a single concurrent client. For the throughput results, we adjusted the number of concurrent clients to maximize throughput. Figure 9 summarizes the results.

	Throughput (req/sec)		**Latency** (ms/req)	
	MoinMoin	FlumeWiki	MoinMoin	FlumeWiki
Read	33.2	18.8	117	156
Write	16.9	11.1	237	278

Figure 9: Latency and throughput for FlumeWiki and unmodified MoinMoin averaged over 10,000 requests.

FlumeWiki is 43% slower than MoinMoin in read throughput, 34% slower in write throughput and it adds a latency overhead of roughly 40ms. For both systems, the bottleneck is the CPU. MoinMoin spends most of its time interpreting Python and FlumeWiki has the additional system-call and IPC overhead of Flume.

Most of FlumeWiki's additional cost comes from calls to `open` and `stat` when Python is opening modules. For each page read request, the RM serves 753 system calls including 487 `open`s and 186 `stat`s. Of the calls to `open`, 18 are for existing non-public files, 73 are for existing public files, 16 are for non-existent non-public files and 380 are for non-existent public files. Of the `stat`s, 156 are for public files and 30 are for non-public files. These calls sum to 28ms of overhead per request, which accounts for much of the 39ms difference in read latency. FlumeWiki also incurs an extra `fork` and `exec` to spawn `wiki.py` as well as extra system calls on each request to setup labels, pipes and filters.

The numbers reported in Figure 9 reflect *frozen* Python packages, both in the case of FlumeWiki and MoinMoin. Frozen Python packages store many Python packages in one file, and in the case of FlumeWiki reduce the combined number of `open` and `stat` calls from more 1900 to fewer than 700. Frozen packages especially benefit FlumeWiki's performance, since its system call overhead is higher than standard Moin's.

8.4 Cluster Performance

Despite Flume's slowdown, FlumeWiki may be fast enough already for many small wiki applications. The Flume implementation could be optimized further, but Flume's support for a centralized tag registry and FS file sharing supports another strategy for improving performance, namely clustering. To investigate the scalability of the cluster mechanism, we ran the FlumeWiki read throughput experiment on the same server hardware, but with a varying number of single CPU virtual machines on top of a Linux-based virtual machine monitor. Each virtual machine is limited to a single hardware CPU, and within each virtual machine, we ran Flume on a guest Linux OS.

In this experiment, FlumeWiki stores shared data including pages and user profiles in an NFS file system and all other data is duplicated on each VM's private disk. The NFS file system and the tag registry are both served by the host machine. With a single VM (i.e., a 1-node cluster), throughput was 4.3 requests per second. Throughput scales linearly to an aggregate of 15.5 requests per second in the case of four VMs (i.e., a 4-node cluster), which is the maximum number of CPUs on our available hardware. This cluster configuration achieves lower throughput than the single-machine configuration because of VM and NFS overhead.

8.5 Discussion

Although FlumeWiki's cluster performance may already be suitable for some services, one direction for future performance improvements is to modify FlumeWiki to run as a FastCGI service which amortizes a CGI process's startup cost over multiple requests. Benchmarks posted on the MoinMoin site [31] show a tenfold performance improvement when running MoinMoin as a FastCGI application [9] rather than a standalone CGI (as in our

benchmarks) and FlumeWiki could benefit from a similar architecture. One approach is to emulate Asbestos's event processes: keep one Python instance running for each (S, I, O) combination of labels currently active, and route requests to instances based on labels.

9 Conclusion

Flume demonstrates that the advantages of DIFC can be brought to bear on standard operating systems and applications. Using Flume a programmer can provide strong security for Unix applications, even if parts of the application contain bugs. We hope that by simplifying DIFC and allowing it to coexist with legacy software, both in the kernel and at application level, Flume can expose a wide audience of developers to DIFC-style security policies and programming techniques.

Acknowledgments

The authors thank Barbara Liskov, Steve VanDeBogart, Mike Walfish, Nickolai Zeldovich, the anonymous reviewers and shepherd Andrew Myers for their comments on drafts of this paper. We thank the members of the Asbestos Project team for feedback and suggestions on the Flume label model and its implementation. We thank Mark Seaborn for his work on Plash, which helped us make `glibc` system call interposition work on Linux. Maxwell Krohn was awarded an SOSP student travel scholarship, supported by the National Science Foundation, to present this paper at the conference. This work was supported by the joint NSF Cybertrust/DARPA grant CNS-0430425, Nokia, Taiwan's Industrial Technology Research Institute (ITRI), and an NSF Graduate Student Fellowship.

References

[1] D. E. Bell and L. L. Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[2] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Rev. 1, MITRE Corp., Bedford, MA, 1976.

[3] M. Brodsky et al. Toward secure services from untrusted developers. Technical Report TR-2007-041, MIT CSAIL, Aug. 2007.

[4] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security*, Aug. 2007.

[5] C. Cowan et al. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proc. 11th USENIX Security*, Aug. 2002.

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

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. 2002 OSDI*, Dec. 2002.

[8] P. Efstathopoulos et al. Labels and event processes in the Asbestos operating system. In *Proc. 20th SOSP*, October 2005.

[9] FastCGI. Open Market. `http://www.fastcgi.com`.

[10] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proc. 2000 IEEE Security and Privacy*, May 2000.

[11] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proc. IEEE Security and Privacy*, 1999.

[12] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. 2004 NDSS*, February 2004.

[13] J. Gelinas. Virtual private servers and security contexts, Jan. 2003. `http://linux-vserver.org`.

[14] R. Goldberg. Architecture of virtual machines. In *1973 NCC AFIPS Conf. Proc.*, volume 42, pages 309–318, 1973.

[15] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *Proc. 22st ACSAC*, December 2006.

[16] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. 14th SOSP*, Dec. 1993.

[17] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd SANE*, May 2000.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. 11th USENIX Security*, Aug. 2002.

[19] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proc. 2007 USENIX*, June 2007.

[20] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. 2001 USENIX*, June 2001. FREENIX track.

[21] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, 1992.

[22] MoinMoin. The MoinMoin Wiki Engine, Dec. 2006. `http://moinmoin.wikiwikiweb.de/`.

[23] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 16th SOSP*, Oct. 1997.

[24] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[25] National Vulnerability Database. CVE-2007-2637. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-2637`.

[26] osvdb.org. Open Source Vulnerability Database. `http://osvdb.org/searchdb.php?base=moinmoin`.

[27] N. Provos. Improving host security with system call policies. In *Proc. 12th USENIX Security*, Aug. 2003.

[28] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sept. 1975.

[29] M. Seaborn. Plash: tools for practical least privilege. `http://plash.beasts.org`.

[30] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module, February 2006. `http://www.nsa.gov/selinux/papers/module-abs.cfm`.

[31] N. Soffer. MoinBenchmarks. `http://moinmoin.wikiwikiweb.de/MoinBenchmarks`.

[32] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *Proc. 2006 OSDI*, Nov. 2006.

[33] VMware. VMware and the National Security Agency team to build advanced secure computer systems, Jan. 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[34] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. 2003 USENIX*, June 2003.

[35] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 2002 OSDI*, Dec. 2002.

[36] 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*, Aug. 2002.

[37] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *Proc. 2007 NSDI*, Apr. 2007.

[38] N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th OSDI*, Nov. 2006.