# Information Flow Control for Secure Web Sites

by

Maxwell Norman Krohn

A.B., Harvard University (1999)
S.M., Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008



Author .....
Department of Electrical Engineering and Computer Science
August 29, 2008

Certified by .....
Frans Kaashoek
Professor
Thesis Supervisor

Certified by .....
Robert Morris
Professor
Thesis Supervisor

Certified by .....
Eddie Kohler
Associate Professor, UCLA
Thesis Supervisor

Accepted by .....
Professor Terry P. Orlando
Chair, Department Committee on Graduate Student

**Information Flow Control for Secure Web Sites**

by

Maxwell Norman Krohn

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

Sometimes Web sites fail in the worst ways. They can reveal private data that can never be retracted [60, 72, 78, 79]. Or they can succumb to vandalism, and subsequently show corrupt data to users [27]. Blame can fall on the off-the-shelf software that runs the site (e.g., the operating system, the application libraries, the Web server, etc.), but more frequently (as in the above references), the custom application code is the guilty party. Unfortunately, the custom code behind many Web sites is difficult to secure and audit, due to large and rapidly-changing trusted computing bases (TCBs).

A promising approach to reducing TCBs for Web sites is *decentralized information flow control* (DIFC) [21, 69, 113]. DIFC allows the split of a Web application into two types of components: those inside the TCB (trusted), and those without (untrusted). The untrusted components are large, change frequently, and do most of the computation. Even if buggy, they cannot move data contrary to security policy. Trusted components are much smaller, and configure the Web site's security policies. They need only change when the policy changes, and not when new features are introduced. Bugs in the trusted code can lead to compromise, but the trusted code is smaller and therefore easier to audit.

The drawback of DIFC, up to now, is that the approach requires a major shift in how programmers develop applications and thus remains inaccessible to programmers using today's proven programming abstractions. This thesis proposes a new DIFC system, *Flume*, that brings DIFC controls to the operating systems and programming languages in wide use today. Its key contributions are: (1) a simplified DIFC model with provable security guarantees; (2) a new primitive called *endpoints* that bridges the gap between the Flume DIFC model and standard operating systems interfaces; (3) an implementation at user-level on Linux; and (4) success in securing a popular preexisting Web application (MoinMoin Wiki).

Thesis Supervisor: Frans Kaashoek
Title: Professor

Thesis Supervisor: Robert Morris
Title: Professor

Thesis Supervisor: Eddie Kohler
Title: Associate Professor, UCLA

## Previously Published Material

Chapters 3 and 6 through 9 appeared as part of a previous publication [58]: Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

# Acknowledgments

As I finish up five years of graduate work at MIT, I begin to take stock of how indebted I am to so many, from people I see on a daily basis, to those who were profoundly influential during life's previous seasons. The list begins with the faculty advisors who have taken me under their wing. M. Frans Kaashoek is an inexhaustible fountain of enthusiasm, encouragement and optimism. I hope just some of his mastery of everything from grand vision to x86 minutia has rubbed off on me. But more than anything, his always sunny disposition was the support for my graduate work, the counterbalance to my histrionic proclivity toward doom and gloom. Not only did Frans figuratively push me up the mountain with financial, academic, and emotional backing, he literally did so in our strenuous[1] bike rides together. The cliché is that a graduate advisor acts *in loco parentis*, but in this case the cliché rings true in the best way. Thanks Frans for everything.

Robert Morris played the part of the skeptical outsider. It took me a while to realize this was a personae he assumed to make my work better, more focused, and more convincing to those who lacked our MIT/LCS/PDOS biases. By now, his lesson has started to sink in, and I will always hear a voice: how can you say this in a simpler way? how much mechanism is really required? what problem are you really solving? In a world easily won over by jargon and pseudo-intellectualism, Robert's language, approach, precision and insights are genuine. I thank Robert for the time and effort he has invested in my papers, my talks and me over the years, and hope I can instill the same values in my students.

Rounding out this team is Eddie Kohler. Eddie and I don't always see eye-to-eye on how best to present information, but given his community-wide reputation for amazing graphical design, I must be in the wrong. Eddie worked tirelessly with me (and in spite of me) to refine the paper versions of Chapters 3 and 6, improving their quality tremendously. Our collaboration on earlier projects, like OKWS, Asbestos and Tame, produced similar results: different styles, but a product that shined in the end. Eddie's dedication to improving my work, his curiosity to learn more about the ideas that interest me, and his coaching on the finer points of presentation have made me a better researcher and communicator. Thank you Eddie.

Some faculty in and around MIT were *not* my advisors, but they nonetheless advised me well. Butler Lampson, a thesis reader, gave this thesis a thorough and careful read. I sincerely thank him for his comments and insights, which have helped improve this document greatly. Srini Devadas, Arvind, and Jamie Hicks sat through my practice talks and internal MIT-Nokia presentations, giving me helpful commentary on content and style. Hari Balakrishnan and John Guttag sat through those talks *and* practice job talks, helping me to put my best foot forward on the job market.

Eran Tromer helped me understand non-interference, and he suggested the high-level Flume formal model. Chapters 4 and 5 bear his influence, and we plan to turn them into a coauthored paper. I thank Alex Yip for his deep contributions to the Flume, Asbestos and W5 projects, his tireless enthusiasm, his talents in hacking and writing, and his inexhaustible wisdom in matters of cycling, cycle repair, and general do-it-yourself. Thanks to Micah Brodsky for his contributions to Flume, W5 and many fascinating conversations. Thanks to Steve VanDeBogart and Petros Efstathopoulos—friends and colleagues at UCLA—who were coauthors on the Asbestos paper. They along with Nickolai Zeldovich offered advice and commentary on the Flume paper and talk. Thanks to Nickolai for also reviewing Chapters 4 and 5 of this thesis. Natan Cliffer

[1] For me, not for him.

contributed to Flume, Neha Narula to W5, and Cliff Frey and Dave Ziegler to Asbestos.

Material in this thesis has been published elsewhere, thus it benefits from careful readings and comments from reviewers and paper shepherds. I thank Andrew Myers, Emin Gün Sirer, and the anonymous reviewers from Usenix '04, HotOS '05, SOSP '05 and SOSP '07. I thank Nokia, ITRI, MIT, and the NSF for their generous financial support.

I graduated from the same high school, college and now graduate program one year behind Mike Walfish and have benefited outrageously from his having to figure things out on his own. Mike has advised me on: how to get into college, which classes to take, how to get into graduate school, how to write papers and talks, how to get a job, and everything in between. He's also been a patient, selfless and endlessly entertaining friend throughout. Mike, since this is the one page of text I've written in the last five years that you haven't been kind enough to proofread, I apologize in advance for any typos. Jeremy "Strib" Stribling and I showed up to graduate school on the same day, and he has been an inspiring friend and colleague ever since. I thank Strib for everything from reading drafts of my papers, to dropping numerous well-timed jokes. Dan Aguayo, he of perpetual `positive-outlook`, deserves credit for making me laugh as hard as physically possible. Props to Strib and Dan for being my coauthors on infinitely many papers regarding the KGB's underwater testbed of Commodore 64s.

Bryan Ford and Chris Lesniewski-Laas always went above and beyond in their commentary on drafts and practice talks, helping defend my work when my own attempts faltered. Thanks to Russ Cox and Emil Sit for their management of PDOS computer resources, and more important their willingness to drop everything to help out. I thank Frank Dabek and Jinyang Li for their mentorship and friendship. To Meraki folks like John Bicket, Sanjit Biswas, Thomer Gil and Sean Rhea: we've missed you in 32-G980, but it was great while it lasted. I thank those in PDOS whom I've overlapped with, such as: Dave Andersen, Silas Boyd-Wickizer, Benjie Chen, Douglas De Couto, Kevin Fu, Michael Kaminsky, Petar Maymounkov, Athicha Muthitacharoen, Alex Pesterev, Jacob Strauss, Jayashree Subramanian, and Emmett Witchel. Thanks to Neena Lyall for her help in getting things done in CSAIL. And special thanks to the NTT Class of 1974.

The few times I ventured outside of 32-G980, I enjoyed the company of and learned much from: Dan Abadi, Emma Brunskill, James Cowling, Alan Donovan, Sachin Katti, Sam Madden, Dan Myers, Hariharan Rahul, the other members of CSAIL's hockey and softball teams, and the MIT Jazz Ensemble to name but a few. Barbara Liskov ensured the espresso beans rained like manna from the heavens; Kyle Jamieson, managed the milk pool. They conspired to keep me caffeinated, and I can't thank them enough.

Some of the work alluded to in this thesis (like Tame and OKWS) has found adoption at a Web concern dubbed `OkCupid.com`. I thank the engineers there for using this software and helping to make it better. I thank Sam Yagan and Chris Coyne for manning the OkCupid fort while I manned the ivory tower, and thanks to Chris for sharing just a fraction of his original ideas, which often made for interesting academic research projects.

Before I came to graduate school, I was David Mazières's research scientist at New York University. From him I learned the ropes of systems research: how to program, which problems were interesting, which solutions to consider, how to break in the back door, and generally speaking, how to be a scientist. I thank David for his software tools, his formative mentorship, and in general his intellectual-freedom-loving ethos. I thank other collaborators at NYU, including Michael Freedman, Dennis Shasha, Antonio Nicolosi, Jinyuan Li, and Yevgeniy Dodis.

Before starting at NYU, I met the love of my life (now wife) Sarah Friedberg (now Krohn). Sarah, thanks for your love, help, companionship and support through these five years of graduate school. I could not have done it without you, and I love you!

Still further in the past, I grew as an undergraduate under the tutelage of many luminaries, chief among them Michael O. Rabin, Peter Sacks and Helen Vendler. Thanks to them for instilling in me a love of academics and intellectual pursuits. In high school, Julie Leerburger more than anyone else taught me to write.[2] A final heartfelt thank-you to my mom, dad and sister, who taught me just about everything else.

[2]This sentiment was expressed in the same context by someone else one year ago.

# Contents

# Chapter 1

# Introduction

At least three trends indicate that the World Wide Web will expand upon its successes for the foreseeable future. Most obviously, more people than ever use it, with penetration in China alone reaching 253 million, up 56% year-over-year [65]. Second, Web sites are gaining prominence, amassing more sensitive data and exerting more influence over everyday life. Third, Web technology continues to mature, with Web sites progressing from static pages to dynamic, extensible computing platforms (e.g. Facebook.com [25] and OpenSocial [40]). All the while, Web security remains weak, as seen in the popular press [27, 60, 72, 52, 78, 79, 103] and published technical surveys [33, 101].

All trends spell potential for Web attackers, who see a growing population of victims using browsers and Web sites with unpatched security holes. As Web sites increase in sophistication, attackers get more powerful tools. As sites amass data and sway, attackers see increased payoffs for infiltration. In general, a worsening security trend threatens to mar the World's transition to Web-based computing, much as it marred the previous shift to networked PCs.

This thesis aims to improve the overall state of Web security. The first aspect of this challenge is to identify the right problem to attack: with so much wrong with the current Web infrastructure, which components need the most attention, and which are most amenable to drastic change? This introductory chapter provides a survey of the Web's weaknesses, and focuses attention on one in particular: the code running on Web servers that varies from site to site. We hypothesize that to secure such software, operating systems (OS) ought to expose better interfaces and better security tools to Web application developers. This thesis develops a general theory for what those interfaces and tools should be.

## 1.1 Threats to Web Security

Figure 1-1 shows a schematic diagram of how the Web works today. Client machines run Web browsers that routinely connect to different servers spread through the Internet. In this example, servers `a.com`, `b.com` and `c.com` are involved. The browser queries these servers, receiving HTML responses, then renders that HTML into a convenient user interface. The browser maps different server responses to different user interface (UI) elements, such as browser *tabs*, browser

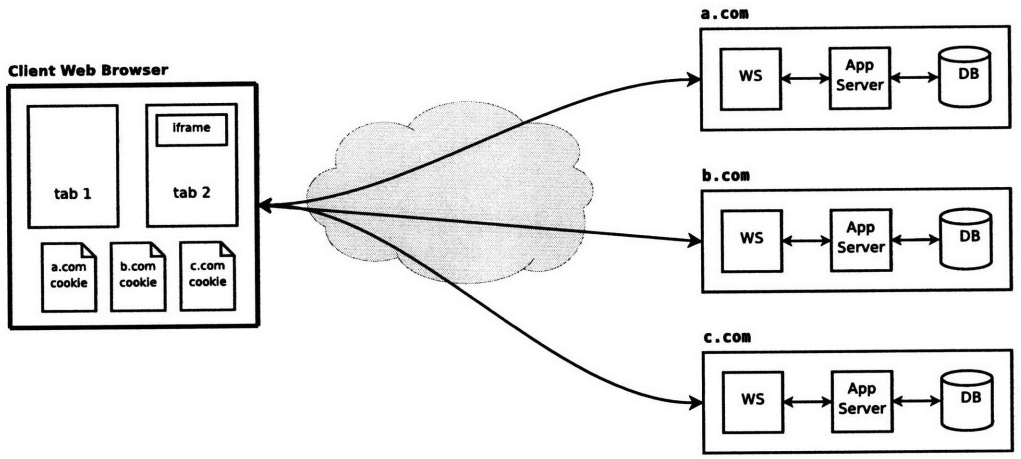


Figure 1-1: A schematic diagram of the Web architecture.

*windows*, or even `iframes`, which embed one server's response inside another's. In many Web systems, databases on the server side (those run by `a.com`, `b.com`, and `c.com`) house a majority of the useful data. However, those servers can store small data packets, known as *cookies*, persistently on clients' browsers. The browser enforces a security policy: allow `a.com` to set and retrieve a browser cookie for its own use, but disallow it to tamper with `b.com`'s cookie.

This description is coarse, but it gives a framework for taxonomizing Web attacks with more precision. In modern settings, all parts of the system in Figure 1-1 are under attack: the UI elements on the browser, the cookie system, the server infrastructure, the transport between client and server, etc. Weaknesses anywhere along this chain can allow adversaries to steal or corrupt sensitive data belonging to honest users. These attacks fall under three general headings: client, channel and server.

### 1.1.1 Exploiting the Channel

Most Web systems depend upon some authenticated, eavesdropping-resistant channel between the client and server. In practice, such a channel is difficult to establish. Users typically prove their identity to servers with simple passwords, but various social engineering attacks (especially *phishing* [18]) allow attackers to capture and replay these login credentials. In other cases, users have weak passwords, or the same passwords for many sites, allowing an unscrupulous site administrator at site `a.com` steal Alice's data from `b.com`. Also, users on compromised clients (those infected with malware) can lose their keystrokes (and hence their login credentials) to whomever controls their machines.

Even if a user can authenticate successfully to the remote server, other challenges to his session remain. Many Web sites do not use Secure Sockets Layer (SSL) [19], and hence their network communication is susceptible to tampering or eavesdropping. For even those Web sites that do, the question of how to distribute server certificates remains open. For instance, attackers

can distribute bogus certificates for well-known sites by exploiting confusing client UIs and weaknesses in the domain name system (DNS) [20, 47, 107].

### 1.1.2 Exploiting the Web Browser

Other attacks target the Web browser, preying on its management of sensitive data (like cookies) and its ability to install software like plug-ins. The most tried-and-true of these methods is Cross-Site Scripting (XSS) [10, 22, 54]. In XSS, an attacker Mallory controls site `c.com` but wishes to steal information from Victor as he interacts with site `a.com`. Mallory achieves these ends by posting JavaScript code to `a.com`, perhaps as a comment in a forum, a blog post, or a caption on a photo. If buggy, code running on `a.com`'s server will show Mallory's unstripped JavaScript to other users like Victor. His browser will then execute the code that Mallory crafted, with the privileges of `a.com`. Thus, Mallory's code can access Victor's cookie for `a.com`, and instruct the browser to send this data to `c.com`, which she controls. Once Mallory recovers Victor's cookie, she can impersonate Victor to `a.com` and therefore tamper with or steal his data. Other attacks use similar techniques to achieve different ends. Cross-site Request Forgery [50] hijacks user sessions to compromise server-side data integrity, for example, to move the user's money from his bank account to the attacker's. "Drive-by downloading" exploits browser flaws or XSS-style JavaScript execution to install malware on Victor's machine [81].

### 1.1.3 Exploiting the Web Servers

The third category of attacks will be the focus of this thesis: attacks on server-side Web computing infrastructure, like the Web servers, databases and middleware boxes that power sites `a.com`, `b.com` and `c.com` in our examples. Web servers are vulnerable to all of the "classic" attacks observed throughout the Internet's history, such as: weak administrator passwords, server processes (like `sendmail` or `ssh`) susceptible to buffer overflows, kernel bugs that allow local non-privileged users to assume control of a machine, or physical tampering.

Even if resilient to traditional attacks, Web servers face grave challenges. Many of today's dynamic Web sites serve as interfaces to centralized databases. They are gatekeepers, tasked with keeping sensitive data secret and uncorrupted. Unfortunately, these Web sites rely on huge and growing trusted computing bases (TCBs), which include: the operating system, the standard system libraries, standard system services (like e-mail servers and remote shell servers), application libraries, Web servers, database software, and most important, the Web application itself. In some cases, the application is off-the-shelf software (like MoinMoin Wiki, discussed in Chapter 8), but in many others, the code varies wildly from site to site.

Most popular search engines, e-commerce sites, social networks, photo-sharing sites, blog sites, online dating sites, etc., rely heavily on custom code, written by in-house developers. In documented cases, custom code can reach hundreds of thousands [57] or millions [15] of lines, contributed by tens to thousands of different programmers. Some Web sites like Amazon.com serve each incoming Web request with sub-requests to hundreds of different logical services (each composed undoubtedly of thousands of lines of proprietary code), running on machines distributed throughout a huge cluster [16]. In all cases, these exploding codebases do not benefit

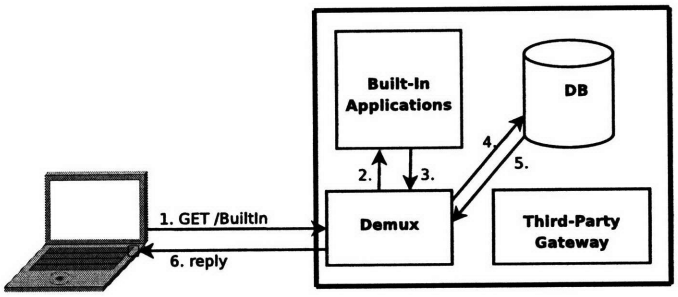


Figure 1-2: A request for a built-in Facebook Application

from the public scrutiny invested in popular open source software like the Linux Kernel or the Apache Web server.

In practice, bugs in custom code can be just as dangerous as those deeper in the software stack (like the kernel). In-house Web developers can introduce vulnerabilities in any number of ways: they can forget to apply access control checks, fail to escape input properly (resulting in SQL-injection attacks [42]), allow users to input JavaScript into forms (resulting in the XSS attacks mentioned previously), and so on. Web sites often update their code with little testing, perhaps to enhance performance in the face of a flash crowd, or to enable new features to stay abreast of business competitors. Tech news articles and vulnerability databases showcase many examples in which security measures failed, leaking private data [60, 72, 52, 78, 79, 103], or allowing corruption of high-integrity data [27]. In most of these cases, the off-the-shelf software worked, while site-specific code opened the door to attack. Indeed, surveys of Web attacks show an uptick in SQL-injection style attacks on custom-code, while popular off-the-shelf software seems to be stabilizing [101].

### 1.1.4 Exploiting Extensible Web Platforms

Innovations in Web technology have in recent years made server-side security worse. Whereas Web code was previously written by well-meaning yet sometimes careless professional Web developers, new extensible Web platforms like Facebook and OpenSocial allow malicious code authors to enter the fray. The stated intention of these platforms is to allow third-party application developers to extend server-side code. By analogy, the base Website (Facebook.com) is a Web-based operating system, providing rudimentary services like user authentication and storage. Independent developers then fill in the interesting site features, and the software they write can accesses the valuable data that Facebook stores on its servers. Applications exist for all manner of social interactions, from playing poker games to boasting about vacation itineraries.

Company white papers imply the implementation is straightforward. Figure 1-2 shows the request flow for one of Facebook's *built-in* applications, written by the Facebook developers. The browser makes a Web request to Facebook's server; the request is routed to Facebook's built-in code, which outputs a response after communicating with the company's database. For third-party applications (as in Figure 1-3), Facebook routes the client's request to Web servers

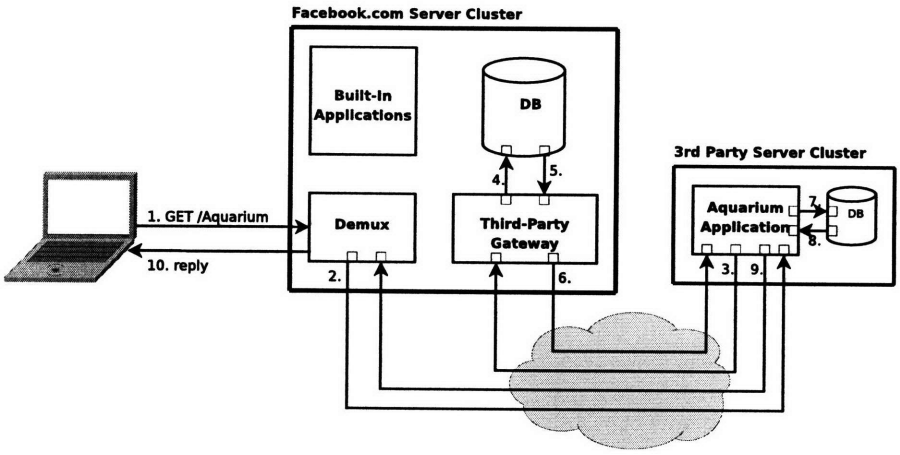


Figure 1-3: A request for a Third-Party Facebook Application

controlled by the third party. When, as shown in the example, Alice accesses the "Aquarium" third-party application, that software can access some of Alice's records stored on Facebook's database and then move that data to its own third-party database. (Some data like the user's e-mail address and password remain off-limits). If the "Aquarium" application becomes popular, it can collect data on behalf of more and more Facebook users, yielding a private copy of Facebook's database [24].

When Alice installs the Aquarium application, it becomes part of Facebook's TCB from her perspective. Alice must trust this software to adequately safeguard her data, but in practice, she has no guarantee Aquarium is up to the task. In some cases, vulnerabilities in well-intentioned third-party Facebook applications allow attackers to steal data [100], like that stored in the "Aquarium" database in Figure 1-3. In others, the applications are explicitly malicious, created with the primary goal of lifting data from Facebook [64]. Thus, Facebook has recreated for server applications a plague that already infests desktop applications: malware.

Following the well-known script, Facebook takes the same precaution against malware that current operating systems do: it prompts the user, asking the user if he really wants to run and trust the new application. This malware countermeasure has the same limitation on the Web as it does on the desktop: users have no way of knowing what software will do. Even the best automated static analysis tools (which Facebook does not offer) cannot fully analyze an arbitrary program (due to decidability limitations). The best such a scheme can accomplish is to partition the set of users into two groups: those who are cautious, who avoid data theft but do not enjoy new extensions; and those who are eager-adopters, who are bound to install malware by accident. A representative from Facebook captures this dismal state of affairs aptly: "Users should employ the same precautions while downloading software from Facebook applications that they use when downloading software on their desktop" [64]. In other words, Facebook's security plan is at best as good as the desktop's, which we know to be deeply flawed.

While we have focused on Facebook, the competing OpenSocial platform is susceptible to many of the same attacks [41]. Along the current trajectory, Web applications have the potential

to replace desktop applications, but not with secure alternatives. Rather, the more they mimic the desktop in terms of flexibility and extensibility, the less secure they become.

## 1.2 Approach

In either case—the traditional attacks against Web sites that exploit bugs, or the new "malware" attacks against extensible platforms—the problem is the same: TCBs that are too large, which contain either buggy code that acts maliciously in an attack, or inherently malicious code. The challenge, therefore, is to give Web developers the tools to cleanse their TCBs of bloated, insecure code. In particular, we seek an approach to purge site-specific code (e.g. Facebook's proprietary software) and third-party code (e.g. Aquarium) from TCBs, under the assumption that it receives a fraction of the scrutiny lavished on more popular software like Linux or Apache. The thesis presents a new design, in which a small, isolated module controls the Web site's security. Bugs outside of this module (and outside the TCB) might cause the site to malfunction, but not to leak or corrupt data. A security audit can therefore focus on the security module, ignoring other site-specific code. Though this thesis focuses on "traditional" attacks in which adversaries exploit bugs in well-intentioned code, the techniques generalize, and have the potential to secure server-side malware in future work.

Recent work shows that *decentralized information flow control* (DIFC) [70, 21, 113] can help reduce TCBs in complex applications. DIFC is a variant of information flow control (IFC) [5, 7, 17] from the 1970s and 1980s. In either type of system, a tamper-proof software component (be it the kernel, the compiler, or a user-level *reference monitor*) monitors processes as they communicate with each other and read or write files. A process $p$ that reads "secret" data is marked as having seen that secret, as are other processes that $p$ communicates with, and processes that read any files that $p$ wrote after it saw the secret. In this manner, information flow control systems compute the transitive closure of all processes on the system who could possibly have been influenced by a particular secret. The system then constrains how processes in the closure export data out of the system. In the case of IFC systems, only a trusted "security officer" can authorize export of secret data. In a DIFC system the export privilege is decentralized (the "D" in DIFC), meaning any process with the appropriate privileges can authorize secret data for export.

This thesis applies DIFC ideas to Web-based systems. As usual, an a chain of communicating server-side processes serves incoming Web requests. Processes in the chain that handle secret data are marked as such. If these marked processes wish to export data, they can do so only with the help of a process privileged to export secrets (known as a *declassifier*). A simple application-level declassifier appropriate for many Web sites allows Bob's secret data out to Bob's browser but not to Alice's. Those processes without declassification privileges then house a majority of the site specific code. They can try to export data against the system's security policies, but such attempts will fail unless authorized by a declassifier.

**Example: MoinMoin Wiki** This thesis considers the popular MoinMoin Website package [68] as a tangible motivating example. MoinMoin is Wiki application: a simple-to-use Web-based

file system that allows Web users to edit, create, modify, and organize files. The files are either HTML pages that link to one another, or binary files like photos and videos. Like a file system, MoinMoin associates an access control list (ACL) [90] with each file, which limit per-file read and write accesses to particular users.

Internally, the Wiki's application logic provides all manners of text-editing tools, parsers, syntax highlighting, presentation themes, indexing, revision control, etc., adding up to tens of thousands of lines of code. The security modules that implement ACLs are largely orthogonal in terms of code organization. However, in practice, all code must be correct for the ACLs to function properly. Consider the simple case in which Alice creates a file that she intends for Bob never to read. The code that accepts Alice's post, parses it and writes it to storage on the server must faithfully transmit Alice's ACL policy, without any side-effects that Bob can observe (like making a public copy of the file). When Alice's file later exits the server on its way out to client browsers, the code that reads the file from the disk, parses it, renders the page and sends the page out to the network must all be correct in applying ACL policies, lest it send information to Bob against Alice's wishes. All of this code is in MoinMoin's TCB.

By contrast, a DIFC version of MoinMoin, (like that presented in Chapter 8) enforces *end-to-end secrecy* with small isolated declassifiers (hundreds of lines long). Bugs elsewhere in the Wiki software stack can cause program crashes or other fail-stop behavior but do not disclose information in a way that contradicts the security policy. Only bugs in the declassifiers can cause such a leak.

Web applications like Wikis also stand to benefit from *integrity* guarantees: that important data remains uncorrupted. Imagine Alice and Charlie are in a group that communicates through the Wiki to track important decisions, such as whether or not to hire a job applicant. They must, again, trust a large stack of software to faithful relay their decisions, and any bugs in the workflow can mistakenly or maliciously flip that crucial "hire" bit. An *end-to-end* integrity policy for this example only allows output if all data handlers meet integrity requirements — that they all were certified by a trusted software vendor, for instance. As above, bugs that cause accidental invocation of low-integrity code produce fail-stop errors: the program might crash or refuse to display a result, but it will never display a result of low integrity.

## 1.3 Challenges

DIFC techniques are powerful in theory, but at least five important hurdles prevent their application to modern Web sites (like MoinMoin Wiki), which no existing system has entirely met. They are:

**Challenge 1: DIFC clashes with standard programming techniques.** DIFC originated as a programming language technique [69], and it has found instantiations in many strongly typed languages, such as Java [70], Haskell [61], the typed lambda calculus [115], ML [77], and so on. However, such techniques are not compatible with legacy software projects and generally require rewriting applications and libraries [44]. Also, language-based information flow control is a specialized form of type-checking. It cannot apply to languages without static typing, such

as Python, PHP, Ruby, and Perl, which are popular languages for developing modern Web sites. Thus, a challenge is to adopt DIFC techniques for programs written for existing, dynamically-typed, threaded languages.

**Challenge 2: DIFC clashes with standard OS abstractions.** Popular operating systems like Linux and Microsoft Windows have a more permissive security model than DIFC systems: they do not curtail a process's future communications if it has learned secrets in the past. Thus, programs written to the Linux or Windows API often fail to operate on a DIFC system. Current solutions make trade-offs. On one extreme, the Asbestos Operating System [21] implements DIFC suitable for building Web applications, but does not expose a POSIX interface; it therefore requires a rewrite of many user-space applications and libraries. On the other extreme, the SELinux system (a "centralized" information flow control system) preserves the entire Linux interface, but at the expense of complicated application development. It requires lengthy and cumbersome application-specific policy files that are notoriously difficult to write.

**Challenge 3: DIFC at the kernel-level requires a kernel rewrite.** Asbestos [21] and HiStar [113] are DIFC-based kernels, but are written from scratch. Such research kernels do not directly benefit from the ongoing (and herculean) efforts to maintain and improve kernel support for hardware and an alphabet-soup of services, like NFS, RAID, SMP, USB, multicores, etc. Asbestos and HiStar are also fundamentally *microkernels*, whereas almost every common operating system today, for better or worse, follows a *monolithic* design. So a challenge is to reconcile DIFC with common OS organization and popular kernel software.

**Challenge 4: DIFC increases complexity.** Common operating systems like Linux already complicate writing secure programs [56]. Information flow systems are typically more complex. Security-typed languages require more type annotations; system calls in security-enabled APIs have additional arguments and error cases. Moreover, most information flow systems use lattice-based models [17], adding mathematical sophistication to the basic programming API. A challenge is to curtail complexity in the DIFC model and the API that follows.

**Challenge 5: DIFC implementations have covert channels.** All information flow control implementations, whether at the language level or the kernel level, suffer from *covert channels*: crafty malicious programs can move information from one process to another outside of modeled communication channels. For example, one exploitative process might transmit sensitive information by carefully modulating its CPU use in a way observable by other processes. For fear of covert channels, language-based systems like Jif disable all threading and CPU parallelism. As discussed in Section 3.4, some OS-based IFC implementations like Asbestos and IX [66] have wide covert channels inherent in their API specifications. HiStar sets a high standard for covert channel mitigation, but timing and network-based channels persist. So a challenge, as always, is to build a DIFC system that suffers from as few covert channels as possible.

## 1.4 *Flume*

We present *Flume*, a system that answers many of the above challenges. At a high level, *Flume* integrates OS-level information flow control with a legacy Unix-like operating system. Flume allows developers to build DIFC into legacy applications written in any language, either to upgrade their existing security policies or to achieve new policies impossible with conventional security controls.

### 1.4.1 Design

Because it is notoriously difficult to reason about security in Unix [56]—due to its wide and imprecisely defined interface—DIFC for Unix unfolds in four steps, each corresponding to a chapter.

1. First, we describe, independent of OS specifics like reliable communication, what properties a DIFC system ought to uphold. The logical starting point for such a definition is the IFC literature [5, 7, 17], in which processes communicate pairwise with one-way messaging. A new and simplified model, called the Flume model, extends the original IFC definitions to accommodate decentralized declassification (as in DIFC). See Chapter 3.

2. Then we state this model formally, using the Communicating Sequential Processes (CSP) Process Algebra [46]. See Chapter 4.

3. We then prove that the Flume model fits standard definitions of *non-interference*: that is, the processes who have seen secret data cannot have any impact on the processes who haven't. Such formalisms separate Flume from other IFC operating systems whose APIs encapsulate wide data leaks. See Chapter 5.

4. Finally, we provide details for building a practical system that fits the model, filling in details like reliable, flow-controlled interprocess communication (IPC) and file I/O. See Chapter 6.

The new OS abstraction that allows Flume to fit DIFC to a Unix-like API is the *endpoint*. Flume represents each resource a process uses to communicate (e.g., pipes, sockets, files, network connections) as an endpoint. A process can configure an endpoint, communicating to Flume what declassification policy to apply to all future communication across that endpoint. However, Flume constrains endpoints' configurations so that processes cannot leak data that they do not have privileges to declassify. For instance, a process that has a right to declassify a secret file can establish one endpoint for reading the secret file and another endpoint for writing to a network host. It can then read and write as it would using the standard API. Flume disallows a process without those declassification privileges from holding both endpoints at once. Chapter 6 covers endpoints in detail.

### 1.4.2 Implementation

Flume is implemented as a user-space reference monitor on Linux with few modifications to the underlying kernel. Legacy processes on a Flume system can move data as they always did. However, if a process wishes to access data under Flume's control, it must obey DIFC constraints, and therefore cannot leak data from the system unless authorized to do so. Unlike prior OS-level DIFC systems, Flume can reuse a kernel implementation, driver support, SMP support, administrative tools, libraries, and OS services (TCP/IP, NFS, RAID, and so forth) already built and supported by large teams of developers. And because it maintains the same kernel API, Flume supports existing Linux applications and libraries. The disadvantage is that Flume's trusted computing base is many times larger than Asbestos's or HiStar's, leaving the system vulnerable to security flaws in the underlying software. Also, Flume's user space implementation incurs some performance penalties and leaves open some covert channels solvable with deeper kernel integration. Chapter 7 discusses implementation details.

### 1.4.3 Application and Evaluation

To evaluate Flume's programmability, we ported MoinMoin Wiki to the Flume system. As mentioned above, MoinMoin Wiki 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. The challenge is to capture MoinMoin'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. With such a refactoring of code, only bugs in the security module (as opposed to the large tangle of MoinMoin code and its plug-ins) can compromise end-to-end security.

An additional challenge is to graft a new policy onto MoinMoin code that could not exist outside of Flume: *end-to-end integrity protection*. Though MoinMoin can potentially pull third-party plug-ins into its address space, cautious users might demand that plug-ins never touch (and potentially corrupt) their sensitive data, either on the way into the system, or on the way out. We offer a generalization of this policy to include different integrity classes based on which plug-ins are involved.

FlumeWiki achieves these security goals with only a thousand-line modification to the original MoinMoin system (in addition to the new thousand-line long security module). Though prior work has succeeded in sandboxing legacy applications [113] or rewriting them anew [21], the replacement of an existing dataflow policy with a DIFC-based one is a new result. The FlumeWiki TCB therefore looks different from that of original MoinMoin: it contains the Flume system, and the security module, but not the bulk of MoinMoin's application-specific code. MoinMoin's TCB does not contain Flume, but it does contain all MoinMoin code, including any plug-ins installed on-site. As a result of this refactoring of the TCB, FlumeWiki solves security bugs in code it inherited from the original MoinMoin, as well as previously unreported MoinMoin bugs discovered in the process of implementing FlumeWiki. Chapter 8 describes the FlumeWiki application in greater detail.

As described in Chapter 9, experiments with FlumeWiki on Linux show the new system per-

forms within a factor of two of the original. Slow-downs are due primarily to Flume's user-space implementation, though the Flume design also accommodates kernel-level implementations.

## 1.5 Contributions

In sum, this thesis makes the following technical contributions:

- 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 "centralized" information flow control [5, 7, 17], with small extensions for decentralization and communication abstractions found in widely-used operating systems.
- A formal model and proof of correctness.
- A new abstraction—endpoints—that bridge the gap between DIFC and legacy Unix abstractions.
- The first design and implementation of process-level DIFC for stock operating systems (OpenBSD and Linux). Flume is useful for securing Web sites, and also other client- and server-side software.
- 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.

## 1.6 Limitations, Discussion and Future Work

This thesis has important limitations. Thought it presents a formal Flume model, there are no guarantees that the implementation actually meets the model. Even if it did, Flume is susceptible to covert channels, such as *timing channels*, quota-exhausting channels, wallbanging, etc. Though Flume is simpler in its mathematics and specifications than some other recent IFC systems, it is still significantly more complicated at this stage than standard Unix. Indeed, Flume can run atop a Linux system and coexist with many legacy applications, but not all Unix system calls fit the DIFC model, and hence, Flume's support of standard APIs is approximate at best. Similarly, not all of FlumeWiki's features work; some are fundamentally at odds with DIFC guarantees (e.g., hit counters; see Section 8.9).

Other techniques might also solve some of the same problems but with less overhead. One can envision other design paths considered but not taken, such as: building DIFC into a runtime (like Python's) rather that into the operating system; protecting programmers against their own bugs rather than generalizing to the more prickly defense against malicious code; a language-based approach like Jif's with finer-grained labeling.

Chapter 10 discusses these limitations as well as the more positive outcomes of the Flume experience: lessons gleaned, ideas about programmability and practicality, and some projections for future projects. The end goal of this research is to build secure and extensible server-based computing platforms. We discuss how Flume has fared in that endeavor, and what work remains.

# Chapter 2

# Related Work

Attacks against Web services have kept security researchers busy. In answer the bonanza of weaknesses that dog Web-based system today (surveyed in Section 1.1), a similarly diverse set of solutions has sprung up.

## 2.1 Securing the Web

**Securing the Channel** Many proposals exist to combat phishing, some using browser-based heuristics [13], others using mobiles phones as a second authentication factor [75]. In general, no solution has found either widespread acceptance or adoption, and indeed phishing promises to be an important problem for years to come. We offer no solution in this thesis but believe the problem is orthogonal to those we consider.

As hardware manufacturers continue to expand the computing resources available to applications (even with dedicated cores for cryptography), SSL channel protection should become available to more Web sites. And improved user interfaces [84] and dedicated infrastructure [107] has taken aim at attacks that attempt to distribute bogus SSL certificates.

**Securing the Browser** As for browser-based attacks, such as XSS, XSRF, and drive-by downloads, the most obvious solutions involve better sanitization on the server-side, using packages like HTML purify [110]. Of course, servers cannot sanitize HTML in `iframes` they did not create and cannot modify (like third party advertisements), and with current technology are at the mercy of those third parties to provide adequate security protections. Research work like MashupOS gives the browser finer-grained control over its own security, at the cost of backwards compatibility [105].

**Securing the Server** As for the server-side, over time, system administrators have developed practices to mitigate the risks of well-understood attacks, such as data-center firewalls, and two-factor authentication for administrators [93]. Administrators can apply security patches to close

holes in services that listen on network ports, and holes in the kernel that allow privilege escalation [3].

The remaining issue is one of the most challenging, and the subject of this thesis: how to secure the custom-written server-side software that makes the Web site work while upholding a desired security policy. Web *application* code is mushrooming, especially as new toolkits (e.g. Ruby on Rails [97] and Django [30]) open up Web programming to an ever-larger pool of developers, many of whom are not well-versed in secure programming principles. The question becomes, what tools should the underlying infrastructure (be it the operating system, compilers or interpreters) give application developers to help them write secure code? And can these techniques apply to Facebook-like architectures, in which unvetted third party developers contribute application code?

One vein of work in this direction is backwards-compatible security improvements to stock operating systems and existing applications, such as: buffer overrun protection (e.g., [14, 55]), system call interposition (e.g., [34, 49, 80, 32, 102]), isolation techniques (e.g., [35, 51]) and virtual machines (e.g., [37, 104, 108]). Flume uses some of these techniques for its implementation (e.g., LSMs [109] and systrace [80]). Few if any of these techniques, however, would protect against bugs high up in the Web software application stack (say in MoinMoin code). That is, many exploitable Web applications behave normally from the perspective of buffer-overrun analysis or system call interposition, even as they send Alice's data to Bob's network connection, against Alice's wishes. This thesis instead explores deeper changes to the API between (Web) applications and the kernel, allowing applications to express high-level policies, and kernel infrastructure to uphold them. This approach is heavily influenced by previous work in mandatory access control (MAC).

## 2.2 Mandatory Access Control (MAC)

Mandatory access control (MAC) [91] 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. Decentralized information flow control (DIFC) promises to support such situations better than most MAC mechanisms, because it partially delegates the setting of policy to the individual applications.

Bell and LaPadula describe an early mathematical model for MAC [4] and an implementation on Multics [5]. Their work expresses two succinct rules that capture the essence of mandatory security. The first is the simple security property, or "no read-up:" that when a "subject" (like an active process) "observes" (i.e. reads) an "object" (like a static file), the subject's security label must "dominate" (i.e. be greater than) that of the object. The second is the $\star$-property, or "no write-down:" that a subject's label must dominate the label of any object that it influences. Biba noted that similar techniques also apply to integrity [7]. Denning concurrently expressed MAC ideas in terms of mathematical lattices [17] but advocated a compile-time rather than run-

time approach, for fear of covert channels (see Section 3.4).

SELinux [62] and TrustedBSD [106] 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 [98]. Flume, like SELinux, uses the Linux security module (LSM) framework in its implementation [109]. 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 thesis.

TightLip [111] 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. Like TightLip, Flume can also be used to prevent privacy leaks. 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 [66] and LOMAC [31] 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.

## 2.3 Specialized DIFC Kernels

One line of DIFC research, taken by the Asbestos [21] and HiStar [113] projects, is to replace a standard kernel with a new security kernel, then build up, eventually exposing DIFC to applications.

Asbestos [21, 9] and HiStar [113] 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.

The HiStar project in particular has succeeded in exposing a standard system call interface to applications, so that some applications work on HiStar as they would under standard Unix. 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. The complication with the HiStar approach, however, is that the specifics for managing information flow are hidden deep in the system library. If a legacy application component (such as a Web server, a Web library, or an off-the-shelf Web application) causes a security violation when run on HiStar's library, it would be difficult to refactor the application to solve the problem. Such controls could be factored into a new library, and this thesis 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.

## 2.4 Language-Based Techniques

The original line of research in DIFC comes from the programming languages community. Myers and Liskov introduced a decentralized information model [69], 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 [70]. In Jif, the programmer annotates variable declarations, function parameters, function return values, structure members, etc., to describe what type of secrecy and integrity that data ought to have. The compiler then type-checks the program to ensure that it does not move data between incompatible secrecy and integrity categories. In such a system, declassifiers are class methods (rather than processes). Programs that type-check and have correct declassifiers are believed to be secure.

There are several benefits to this approach. First, security labels can apply to data elements as small as a byte, whereas in operating system-based techniques, the smallest labeled granularity is a process (in the case of Asbestos and Flume) or thread (in the case of HiStar). Second, Jif can limit declassification privileges to specific function(s) within a process, rather than (as with Flume) to the entire process. Jif programs can also run on legacy operating systems, and some work has shown how Jif's computations can be split across multiple processes and/or machines [112]. The Swift work, based on Jif, focuses on splitting Web applications in particular between the server and browser. The technique is to write an application in Jif, and then automated tools generate server-side Java and client-side JavaScript that uphold the intended security policy [11].

On the other hand, Jif requires applications (such as Web services [12] and mail systems [44]) to be rewritten while Flume provides better support for applying DIFC to existing software. OS-based approaches can also accommodate dynamically typed scripting languages like Perl, Python, PHP, Ruby, and even shell scripting. These languages are exactly the ones that many programmers call upon when developing Web applications. Finally, language-based approaches like Jif still depend upon the underlying operating system for access to the file system, network, and other resources available through the system call interface. As Hicks et al. point out, the Jif work is indeed complementary to progress in mandatory access control for the operating system [45].

## 2.5 Capabilities Systems

The *capabilities* literature proposes another technique for securing kernels and higher level applications. In their seminal work on the "principle of least privilege" [91], Saltzer and Schroeder argue that processes should possess as few privileges as possible when performing their task;

in the case of compromise, such processes are less useful to attackers than those that own many privileges they do not require. By contrast, operating systems like Linux and Windows implicitly grant all manner of privileges (i.e., capabilities) to applications, which often misuse them due to bugs or misconfiguration [43]. For example, if Alice runs Solitaire on Windows, the program can "scan [her] email for interesting tidbits and them on eBay to the highest bidder" [67]. Capabilities systems like KeyKOS [53], ErOS [96] and CoyotOS [95] require processes to request, grant and manage these privileges (i.e. capabilities) explicitly. The assumption is that if kernel components and/or applications must more carefully manage their capabilities to complete their tasks, they are less likely to request (and lose control of) capabilities they do not need.

The core idea behind capability systems—that system resources are individually addressable and accessible via capabilities—does not solve the problem explored in this thesis. Imagine building a Web-based application like MoinMoin with capabilities. When Alice logs onto the system, an instance of MoinMoin launches with Alice's read and write capabilities, meaning it can read and write those files belonging to Alice. But such an application can also copy Alice's data to a temporarily, world-readable file in the Wiki namespace, allowing Bob to later access it. In other words, capability-based policies do not, in and of themselves, capture the transitivity of influence that MAC systems do: they do not track Alice's data through the Wiki code to the world-readable file.

There is, however, an historical confluence between capability and MAC systems. Capability systems like KeyKOS have succeeded in implementing MAC policies as a policy in a reference monitor on top of a capabilities-based kernel. Conversely, the capabilities literature has influenced some variants of MAC like Asbestos [21] (described below). These system have a default security policy as in Bell-LaPadula, but processes that hold the appropriate capabilities are permitted to alter this policy. Thus, MAC and capabilities prove complementary.

# Chapter 3

# The Flume Model For DIFC

The Flume Model for DIFC describes an interface between an operating system kernel and user-space applications. Like typical OS models, Flume's assumes a trust division between the kernel and applications: that the kernel ought be correct and bug free; and that the kernel can defang buggy or malicious user-space applications. Like MAC models (see Section 2.2), Flume's requires that the kernel track sensitive information flow through an arbitrary network of user-space applications. As in *decentralized* information flow control, the Flume model permits some (but not all) of those applications to act as declassifiers, selectively disclosing sensitive information.

This chapter *informally* specifies Flume's DIFC model, describing how it answers the challenges listed in Section 1.3. In particular:

- programming language agnosticism, in answer to Challenge 1;
- compatibility with Unix primitives, in answer to Challenges 2 and 3;
- simplicity, in answer to Challenge 4;
- and mitigation of covert channels, in answer to Challenge 5.

## 3.1 Tags and Labels

Flume uses *tags* and *labels* to track data as it flows through a system. Let $\mathcal{T}$ 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 $\mathcal{T}$. Labels form a lattice under the partial order of the subset relation [17]. Each Flume process $p$ has two labels, $S_p$ for secrecy and $I_p$ for integrity. Both labels serve to (1) summarize which types of data have influenced $p$ in the past and (2) regulate where $p$ can read and write in the future. Consider a process $p$ and a tag $t$. If $t \in S_p$, then the system conservatively assumes that $p$ has seen some private data tagged with $t$. In the future, $p$ can read

more private data tagged with $t$ but requires consent from an authority who controls $t$ before it can reveal any data publicly. If there are multiple tags in $S_p$, then $p$ requires independent consent for each tag before writing publicly. Process $p$'s integrity label $I_p$ serves as a lower bound on the purity of its influences. If $t \in I_p$, then every input to $p$ has been endorsed as having integrity for $t$. To maintain this property going forward, the system only allows $p$ to read from other sources that also have $t$ in their integrity labels. Files (and other objects) also have secrecy and integrity labels; they can be thought of as passive processes.

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".

**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. The following four properties suffice to protect Bob's data. For any 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. Any untrusted process $p$ cannot remove $b$ from $S_p$
4. $p$ with $b \in S_p$ cannot write over the network (or to any other destinations outside the system).

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: how to prevent untrustworthy software from corrupting important files. 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 (perhaps due to an `LD_LIBRARY_PATH` misconfiguration).

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 $I_p$;
4. and $p$ 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

Decentralized IFC (DIFC) is a relaxation of centralized (or "traditional") IFC. 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. Capabilities are objects from the set $\mathcal{O} = \mathcal{T} \times \{-, +\}$. For tag $t$, the capabilities are $t^+$ and $t^-$. Each process *owns* a set of capabilities $O_p \subseteq \mathcal{O}$. 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 $D_p$ where

$$D_p \triangleq \{t \mid t^+ \in O_p \text{ and } t^- \in O_p\}$$

represents all tags for which $p$ has dual privilege.

Any process can allocate a tag. Tag allocation yields a randomly-selected tag $t \in \mathcal{T}$ and sets $O_p \leftarrow O_p \cup \{t^+, t^-\}$, granting $p$ dual privilege for $t$. Thus, tag allocation exposes no information about system state. Flume also supports a *global* capability set $\hat{O}$. Every process has access to every capability in $\hat{O}$, useful for implementing certain security policies. For instance, *export*

*protection* is a policy in which unprivileged processes can traffic secret data internally to the system but cannot expose it. To implement such a policy, a process creates a tag $t$, adding $t^+$ to $\hat{O}$. Now any process can add $t$ to their secrecy label, and read data tagged with tag $t$. But only a privileged few have access to $t^-$, required to remove $t$ from a secrecy label and therefore export data tagged with $t$ from the system. See Section 3.2 for more detail.

A process $p$'s effective set of capabilities is given by:

$$\bar{O}_p \triangleq O_p \cup \hat{O}$$

Similarly, its effective set of dual privileges is given by:

$$\bar{D}_p \triangleq \{t \mid t^+ \in \bar{O}_p \wedge t^- \in \bar{O}_p\}$$

Tag allocation can update $\hat{O}$; an allocation parameter determines whether the new tag's $t^+$, $t^-$, or neither is added to $\hat{O}$ (and thus to every current and future process's $\bar{O}_p$).

Lest processes manipulate the shared set $\hat{O}$ to leak data, Flume must control it carefully. A first restriction is that processes can only add tags to $\hat{O}$ when allocating tags. If Flume allowed arbitrary additions to $\hat{O}$, a process $p$ could leak information to a process $q$ by adding either adding or refraining from adding a pre-specified tag to $\hat{O}$. A second restriction is that no process $p$ can enumerate $\hat{O}$ or $\bar{O}_p$. If Flume allowed enumeration, $p$ could poll $\| \hat{O} \|$ while $q$ allocated new tags, allowing $q$ to communicate bits to $p$. Processes can, however, enumerate their non-global capabilities (those in $O_p$), since they do not share this resource with other processes. See Chapter 4 for a formal treatment of potential pitfalls induced by $\hat{O}$.

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 6.2). And finally, some notation for manipulating sets of capabilities: for a set of *capabilities* $O \subseteq \mathcal{O}$, we define:

$$O^+ \triangleq \{t \mid t^+ \in O\}$$
$$O^- \triangleq \{t \mid t^- \in O\}$$

**Example: Secrecy** As described above, 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 $\hat{O}$, 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 $\hat{O}$. 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 is useful for protecting short and very sensitive secrets, like passwords. That is, if Alice thinks that her system has some low-capacity covert channels, she must concede that Bob can leak her export-protected out of the system, if given the time and

resources. But Bob cannot see her read-protected data in the first place, and thus, it is better protected against covert channels (including timing channels) [113].

**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 $\hat{O}$. 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 $v^+ \notin \bar{O}_p$, 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 (Security in the Flume model). 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 6.

**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 [21, 31, 66], but implicit label changes turn the labels themselves into covert channels [17, 113] (see Section 3.4). When a process requests a change, only those label changes permitted by a process's capabilities are safe:

*Definition* 2 (Safe label change). 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:

$$L' - L \subseteq (\bar{O}_p)^+ \quad \text{and} \quad L - L' \subseteq (\bar{O}_p)^-$$

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.

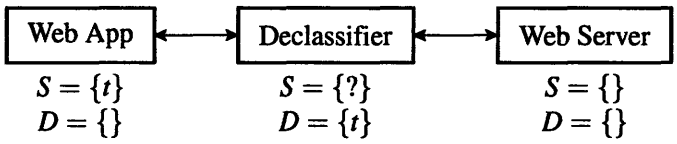


Figure 3-1: An example Web system; what should the declassifier's label be?

### 3.3.1 Safe Messages

Information flow control restricts process communication to prevent data leaks. The Flume model restricts communication among unprivileged processes as in traditional IFC: $p$ can send a message to $q$ only if $S_p \subseteq S_q$ ("no read up, no write down" [5]) and $I_q \subseteq I_p$ ("no read down, no write up" [7]).

For declassifiers—those processes that hold special privileges—these traditional IFC rules are too restrictive. Consider Figure 3-1 for example. In this simple example, a Web application runs with secrecy $S = \{t\}$, meaning it can read and compute on data tagged with secrecy $t$. Since the application is unprivileged ($D = \{\}$), it cannot export this data on its own; it relies on the privileged declassifier ($D = \{t\}$) on its right to do so. If the declassifier decides to declassify, it sends the data out to the network via the Web server, which runs with an empty secrecy label. Thus, data can flow in this example from high ($S = \{t\}$) to low ($S = \{\}$) with the help of the declassifier in the middle.

The question becomes, what should the declassifier's secrecy label be? One idea is for the declassifier to explicitly switch between $S = \{\}$ and $S = \{t\}$, as it communicates to its left or right. Though this solution sometimes works for *sending*, it is impractical for asynchronous *receives*: the declassifier has no way of knowing when its left or right partner will send, and therefore cannot make the necessary label change ahead of time. Another idea is that the declassifier run with $S = \{t\}$ and only lower its label to $S = \{\}$ when it sends to its right. But this approach does not generalize—imagine a similar scenario in which the Web server runs with secrecy $S = \{u\}$ and the declassifier has dual privileges for both $t$ and $u$. This second approach is also ungainly for multithreaded declassifiers with blocking I/O operations.

Flume's solution is a general relaxation of communication rules for processes with privilege, like declassifiers. Specifically, if two processes *could* communicate by changing their labels, sending a message using the traditional IFC 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 $\bar{D}_p$, 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 $S_p - \bar{D}_p$ and raise its integrity to $I_p \cup \bar{D}_p$. It could receive the most messages if it were to raise secrecy to $S_p \cup \bar{D}_p$ and lower integrity to $I_p - \bar{D}_p$.

The following definition captures these *hypothetical* label changes to determine what messages are safe:

*Definition* 3 (Safe message). A message from $p$ to $q$ is *safe* iff

$$S_p - \bar{D}_p \subseteq S_q \cup \bar{D}_q \quad \text{and} \quad I_q - \bar{D}_q \subseteq I_p \cup \bar{D}_p$$

For processes with no dual privilege ($\bar{D}_p = \bar{D}_q = \{\}$), Definition 3 gives the traditional IFC definition for safe flows. On the other hand, if $p$ must send with a hypothetical secrecy label of $S_p - \bar{D}_p$, then $p$ is declassifying the data it sends to $q$. If $q$ must receive with secrecy $S_q \cup \bar{D}_q$, then it is declassifying the data it received from $p$. In terms of integrity, if $p$ must use an integrity label $I_p \cup \bar{D}_p$, then it is endorsing the data sent, and similarly, $q$ is endorsing the data received with integrity label $I_q - \bar{D}_q$.[1]

This definition of message safety might raise fears of *implicit declassification*. A process $p$ with a non-empty $D_p$ is *always* declassifying (or endorsing) as it sends or receives messages, whether it intends to or not. The capabilities literature strongly discourages such behavior, claiming that implicit privilege exercise inevitably results in "confused deputy problems," in which attackers exploit honest applications' unintended use of privileges [43]. In defining the Flume model, we present rules that make communication as permissive as possible without leaking data. Chapter 6 describes how the Flume implementation avoids the confused deputy problem, requiring applications to explicitly declassify (and endorse) data as they send and receive.

### 3.3.2 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 $O_x = \{\}$. 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 $I_x \subseteq I_p$).

### 3.3.3 Objects

Objects such as files and directories are modeled as processes with empty ownership sets, and 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.

[1] Declassification or endorsement can also occur when a process $p$ makes actual (rather than hypothetical) label changes to $S_p$ or $I_p$, respectively. See Section 3.3.

### 3.3.4 Examples

**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 $b^- \notin \bar{O}_p$) meet the four stated requirements from Section 3.1. In the logic below, recall that $b$ is an export-protect tag; therefore $b^+ \in \hat{O}$ and also $b^- \notin \hat{O}$. For unprivileged processes like Bob's editor: $b^- \notin \bar{O}_p$, and thus $b \notin \bar{D}_p$.

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 $b^+ \in \hat{O}$, any process can write such files. Reading an object is modeled as an information flow from $f$ to $p$, which requires that $S_f \subseteq S_p \cup \bar{D}_p$ by Definition 3. Since $b \in S_f$, and $b \notin \bar{D}_p$, 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, $S_p - \bar{D}_p \subseteq S_q \cup \bar{D}_q$. Since $b$ is in neither $\bar{D}_p$ nor $\bar{D}_q$, 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 $b^+ \in \hat{O}$, 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.

**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 $O_p = \{b^-\}$. 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's programs can export it—and in particular, software running as "Bob" cannot export it (since it contains Alice's private data). When Alice logs on, she can use a similar protocol to read Bob's suggestions, choose one, and

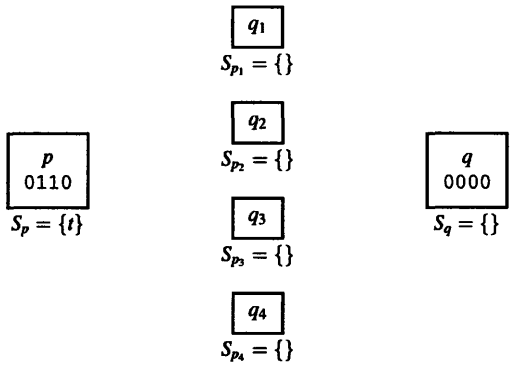


Figure 3-2: The "leaking" system initializes.

export that choice to Bob in a file $g$ labeled $S_g = \{b\}$. Because at least one of Alice's or Bob's export-protection tag protects all data involved with the exchange, other users like Charlie can learn nothing from it.

## 3.4 Covert Channels in Dynamic Label Systems

As described in Section 3.3, processes in Flume change their labels explicitly; labels do not change implicitly upon message receipt, as they do in Asbestos [21] or IX [66]. We show by example why implicit label changes (also known as"floating" labels) enable high-throughput information leaks.

Consider a process $p$ with $S_p = \{t\}$ and a process $q$ with $S_q = \{\}$, both with empty ownership sets. In a floating label system like Asbestos, $p$ can send a message to $q$, and $q$ will successfully receive it, but only after the kernel raises $S_q = \{t\}$. Thus, the kernel can track which processes have seen secrets tagged with $t$, even if those processes are uncooperative. Of course, such a scheme introduces new problems: what if a process $q$ doesn't want its label to change from $S_q = \{\}$? For this reason, Asbestos also introduces "receive labels," which serve to filter out incoming traffic to a process, allowing it to avoid unwanted label changes.

The problem with floating is best seen through example (c.f., Figures 3-2 through 3-4). Imagine processes $p$ and $q$ as above, with $p$ wanting to leak the 4-bit secret "`0110`" to $q$. The goal is for $p$ to convey these bits to $q$ without $q$'s label changing. Figure 3-2 shows the initialization. $q$ launches 4 helper processes ($q_1$ through $q_4$), each with a label initialized to $S_{q_i} = \{\}$. $q$'s version of the secret starts out initialized to all `0`s, but it will overwrite some of those bits during the attack.

Next $p$ communicates selected bits of the secret to the helper $q_i$'s. If the $i$th bit of the message is equal to `0`, then $p$ sends the message "`0`" to the process $q_i$. If the $i$th bit of the message is `1`, $p$ does nothing. Figure 3-3 shows this step. Note that as a result of receiving these `0` bits, $q_1$ and $q_4$ have changed labels! Their labels floated up from $\{\}$ to $\{t\}$, as the kernel accounts for how information flowed in the system.

In the last step (Figure 3-4), the $q_i$ processes wait for a predefined time limit before giving

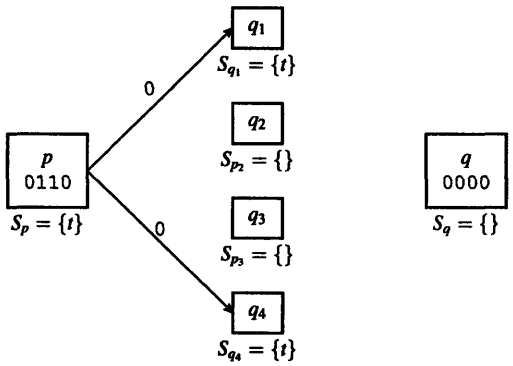


Figure 3-3: $p$ sends a "0" to $p_i$ if the $i$th bit of the message is 0.

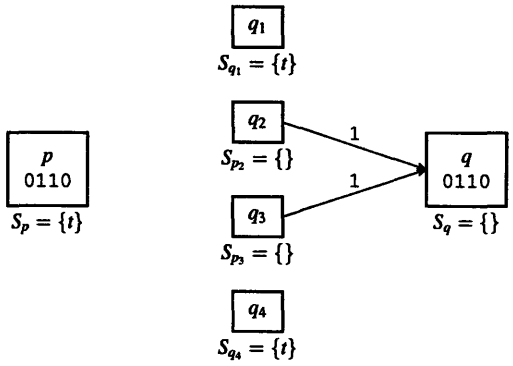


Figure 3-4: If $p_i$ did not receive a "0" before the timeout, it assumes an implicit "1" and writes "1" to $q$ at position $i$.

up. At the timeout, those who have not received messages ($q_2$ and $q_3$) write a 1 to the $q$, at the bit position that corresponds to their process ID. So $q_2$ writes a 1 at bit position 2, and $q_3$ writes a 1 at bit position 3. Note that $q_1$ and $q_4$ do not write to $q$, nor could they without affecting $q$'s label. Now, $q$ has the exact secret, copied bit-for-bit from $p$. This example shows 4 bits of data leak, but by forking $n$ processes, $p$ and $q$ can leak $n$ bits per timeout period. Because Asbestos's *event process* abstraction makes forking very fast, this channel on Asbestos can leak kilobits of data per second.

What went wrong? Denning's paper from 1976 [17] identifies the issue:

> There is one intrinsic problem with dynamic updating mechanisms: a change in an object's class may remove that object from the purview of a user whose clearance no longer permits access to the object. The class-change event can thereby be used to leak information, e.g., by removing from the user's purview a file meaning "0."

Here, processes $q_1$ and $q_4$ disappeared from $q$'s view. Their absence means the first and the fourth bit stay at 0 and therefore reflect the original secret.

Consider the behaviors of the $q_i$ processes to see why this particular attack would fail against the Flume system. At the step depicted in Figure 3-3, the $q_i$'s must each make a decision: should they change their labels to $S_{q_i} = \{\}$, or should they leave their labels as is? If $q_i$ changes, then it will receive messages from $p$, but it won't be able to write to $q$. If $q_i$ does not change, then it will never receive a message from $p$. Thus, its decision to write a 1 or not write at all has nothing to do with $p$'s message, and only to do with whether or not it decided to change labels, which it must do *before* receiving word from $p$. Thus, Flume appears secure against attacks that succeeds in Asbestos's floating label system. The next two chapters seek a formal proof of these intuitions.

## 3.5 Summary

This chapter informally specified the Flume Model for operating-systems level decentralized information flow control. The emphasis was on a labeling system that allows the kernel to track data throughout a system (whether for security or integrity guarantees) while assigning certain processes (declassifiers and endorsers) the privileges to legislate security policies. As in HiStar's Model, a key idea in Flume is that processes must set their labels explicitly, rather than labels floating dynamically as messages arrive. An important example shows the advantages of Flume's approach.

# Chapter 4

# The Formal Flume Model

In the previous chapter, an example attack on an Asbestos-like system showed an inherent weakness in the "floating" style of labels: with a careful process arrangement, an attacker can leak many bits of information by selectively sending or withholding communication. The same attack appeared to fail against the Flume model, but no formal reasoning proved that other attacks could not succeed. This chapter and the next seek a formal separation between the Asbestos style of "floating" labels and the Flume style of "explicitly specified" labels. The ultimate goal is to prove that Flume exhibits *non-interference* properties: for example, that processes with empty ownership and whose secrecy label contains $t$ cannot in any way alter the execution of those processes with empty labels. Such a non-interference result requires a formal model of Flume, which we build up here. Chapter 5 provides the proof that the Flume Model meets a standard definition of non-interference with high probability.

We present a formal model for the Flume System in the Communicating Sequential Processes (CSP) process algebra [46], with timing extensions [83]. The model captures a kernel, a system call interface, and arbitrary user processes that can interact via the system call interface. The model expresses processes communicating with one other over IPC, changing labels, allocating tags, and forking new processes. Formal techniques can then show that the system call API itself cannot be exploited to leak information (as in Section 3.4's attack on Asbestos). The model does not capture CPU, cache, memory, network or disk usage. Therefore, it is powerless to disprove the existence of covert channels that modulate CPU, cache, memory, network or disk usage to communicate data from one process to another.

The Flume CSP model provides a state machine description of the kernel, that if implemented accurately has useful security properties (see Chapter 5). Thus, this model functions as a high-level design document for kernel implementers, dictating which kernel details are safe to expose to user-level applications, where I/O can safely happen, which return codes from system calls to provide, etc. It leaves many details—like how to manage hardware—unspecified.

Figure 4-1 depicts the Flume model organization. At a high level, the model splits each Unix-like process running on a system (e.g., a Web server or text editor) into two logical components: a user-space half (e.g., $U_i$ and $U_j$) which can take almost any form, and a kernel-space half which behaves according to a strict state machine (e.g., $i:K$ and $j:K$). The user-space half of a process

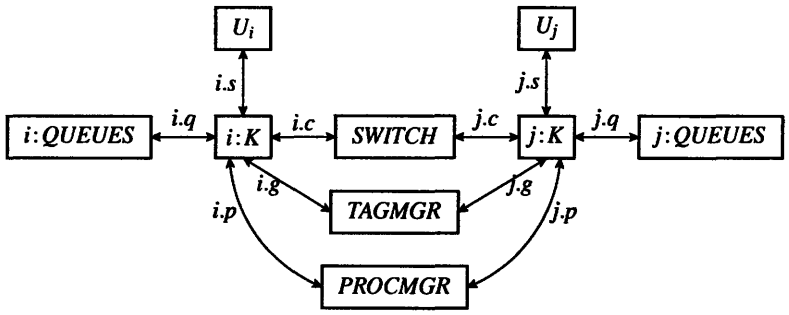


Figure 4-1: Two user-space processes, $U_i$ and $U_j$, in the CSP model for Flume. $i : K$ and $j : K$ are the kernel halves of these two processes (respectively), *TAGMGR* is the process that manages the global set of tags and associated privileges, *PROCMGR* manages the process ID space, and *SWITCH* enables all user-visible interprocess communication. Arrows denote CSP communication channels.

can communicate to its kernel half, and to other user-space processes via a system call interface. This interface takes the form of a CSP channel between the two processes (e.g. $i.s$ and $j.s$). Inside the kernel, the kernel-halves of processes can communicate with one another to deliver IPCs initiated at user-space. Also inside the kernel, a global process (*TAGMGR*) manages the circulation of tags ($\mathcal{T}$), and globally-shared privileges ($\hat{O}$); another global process (*PROCMGR*) manages the process ID space. The process *SWITCH* is involved with communication between user-level processes. The remainder of this chapter seeks to fill out the details of this diagram, first by reviewing CSP basics, and then by explaining details specific to Flume.

## 4.1 CSP Basics

Communicating Sequential Processes (CSP) is a process algebra useful in specifying systems as a set of parallel state machines that sometimes synchronize on events. We offer a brief review of it here, taking heavily from Hoare's book [46]. Among the most basic CSP examples is Hoare's vending machine:

$$VMS = in25 \rightarrow choc \rightarrow VMS$$

This vending machine waits for the event *in25*, which corresponds to the input of a quarter into the machine. Next, it accepts the event *choc*, which corresponds to a chocolate falling out of the machine. Then it returns to the original state, with a recursive call to itself. The basic operator at use here is the prefix operator. If $x$ is an event, and $P$ is a process, then $(x \rightarrow P)$, pronounced "$x$ then $P$" represents a process that engages in event $x$ then behaves like process $P$. For a process $P$, the notation $\alpha P$ describes the "alphabet" of $P$. It is a set of all of the events that $P$ is ever willing to engage in. For example, $\alpha VMS = \{in25, choc\}$.

For any CSP process $P$, we can discuss the "trace" of events that $P$ will accept. For the *VMS*

example, various traces include:

$$\langle\rangle$$
$$\langle in25 \rangle$$
$$\langle in25, choc \rangle$$
$$\langle in25, choc, in25, choc, in25 \rangle$$

The next important operator is "choice," denoted by "|". If $x$ and $y$ are distinct events, then:

$$(x \rightarrow P \mid y \rightarrow Q)$$

denotes a process that accepts $x$ and then behaves like $P$ or accepts $y$ and then behaves like $Q$. For example, a new vending machine can accept either a coin and output a chocolate, or a bill and output an ice cream cone:

$$VMS2 = (bill \rightarrow cone \rightarrow VMS2 \mid in25 \rightarrow choc \rightarrow VMS2)$$

CSP offers more general choice function (for choosing between many inputs succinctly), but the Flume model only requires simple choice.

Related to simple choice are "internal nondeterministic choice" and "external nondeterministic choice," denoted "$\sqcap$" and "$\Box$" respectively. In simple choice, the machine reacts exactly to events it fields from the machine's user. In nondeterministic choice, the machine behaves unpredictably from the perspective of the user, maybe because the machine's description is underspecified, or maybe because the machine is picking from a random number generator. For instance, a change machine might return coins in any order, depending on how the machine was last serviced:

$$CHNG = (in25 \rightarrow (out10 \rightarrow out10 \rightarrow out5 \rightarrow CHNG \sqcap \\ out10 \rightarrow out5 \rightarrow out10 \rightarrow CHNG))$$

That is, the machine takes as input a quarter, and returns two dimes and a nickel in one of two orderings. The "external nondeterministic choice" operator has slightly different semantics, but does not appear in Flume's model.

CSP provides useful predefined processes like *STOP*, the process that accepts no events, and *SKIP*, the process that shows a successful termination and then behaves like *STOP*. Other processes like *DIV*, *RUN* and *CHAOS* are standard in the literature, but are not required here.

The next class of operators relate to parallelism. The notation:

$$P \underset{A}{\parallel} Q$$

denotes $P$ running in parallel with $Q$, synchronizing on events in $A$.[1] Meaning, a stream of

[1]Parallelism differs between Hoare's original CSP formulation and more modern formulations, like Schneider's. We use Schneider's "interface parallelism" in this model.

incoming events can be arbitrarily assigned to either $P$ or $Q$, assuming those events are not in $A$. However, for events in $A$, both $P$ and $Q$ must accept them in synchrony. As an example, consider the vending machine and the change machine running in parallel, synchronizing on the event *in25*:

$$FREELUNCH = VMS \underset{\{in25\}}{\parallel} CHNG$$

Possible traces for this new process are the various interleavings of the traces for the two component machines that agree on the event *in25*. For instance:

$$\langle in25, choc, out10, out10, out5, \ldots \rangle$$
$$\langle in25, out10, choc, out10, out5, \ldots \rangle$$
$$\langle in25, out10, out10, choc, out5, \ldots \rangle$$
$$\langle in25, out10, out10, out5, choc, \ldots \rangle$$
$$\langle in25, choc, out10, out5, out10, \ldots \rangle$$
$$\langle in25, out10, choc, out5, out10, \ldots \rangle$$
$$\langle in25, out10, out5, choc, out10, \ldots \rangle$$
$$\langle in25, out10, out5, out10, choc, \ldots \rangle$$

are possible execution paths for *FREELUNCH*.

Another variation on parallel composition is arbitrary interleaving, denoted: $P \,|||\, Q$. In interleaving, $P$ and $Q$ never synchronize, operating independently of one another. $P \,|||\, Q$ is therefore equivalent to $P \parallel_{\{\}} Q$, which means $P$ and $Q$ run in parallel and synchronize on the empty set.

Processes that run in parallel can communicate with one another over *channels*. A typical channel $c$ can carry various values $v$, denoted $c.v$. The sending process accepts the event $c!v$ while the receiving process accepts the event $c?x$, after which step $x$ is set equal to $v$. Communication on a channel only works if the left process is in the send state and the right process is in the receive state at the same time. If one process is at the communicative state and the other is not, the ready process waits until its partner becomes ready. In a slight deviation from Hoare's semantics, channels here are bidirectional: messages can travel independently in either direction across a channel. The Flume model uses channels extensively.

The next important CSP feature is "concealment" or "hiding." For a process $P$ and a set of symbols $C$, the process $P \backslash C$ is $P$ with symbols in $C$ hidden or concealed. The events in $C$ then become internal transitions, that can happen without other processes being able to observe them. Concealment can induce *divergence*—an infinite sequence of internal transitions. For instance, the process $P = (c \rightarrow P) \backslash \{c\}$ diverges immediately, never to be useful again. The use of concealment in the Flume model is careful never to induce divergence in this manner.

Concealment enables subroutines (or "subordination" in Hoare's terminology). For two process $P$ and $Q$ such that $\alpha P \subseteq \alpha Q$, the new process $P \,/\!/\, Q$ is defined as $(P \parallel Q) \backslash \alpha P$. This means that the subroutine $P$ is available within $Q$, but not visible to the outside world. The notation $p : P \,/\!/\, Q$ means a particular instance $p$ of the subroutine $P$ is available in $Q$. Then

an event such as $p!x?y$ within $Q$ means that $Q$ is calling subroutine $p$ with argument $x$, and that the return value is placed into $y$. Within $P$, the event $?x$ means receive the argument $x$ from the caller, and the event $!y$ means return the result $y$ to the caller.

A final important language feature is "renaming." Given a process $P$, the notation $i:P$ means a "renaming" of $P$ with all events prefixed by $i$. That is, if the event $c!v$ appears in $P$, then the event $i.c!v$ appears in $i:P$, where $i.c$ is the channel $c$ that's been renamed to $i.c$. Thus, for any $i \neq j$, the alphabets of $i:P$ and $j:P$ are disjoint: $\alpha(i:P) \cap \alpha(j:P) = \{\}$. This concludes our whirlwind tour of CSP features. We refer the reader to Hoare's [46], Schneider's [92] and Roscoe's [85] books for many more details.

## 4.2 System Call Interface

We now return to our definition of the Flume CSP model. At a high level, user-level processes communicate with the kernel and each either through a system-call interface. Each user-level process $U_i$ has access to the following calls over its channel $i.s$:

- $t \leftarrow$ create_tag(which)

  Allocate a new tag ($t$), and depending on the parameter *which*, make the associated capabilities for that $t$ globally accessible. Thus, *which* can be one of None, Remove or Add. For Remove, add $t^-$ to $\hat{O}$, essentially granting it to all other processes; for Add, add $t^+$ to $\hat{O}$. *which* cannot specify both Remove and Add at once.

- rc $\leftarrow$ change_label(which, $L$)

  Change the process's *which* label to $L$. Return Ok on success and Error on failure. *which* can be either Secrecy or Integrity.

- $L \leftarrow$ get_label(which)

  Read this process's own label out of the kernel's data structures. *which* can be either Secrecy or Integrity, controlling which label is read.

- $O \leftarrow$ get_caps()

  Read this process's ownership set out of the kernel's data structures.

- send($j$, msg, $X$)

  Send message *msg* and capabilities $X$ to process $j$. Report success if the sender owns the capabilities contained in $X$ and false otherwise. Thus, success is reported even if the message send failed due to label checks.

- (msg, $X$) $\leftarrow$ recv($j$)

  Receive message *msg* and capabilities $X$ from process $j$. Block until a message is ready.

- $Y \leftarrow$ select$(t, X)$

  Given a set of process indices $X$, return a set $Y \subseteq X$. For all $j \in Y$, calling recv$(j)$ will yield immediate results. This call will block until $Y$ is non-empty, or until $t$ clock ticks expire.

- $j \leftarrow$ fork()

  Fork the current process; yield a process $j$. fork returns $j$ in the parent process and 0 in the child process.

- $i \leftarrow$ getpid()

  Return $i$, the ID of the current process.

- drop_caps$(X)$

  Set $O_i \leftarrow O_i - X$.

The Flume model places no restrictions on what the user portions of processes can do other than: (1) such processes cannot communicate with each other; and (2) they can only communicate with the kernel via the proscribed system call interface. Formally, let $C_i = \{c \mid (c.v) \in \alpha U_i\}$ be the set of channels that $U_i$ has. For instance, $i.s \in C_i$ where $i.s$ is the channel that process $i$ uses to make system calls into the kernel. The first requirement on $U_i$ is that for all $j \neq i$, $C_i \cap C_j = \{\}$. That is, no process $U_i$ can communicate directly with another process $U_j$. Also, process $i$ cannot tamper with the system call interface of any other process $j$, meaning $j.s \notin C_i$ for all $j$, $j \neq i$. Finally, each kernel process $j : K$ has four other channels, $\{j.b, j.g, j.c, j.p\}$, all discussed below. No user process can access any of these channels directly. That is, for all $i, j \in \mathcal{P}$:

$$\{j.b., j.g., j.c, j.p\} \cap C_i = \{\}$$

Of course, $C_i$ is not empty. For all $i$, $i.s \in C_i$, where $i.s$ is $U_i$'s dedicated channel for sending system calls to the kernel and receiving replies. $C_i$ can also contain channels from the process $U_i$ to itself, but otherwise is empty.

## 4.3 Kernel Processes

For each user process $U_i$, there is an instantiation of the kernel process $K$ that obeys a strict state machine. We apply CSP's standard technique for "relabeling" the interior states of a process: $U_i$'s kernel half is denoted $i : K$. Because $i : K$ and $j : K$ have different alphabets for $i \neq j$, their operations cannot interfere, and thus $U_i$ remains isolated from $U_j$. Each process $i : K$ will take on a state configuration based upon the labels of the corresponding user process $U_i$. We use $K_{S,I,O}$ to denote a process with secrecy label $S \subseteq \mathcal{T}$, integrity label $I \subseteq \mathcal{T}$, and ownership of capabilities given by $O$.

At a high level, a kernel process $K$ starts idle, then springs to life once receiving an activation message (ultimately because another process spawned it). Once active, it receives either system

calls from its user half, or internal messages from other kernel processes on the system. It eventually dies when the user process exits. In CSP notation:

$$K = b?(S, I, O) \rightarrow K_{S,I,O}$$

$b$ is the channel that $K$ listens on for its "birth" message. It expects arguments of the form $(S, I, O)$, to instruct it which labels and capabilities to start its execution with. Subsequently, $K_{S,I,O}$ handles the meat of the kernel process's duties:

$$K_{S,I,O} = SYSCALL_{S,I,O} \mid INTRECV_{S,I,O}$$

where *SYSCALL* is a subprocess tasked with handling all system calls, and *INTRECV* is the *internal receiving* sub-process, tasked with receiving internal messages from other kernel processes.

For any process ID $i$, the subprocess $i{:}K_{S,I,O}$ handles system calls by listening for incoming messages from $U_i$ along a shared channel $i.s$. In the definition of $K_{S,I,O}$, each system call gets its own dedicated subprocess:

$$\begin{aligned} SYSCALL_{S,I,O} = {} & NEWTAG_{S,I,O} \mid \\ & CHANGELABEL_{S,I,O} \mid \\ & READMYLABEL_{S,I,O} \mid \\ & READMYCAPS_{S,I,O} \mid \\ & DROPCAPS_{S,I,O} \mid \\ & SEND_{S,I,O} \mid \\ & RECV_{S,I,O} \mid \\ & SELECT_{S,I,O} \mid \\ & FORK_{S,I,O} \mid \\ & GETPID_{S,I,O} \mid \\ & EXIT_{S,I,O} \end{aligned}$$

Section 4.5 presents all of these subprocesses in more detail.

## 4.4 Process Alphabets

In the next chapter, we will prove properties about the system, in particular, that messages between "high processes" (those that have a specified tag in their secrecy label) do not influence the activity of "low processes." The standard formula for such proofs is to split the system's alphabet into two disjoint sets: "high" symbols, those that the secret influences; and "low" symbols, those that should not be affected by the secret. We must provide the appropriate alphabets for these processes so that any symbol in the model unambiguously belongs to one set or the other.

Consider some examples. Take process $U_i$ with secrecy label $S_i = \{t\}$ and $I_i = \{\}$. When $U_i$ issues a system call (say create_tag(Add)) to its kernel half $i : K$, the trace for $U_i$ is of the

form

$$\langle \ldots, i.s!(\mathsf{create\_tag}, \mathsf{Add}), \ldots \rangle$$

and the trace for $i:K$ is of the form

$$\langle \ldots, i.s?(\mathsf{create\_tag}, \mathsf{Add}), \ldots \rangle$$

That is, $U_i$ is sending a message (create_tag, Add) on the channel $i.s$, and $i:K$ is receiving it. The problem, however, is that looking at these traces does not capture the fact that $U_i$'s secrecy label contains $t$ and therefore that $U_i$ is in a "high" state in which it should not affect low processes. Such a shortcoming does not inhibit the accuracy of the model, but it does inhibit the proof of non-interference in Chapter 5.

A solution to the problem is simply to include a process's labels in the messages it sends. That is, once $U_i$ has a secrecy label of $S = \{t\}$, its kernel process should be in a state such as $K_{\{t\},\{\},\{\}}$. When a kernel process is in this state, it will only receive system calls of the form $i.s?(\{t\}, \{\}, \mathsf{create\_tag}, \mathsf{Add})$. Thus, $U_i$ must now send system calls in the form: $i.s!(\{t\}, \{\}, \mathsf{create\_tag}, \mathsf{Add})$. Of course, this message format requires $U_i$ to know its current $S$ and $I$ labels, but because processes must request label changes explicitly, the user portions can keep track of what its current labels are.

Because messages of the form $c!(S, I, \ldots)$ and $c?(S, I, \ldots)$, are so common (where $c$ is an arbitrary channel) we invent new notation:

$$c \underset{S,I}{\curlyvee} (a_1, \ldots, a_n) \triangleq c!(S, I, a_1, \ldots, a_n)$$

$$c \underset{S,I}{\curlywedge} (a_1, \ldots, a_n) \triangleq c?(S, I, a_1, \ldots, a_n)$$

In the context of a kernel process $K_{S,I,O}$, we need not specify $S$ and $I$ explicitly; they are inferred from the kernel's state. That is, when appearing inside a process $K_{S,I,O}$, $c\curlyvee$ is defined:

$$c\curlyvee (a_1, \ldots, a_n) \triangleq c!(S, I, a_1, \ldots, a_n)$$

And similarly for $c\curlywedge(\cdots)$.

## 4.5 System Calls

We now enumerate the individual system calls. The first system call subprocess handles a user process's request for new tags. Much of this system call is handled by the global tag manager *TAGMGR*. Note that after tag allocation, the kernel process always transitions to a different state, reflecting the new privilege or privileges it acquired for tag $t$. The definition of *TAGMGR* below

guarantees that $O_{\text{new}}$ is non-empty.

$$\begin{aligned} \mathit{NEWTAG}_{S,I,O} = (& s \curlywedge (\mathsf{create\_tag}, w) \rightarrow \\ & g!(\mathsf{create\_tag}, w)?(t, O_{\text{new}}) \rightarrow \\ & s \curlyvee t \rightarrow \\ & K_{S,I,O \cup O_{\text{new}}}) \end{aligned}$$

We split the *CHANGELABEL* subprocess into two cases, the first for changes to secrecy labels, and the second for changes to integrity labels:

$$\mathit{CHANGELABEL}_{S,I,O} = \mathit{S\text{-}CHANGELABEL}_{S,I,O} \mid \mathit{I\text{-}CHANGELABEL}_{S,I,O}$$

Where:

$$\begin{aligned} \mathit{S\text{-}CHANGELABEL}_{S,I,O} = (& \mathit{chk} : \mathit{CHECK}_{S,I,O} /\!/ \\ & (s \curlywedge (\mathsf{change\_label}, \mathsf{Secrecy}, S') \rightarrow \\ & \mathit{chk} \curlyvee (S, S') \curlywedge r \rightarrow \\ & \textbf{if } r \\ & \textbf{then } s \curlyvee \mathsf{Ok} \rightarrow K_{S',I,O} \\ & \textbf{else } s \curlyvee \mathsf{Error} \rightarrow K_{S,I,O})) \end{aligned}$$

And:

$$\begin{aligned} \mathit{I\text{-}CHANGELABEL}_{S,I,O} = (& \mathit{chk} : \mathit{CHECK}_{S,I,O} /\!/ \\ & (s \curlywedge (\mathsf{change\_label}, \mathsf{Integrity}, I') \rightarrow \\ & \mathit{chk} \curlyvee (I, I') \curlywedge r \rightarrow \\ & \textbf{if } r \\ & \textbf{then } s \curlyvee \mathsf{Ok} \rightarrow K_{S,I',O} \\ & \textbf{else } s \curlyvee \mathsf{Error} \rightarrow K_{S,I,O})) \end{aligned}$$

In both cases, the user process specifies a new label, and the *CHECK* subroutine determines if that label change is valid. In the success case, the kernel process transitions to a new state, reflecting the new labels. In the failure case, the kernel process remains in the same state. The *CHECK* process computes the validity of the label change based on the process's current capa-

bilities, and the global capabilities held by all processes:

$$\begin{aligned} CHECK_{S,I,O} = \lambda\,(L, L') \rightarrow \\ & g!(\textsf{check-}, L - L' - O^-) \rightarrow \\ & g?r \rightarrow \\ & g!(\textsf{check+}, L' - L - O^+) \rightarrow \\ & g?a \rightarrow \\ & \curlyvee\,(r \wedge a) \rightarrow \\ & CHECK_{S,I,O} \end{aligned}$$

As we will see below, the global tag register replies True to (check-, $L$) iff $L \subseteq \hat{O}^-$, and replies True to (check+, $L$) iff $L \subseteq \hat{O}^+$. Thus, we have that the user process can only change from label $L$ to $L'$ if it can subtract all tags in $L - L'$ and add all tags in $L' - L$, either by its own capabilities or those globally owned (see Definition 2 in Section 3.3).

The user half of a process can call into the kernel state to read its own $S$ or $I$ label, or to determine which capabilities it owns. These calls are handled simply by the following subprocesses:

$$\begin{aligned} READMYLABEL_{S,I,O} = (s\,\lambda\,(\textsf{get\_label}, \textsf{Secrecy}) \rightarrow s\curlyvee S \rightarrow K_{S,I,O} \mid \\ s\,\lambda\,(\textsf{get\_label}, \textsf{Integrity}) \rightarrow s\curlyvee I \rightarrow K_{S,I,O}) \end{aligned}$$

And similarly for reading capabilities:

$$READMYCAPS_{S,I,O} = (s\,\lambda\,(\textsf{get\_caps}) \rightarrow s\curlyvee O \rightarrow K_{S,I,O})$$

If a process can accumulate privileges with calls to *NEWTAG*, it can later discard them with calls to *DROPCAPS*:

$$DROPCAPS_{S,I,O} = (s\,\lambda\,(\textsf{drop\_caps}, X) \rightarrow K_{S,I,O-X})$$

On a successful drop of capabilities, the process transitions to a new kernel state, reflecting the reduced ownership set.

The next process to cover is forking. Recall that each active task $i$ on the system has two components: a user component $U_i$ and a kernel component $i : K$. The Flume model does not capture what happens to $U_i$ when it calls fork, but an implementation of the model should provide a mechanism for $U_i$ to copy its address space, and configure the execution environment in the child. The model does capture the kernel-specific behavior in fork as follows:

$$
\begin{aligned}
FORK_{S,I,O} = (s \curlywedge (\mathsf{fork}) \rightarrow \\
p \curlyvee (\mathsf{fork}, O) \rightarrow p?j \rightarrow \\
s \curlyvee j \rightarrow \\
K_{S,I,O})
\end{aligned}
$$

Recall that $i.p$ is a channel from the $i$th kernel process to the process manager in the kernel, *PROCMGR*.

The final three processes are straightforward:

$$
\begin{aligned}
GETPID_{S,I,O} = (s \curlywedge (\mathsf{getpid}) \rightarrow \\
p!(\mathsf{getpid}) \rightarrow p?i \rightarrow \\
s \curlyvee i \rightarrow \\
K_{S,I,O})
\end{aligned}
$$

User processes issue an exit system call as they terminate:

$$
\begin{aligned}
EXIT_{S,I,O} = (s \curlywedge (\mathsf{exit}) \rightarrow \\
q!(\mathsf{clear}) \rightarrow \\
p!(\mathsf{exit}) \rightarrow \\
SKIP)
\end{aligned}
$$

Once a process with a given ID has run and exited, its ID is retired, never to be used again. An alternative implementation is for the last transition of *EXIT* to transition back to a starting state, but such a transition complicates the proof of non-interference in Chapter 5.

## 4.6 Communication

The communication subprocesses are the crux of the Flume CSP model. The require care to ensure that subtle state transition in high processes do not result in observable behavior by low processes. At the same time, they must make a concerted effort deliver messages, so that the system is useful.

The beginning of a message delivery sequence is the process *SEND*, invoked when $U_i$ wishes to send a message to $U_j$. To make *SEND* succeed as often as possible, the kernel attempts to shrink the process's $S$ label and to grow its integrity label $I$ as much as allowed by the process's privileges. The actual message send itself goes through the switchboard process *SWITCH* via channel $i.c$. The switchboard then sends the message onto the destination $j$.

$$
\begin{aligned}
SEND_{S,I,O} = (s \curlywedge (\mathsf{send}, j, X, m) \ &\rightarrow \\
&\textbf{if } X \subseteq O \\
&\textbf{then } g!(\mathsf{dual\_privs}, O) \rightarrow g?D \rightarrow \\
&\qquad c!(S - D, I \cup D, j, X, m) \ \rightarrow \\
&\qquad s \curlyvee \mathsf{Ok} \ \rightarrow \ K_{S,I,O} \\
&\textbf{else } s \curlyvee \mathsf{Error} \ \rightarrow \ K_{S,I,O})
\end{aligned}
$$

The process *SWITCH* listens on the other side of the receive channel $i.c$. It inputs messages of the form $i.c?(S, I, j, X, m)$ and forwards them to the process $j : K$ as $j.c!(S, I, j, X, m)$:

$$
\begin{aligned}
SWITCH \ = \ |_{\forall i}(i.c?(S, I, j, X, m) \ &\rightarrow \\
&(j.c!(S, I, i, X, m) \ \rightarrow SKIP \ |||\ SWITCH))
\end{aligned}
$$

The *SWITCH* process sends messages in parallel with the next receive operation. This parallelism avoids deadlocking the system if the receiving process has exited, not yet started, or is waiting to send a message. In other words, the *SWITCH* process is always willing to receive a new message, delegating potentially-blocking send operations to an asynchronous child process.

Once the message leaves the switch, the receiver process handles it with its *INTRECV* subprocess. After performing the label checks given by Definition 3 in Section 3.3.1, this process enqueues the incoming message for later retrieval:

$$
\begin{aligned}
INTRECV_{S,I,O} \ = c?(S_{in}, I_{in}, j, X, m) \ &\rightarrow \\
&g!(\mathsf{dual\_privs}, O) \rightarrow g?D \rightarrow \\
&\textbf{if } (S_{\text{in}} \subseteq S \cup D) \ \wedge \ (I - D \subseteq I_{\text{in}}) \\
&\textbf{then } q!(\mathsf{enqueue}, (X, m)) \rightarrow K_{S,I,O} \\
&\textbf{else } K_{S,I,O}
\end{aligned}
$$

The final link in the chain is the actual message delivery at user space. For a user-space process to receive a message, it calls into the kernel, asking it to dequeue and deliver any waiting messages. Receiving also updates the process's ownership, to reflect new capabilities it gained.

$$
\begin{aligned}
RECV_{S,I,O} = (s \curlywedge (\mathsf{recv}, j) \ &\rightarrow \\
&q!(\mathsf{dequeue}, j) \rightarrow q?(X, m) \ \rightarrow \\
&s \curlyvee m \ \rightarrow \\
&K_{S,I,O \cup X})
\end{aligned}
$$

The last subprocess of the group is one that allows a user program to wait for the first avail-

able receive channel to become readable:

$$
\begin{aligned}
SELECT_{S,I,O} = (s \curlywedge (\mathsf{select}, t, A) \rightarrow \\
(\mu X \bullet (q!(\mathsf{select}, A) \rightarrow q?B \rightarrow \\
\textbf{if } B = \{\} \\
\textbf{then } INTRECV^*_{S,I,O} ; X \\
\textbf{else } s \curlyvee B \rightarrow K_{S,I,O}) \\
\triangle_t (s \curlyvee \{\} \rightarrow K_{S,I,O}))
\end{aligned}
$$

The "timed interrupt operator" $\triangle_t$ [92] interrupts the selection process after $t$ clicks of the clock and outputs an empty result set. Also, *SELECT* calls subprocess *INTRECV**, which behaves mostly like *INTRECV*, except it keeps receiving until an admissible message arrives:

$$
\begin{aligned}
INTRECV^*_{S,I,O} = c?(S_{in}, I_{in}, j, X, m) \rightarrow \\
g!(\mathsf{dual\_privs}, O) \rightarrow g?D \rightarrow \\
\textbf{if } (S_{\text{in}} \subseteq S \cup D) \wedge (I - D \subseteq I_{\text{in}}) \\
\textbf{then } q!(\mathsf{enqueue}, (X, m)) \rightarrow SKIP \\
\textbf{else } INTRECV^*_{S,I,O}
\end{aligned}
$$

## 4.7 Helper Processes

It now remains to fill in the details for the helper processes that the various $K_{S,I,O}$ processes call upon. They are: *TAGMGR*, which manages all global tag allocation and global capabilities; *QUEUES*, which manages receive message queues, one per process; and finally *PROGMGR*, which manages process creation, deletion, etc.

### 4.7.1 The Tag Manager (*TAGMGR*)

The tag manager maintains a global universe of tags $\mathcal{T}$, keeping track of the global set of privileges available to all processes $\hat{O}$. It also tabulates which tags have already been allocated, so as never to reissue the same tag. The set $\hat{T}$ refers to those tags that were allocated in the past. Thus, its states are parameterized $TAGMGR_{\hat{O},\hat{T}}$. As the system starts, $\hat{O}$ and $\hat{T}$ are empty:

$$TAGMGR = TAGMGR_{\{\},\{\}}$$

Once active, the tag manager engages in the following calls:

$$\begin{aligned} TAGMGR_{\hat{O},\hat{T}} = {} & NEWTAG{+}_{\hat{O},\hat{T}} \mid \\ & NEWTAG{-}_{\hat{O},\hat{T}} \mid \\ & NEWTAG0_{\hat{O},\hat{T}} \mid \\ & DUALPRIVS_{\hat{O},\hat{T}} \mid \\ & CHECK{+}_{\hat{O},\hat{T}} \mid \\ & CHECK{-}_{\hat{O},\hat{T}} \end{aligned}$$

Many of these subprocesses will call upon a subroutine that randomly chooses an element from a given set. We define that subroutine here. Given a set $Y$:

$$CHOOSE_Y = ?(S, I) \rightarrow \bigsqcap_{y \in Y}(!y) \rightarrow STOP$$

That is, the subprocess *CHOOSE* nondeterministically picks an element $y$ from $Y$ and returns it to the caller. As we will see in Chapter 5, *CHOOSE*'s refinement (i.e., its instantiation) has an important impact on security. It can, and in some cases should, take into account the labels on the kernel process on whose behalf it operates.

The first set of calls involve allocating new tags, such as:

$$\begin{aligned} NEWTAG{+}_{\hat{O},\hat{T}} = choose : CHOOSE_{T-\hat{T}} \mathbin{/\!/} \; |_{\forall i}\big( & i.g?(\mathsf{create\_tag}, \mathsf{Add}) \rightarrow \\ & choose!(S, I)?t \rightarrow \\ & i.g!(t, \{t^-\}) \rightarrow \\ & TAGMGR_{\hat{O}\cup\{t^+\},\hat{T}\cup\{t\}}\big) \end{aligned}$$

That is, the subprocess *NEWTAG+* looks at all channels to all other processes ($\forall i \in \mathcal{P}$) and picks the first such $i$ that has input available. Here, it chooses a tag $t$ at random via *CHOOSE*, then returns that tag to the calling kernel process. It then services the next request in a different state, reflecting that fact that a new capability is available to all processes ($t^+$). Upon allocating a tag $t$, the tag manager updates its internal accounting so that it will not reallocate the same tag.

We next define *NEWTAG-* and *NEWTAG0* similarly:

$$\begin{aligned} NEWTAG{-}_{\hat{O},\hat{T}} = choose : CHOOSE_{T-\hat{T}} \mathbin{/\!/} \; |_{\forall i}\big( & i.g?(\mathsf{create\_tag}, \mathsf{Remove}) \rightarrow \\ & choose!(S, I)?t \rightarrow \\ & i.g!(t, \{t^+\}) \rightarrow \\ & TAGMGR_{\hat{O}\cup\{t^-\},\hat{T}\cup\{t\}}\big) \end{aligned}$$

And:

$$NEWTAG0_{\hat{O},\hat{T}} = choose : CHOOSE_{\mathcal{T}-\hat{T}} \;/\!/\; |_{\forall i}(i.g?(\mathsf{create\_tag}, \mathsf{None}) \rightarrow choose!(S,I)?t \rightarrow i.g!(t,\{t^-,t^+\}) \rightarrow TAGMGR_{\hat{O},\hat{T}\cup\{t\}})$$

The purpose of the *DUALPRIVS* subprocess is to augment a user process's ownership set $O$ with all of the globally-held privileges available in $\hat{O}$. That is, to return $\bar{O}_i = O_i \cup \hat{O}$ for a given process $i$. The challenge, however, is to do so without allowing a process to enumerate the contents of $\hat{O}$. To achieve both ends, we specialize the interface to *TAGMGR*. Given a set $O_i$, the tag manager process will return the set of tags that $U_i$ has dual privilege for. Since there are no tags $t$ such that $\{t^-,t^+\} \subseteq \hat{O}$, it follows that the process must own at least one privilege for $t$ to get dual privilege for it. Thus, the *DUALPRIVS* call will not alert any process to the existence of any tags it did not already know of:

$$DUALPRIVS_{\hat{O},\hat{T}} = |_{\forall i}(i.g?(\mathsf{dual\_privs}, O_i) \rightarrow i.g!((O_i^+ \cup \hat{O}^+) \cap (O_i^- \cup \hat{O}^-)) \rightarrow TAGMGR_{\hat{O},\hat{T}})$$

Finally, the behavior of *CHECK+* has already been hinted at. Recall this subprocess checks to see if the supplied set of tags is globally addable:

$$CHECK{+}_{\hat{O},\hat{T}} = |_{\forall i}(i.g?(\mathsf{check+}, L) \rightarrow (\textbf{if } L \subseteq \hat{O}^+ \textbf{ then } i.g!\mathsf{True} \textbf{ else } i.g!\mathsf{False}) \rightarrow TAGMGR_{\hat{O},\hat{T}})$$

And similarly:

$$CHECK{-}_{\hat{O},\hat{T}} = |_{\forall i}(i.g?(\mathsf{check-}, L) \rightarrow (\textbf{if } L \subseteq \hat{O}^- \textbf{ then } i.g!\mathsf{True} \textbf{ else } i.g!\mathsf{False}) \rightarrow TAGMGR_{\hat{O},\hat{T}})$$

### 4.7.2 The Process Manager (*PROCMGR*)

The main job of the process manager is to allocate process identifiers when kernel processes call fork. We assume a large space of process identifiers, $\mathcal{P}$. The process manager keeps track of subset $\hat{P} \subseteq \mathcal{P}$ to account for which of those processes identifiers are already in use. In then allocates from $\mathcal{P} - \hat{P}$.

$$\begin{aligned} PROCMGR_{\hat{P}} = {} & PM\text{-}FORK_{\hat{P}} \mid \\ & PM\text{-}GETPID_{\hat{P}} \mid \\ & PM\text{-}EXIT_{\hat{P}} \end{aligned}$$

To answer the fork operation, the process manager picks an unused process ID ($j$) for the child, gives birth to the child ($j : K$) with the message $j.b!(S, I, O)$, and returns child's process ID to the caller (parent):

$$\begin{aligned} PM\text{-}FORK_{\hat{P}} = choose : CHOOSE_{\mathcal{P}-\hat{P}} \;/\!/\; |_{\forall i}\big( & i.p?(S, I, \mathsf{fork}, O) \rightarrow \\ & choose!(S, I)?j \rightarrow \\ & j.b!(S, I, O) \rightarrow \\ & i.p!(j) \rightarrow \\ & PROGMGR_{\hat{P}\cup\{j\}}\big) \end{aligned}$$

Trivially:

$$PM\text{-}GETPID = |_{\forall i}\big(i.p?(\mathsf{getpid})!i \rightarrow PROCMGR\big)$$

Kernel processes notify the process manager of their exits. Of course, such notification would give it opportunity to update its accounting and to return the exiting process identifier into circulation. But, for now, it handles process exits as no-ops:

$$PM\text{-}EXIT = |_{\forall i}\big(i.p?(\mathsf{exit}) \rightarrow PROCMGR\big)$$

A final task for the process manager is to initialize the system, launching the first kernel process. This process runs with special process ID init, off-limits to other processes. Thus:

$$PROCMGR0 = \mathsf{init}.b!(\{\}, \mathcal{T}, \{\}, \{\}) \rightarrow PROCMGR_{\mathcal{P}-\{\mathsf{init}\}}$$

### 4.7.3 Per-process Queues (*QUEUES*)

Each kernel process $i : K$ needs it own set of queues, to handle messages received asynchronously from other processes. For convenience, we package up all of the queues in a single process $i$ : *QUEUES*, which $i : K$ can access in all of its various states. The channel $q$ serves communication between the queues and the kernel process. The building block of this process is a single *QUEUE* process, similar to that defined in Hoare's book. This process is parameterized by the value stored

in the queue, and of course the queue starts out empty:

$$QUEUE = QUEUE_{<>}$$

From here, we define state transitions:

$$\begin{aligned}
QUEUE_{\langle\rangle} &= (?(\mathsf{enqueue}, x) \rightarrow QUEUE_{\langle x\rangle} \mid \\
&\quad\ ?(\mathsf{select}, j)!\{\} \rightarrow QUEUE_{\langle\rangle}) \\
QUEUE_{\langle x\rangle^\frown s} &= (?(\mathsf{enqueue}, y) \rightarrow \mathsf{if}\ \#s + 1 < N_Q \\
&\qquad\qquad\qquad\qquad \mathsf{then}\ QUEUE_{\langle x\rangle^\frown s^\frown \langle y\rangle} \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ QUEUE_{\langle x\rangle^\frown s} \mid \\
&\qquad ?(\mathsf{dequeue})!x \rightarrow QUEUE_s \mid \\
&\qquad ?(\mathsf{select}, j)!\{j\} \rightarrow QUEUE_{\langle x\rangle^\frown s})
\end{aligned}$$

Note that these queues are bounded beneath $N_Q$ elements. Attempts to enqueue messages on filled queues result in dropped messages. The model combines many *QUEUE* subprocesses into a collection processes called *QUEUESET*:

$$QUEUESET = \|_{i \in \mathcal{P}}\ i{:}QUEUE$$

The process called *QUEUES* communicates with kernel processes. Recall that $i.q$ is the channel shared between $i{:}K$ and $i{:}QUEUES$:

$$\begin{aligned}
QUEUES = s :{}& QUEUESET \ /\!/\ sel : QSELECT_s \ /\!/\ \mu X \bullet \\
& (q?(\mathsf{enqueue}, j, m) \rightarrow s.j!(\mathsf{enqueue}, m) \rightarrow X \mid \\
& \ q?(\mathsf{dequeue}, j) \rightarrow s.j!(\mathsf{dequeue})?m \rightarrow q!m \rightarrow X \mid \\
& \ q?(\mathsf{select}, Y) \rightarrow sel!Y?Z \rightarrow q!Z \rightarrow X \mid \\
& \ q?(\mathsf{clear}) \rightarrow QUEUES)
\end{aligned}$$

Finally, the point of *QSELECT* is to determine which of the supplied queues have messages ready to receive. This process uses tail recursion to add to the variable $Z$ as readied queues are

found.

$$
\begin{aligned}
QSELECT_s = Z : VAR /\!/ ?Y \rightarrow \\
& Z := \{\}; \\
& (\mu X \bullet (\textbf{if } Y = \{\} \\
& \quad \textbf{then } (!Z \rightarrow QSELECT_s) \\
& \quad \textbf{else pick } j \in Y; \\
& \quad\quad Y := Y - \{j\}; \\
& \quad\quad (s.j!(\mathsf{select}, j) \rightarrow s.j?A \rightarrow \\
& \quad\quad (Z := Z \cup A; X)))
\end{aligned}
$$

## 4.8 High Level System Definition

The overall system *SYS* is an interleaving of all the processes specified. Consider some subset $J \subseteq \mathcal{P}$ of all possible process IDs. The user-half of the system, restricted to those processes in $J$, is:

$$UPROCS_J = |||_{j \in J} U_j$$

The kernel processes are:

$$KS_J = \|_{j \in J}\, j : \big((K \underset{\{j.q\}}{\|} QUEUES) \setminus \alpha QUEUES\big)$$

Adding in the helper process gives the complete kernel:

$$
\begin{aligned}
KERNEL1_J &= \big(KS_J \underset{\{j.c \mid j \in J\}}{\|} SWITCH\big) \setminus \alpha SWITCH \\
KERNEL2_J &= \big(KERNEL1_J \underset{\{j.g \mid j \in J\}}{\|} TAGMGR\big) \setminus \alpha TAGMGR \\
KERNEL_J &= \big(KERNEL2_J \underset{\{j.p \mid j \in J\}}{\|} PROCMGR0\big) \setminus \alpha PROCMGR0
\end{aligned}
$$

Finally:

$$SYS_J = UPROCS_J \underset{\{j.s \mid j \in J\}}{\|} KERNEL_J$$

Of course, the whole system is captured simply by $SYS_{\mathcal{P}}$.

This assembly of kernel process makes extensive use of the CSP hiding operator ("\"). That is, the combined process *SYS* does not show direct evidence of internal state transitions such as: communications between any $i : K$ and the switch; communications with the tag manager; communications with the process manager; etc. In fact the only events that remain visible are the workings of the user processes $U_i$ and their system calls given by $i.s\lambda$ and $i.s\curlyvee$. By implication, kernels that implement the Flume model should hide the system's inner workings from unprivileged users, but this is largely the case already. In practical terms, the CSP model for *SYS*

shows what a non-root Unix user might see if examining his processes with the `strace` utility.

## 4.9 Discussion

We have presented a particular CSP model that captures the Flume DIFC rules discussed at a high level in Chapter 3. Of course, this is not the only CSP model that might describe an interesting DIFC kernel. We briefly discuss the advantages and limitations of this approach.

**Limitations** The Flume DIFC model is a "monolithic" kernel design, in which the kernel is a large, hidden black box, and user-level processes have a large system call interface. Some modern approaches to kernel (e.g. the Exokernel [23] and the Infokernel [2]) design expose more of the inner workings of the kernel to give application developers more flexibility. However, in an information flow control setting, such an exposure is potentially dangerous. Imagine, in the Flume CSP model, that interactions between the process $i : K$ and the tag managers *TAGMGR* were not concealed with $\backslash \alpha TAGMGR$. Some process $i$ with secrecy $S_i = \{t\}$ and empty ownership issues the system call create_tag. Assume that $i : K$ makes a call to the tag manager over $i.g$, and then the tag manager makes some progress on allocating the new tag, halting right before $i.g!(t, \{t^+\})$. Then, another process $j$ with empty secrecy and empty ownership also tries to allocate a new tag. Now $j$ can observe that *NEWTAG* cannot proceed past $g!(\mathsf{create\_tag}, w)$, because the tag manager is not currently in a state in which it receives $g?(\mathsf{create\_tag}, w)$. Thus, $i$ can convey bits to $j$ via the tag manager's internal state machine. The simplest way to work around this problem is to conceal the inner workings of the kernel (as we have done). Another, more complicated solution, is to model more parallelism inside the kernel, so that the tag manager can serve both $i$ and $j$ concurrently, without them contending for resources (and therefore communicating bits).

Along similar lines, an important limitation is that the above model captures most of the kernel processes—like the $i : K$, the tag manager, the queues, and the process manager—as single-threaded processes. For instance, if the tag manager is responding to a request for $i.g.(\mathsf{create\_tag}, w)$, it cannot service $j.g.(\mathsf{create\_tag}, w)$ until it replies to $i.g.(\mathsf{create\_tag}, w)$. In practical implementations of this CSP model, such serialization might be a bottleneck for performance. As mentioned above, more parallelism internal to the kernel is possible, but would require explicit synchronization through locks, and more complexity overall.

The Flume CSP model obviously does not describe a full kernel: an implementation would have to fill in many pieces, including primitives for reliable interprocess communication and files (discussed in Chapter 6). In CSP terms, moving from a high-level model to an actual implementation is known as "refinement:" the behavior of the high-level model remains, while details unspecified in the model (such as nondeterminism in the *CHOOSE* operator) are better-specified. Of course, a real kernel also needs to interface with the CPU, memory, network, devices, storage, etc., and the model specifies none of these interactions. Unfortunately, the "refinement paradox" holds that even if a process exhibits non-interference, a refinement of that process might not [48]. Thus, even if a kernel faithfully implements the Flume CSP model, it might still be susceptible to information-leaking attacks.

**Advantages** Though the Flume CSP model does not automatically yield a leak-free implementation, the model still serves an important purpose — to prove that some implementation of the API might exist that does not leak information. The same cannot be said of a model for the Asbestos or the IX API: all systems that implement those models would be susceptible to large information flow control leaks (as seen in Section 3.4). To use a poker analogy, implementers of Asbestos or IX are "drawing dead" — even if they make their hand (achieve a good implementation), they will still lose to their opponent (the attacker) who had a better hand all along. Implementers of Flume at least have a fighting chance.

Relative to semantic models from the language community [114], the Flume model provides much more flexibility as to how the various $U_i$ might behave. The Flume model restricts these processes from accessing certain communication channel, but otherwise they can behave in any manner, and need not be type-checked. The innovation here relative to language-level models is to emulate the user/kernel split already at work in most operating systems.

Finally, the Flume model, relative to the kernel features it does model (e.g., IPC, forking, label creation, etc), is almost completely specified. The one process that uses nondeterminism is *CHOOSE*, and Chapter 5 provides more details about how this process should behave.

# Chapter 5

# Non-Interference

A mature definition in the literature for models like Flume's is *non-interference*. Informally:

> One group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see. [36]

That is, for an export-protection tag $t$, and a process $p$ running with $S_p = \{t\}$, a process $q$ running with $S_q = \{\}$ should have an execution path that is entirely independent of $p$'s. If $p$ could somehow influence $q$, then it could reveal to $q$ information tagged with $t$, which is against the high-level export-protection policy.

This chapter explores the non-interference properties of Flume's CSP model. Previous work by Ryan and Schneider [88] informs which definition of non-interference to apply (see Section 5.2). A proof that Flume fits the definition follows (see Section 5.4).

## 5.1 CSP Preliminaries

Before we can state our working definition of non-interference, we must define some more CSP preliminaries. First, a way to identify processes in states other than their initial states: the process $P/tr$ is $P$ advanced to the state after the trace $tr$ has occurred. Next, we often talk about the effects of "purging" certain events from traces and process states. The operator "$\restriction$" denotes *projection*. The trace $tr \restriction A$ is the trace $tr$ projected onto the set $A$, meaning all events not in $A$ are removed. For instance, if $A = \{a\}$, and $tr = \langle a, b, c, d, a, b, c \rangle$, then $tr \restriction A = \langle b, c, d, b, c \rangle$. For a set $C$, the set $C \restriction A$ is simply the intersection of the two.

A final topic, of great interest in the CSP literature, is process equivalence. In this thesis, we use the "stable failures" model, from Hoare's book [46] and later rephrased in Schneider's book [92] and Roscoe's book [85]. For a process $P$, the failures of $P$, written $\mathcal{SF}[\![P]\!]$, are defined as:

$$\mathcal{SF}[\![P]\!] = \{(s, X) \mid s \in \mathit{traces}(P) \ \wedge \ P/s \downarrow \ \wedge X \in \mathit{refusals}(P/s)\}$$

The traces of $P$ (denoted *traces*$(P)$) is the set of all traces accepted by the process $P$. The notation $Q \downarrow$ is a predicate that denotes the process $Q$ is "stable." Unstable states are those that transition internally, or those that diverge. For example, consider the process:

$$P_0 = (a \rightarrow STOP \sqcap b \rightarrow STOP)$$

$P_0$ begins at an unstable state, since it can make progress in either the left or right direction without accepting any input. However, once it makes its first internal transition, arriving at either $a \rightarrow STOP$ or $b \rightarrow STOP$, it becomes stable. A process that diverges, such as $(c \rightarrow P)\backslash\{c\}$, has no stable states. Conversely, stable states are those that can make no internal progress.

The refusals of $P$ (denoted *refusals*$(P)$) is a set of sets. A set $X$ is in *refusals*$(P)$ if and only if $P$ deadlocks when offered any event from $X$. For instance, consider the process $P_0$ above. We write that *refusals*$(P_0) = \{\{a\}, \{b\}\}$. That is, $P_0$ can nondeterministically choose the left branch, in which case it will only accept $\{a\}$ and will refuse $\{b\}$. On the other hand, if it nondeterministically chooses the left branch, it will accept $\{b\}$ and refuse $\{a\}$. Thus, due to nondeterminism, we write *refusals*$(P)$ as above, and *not* as the flattened union $\{a, b\}$. Applying a similar argument to all states of $P$, we can write:

$$\mathcal{SF}[\![P]\!] = \{(\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$$

In other words, the failures of $P$ captures which traces $P$ accepts, and which sets it refuses after accepting those traces.

In the stable failures model, two processes $P$ and $Q$ are deemed equivalent if and only if $\mathcal{SF}[\![P]\!] = \mathcal{SF}[\![Q]\!]$. Two projected processes $P \upharpoonright A$ and $Q \upharpoonright A$ are equivalent if and only if $\mathcal{SF}[\![P]\!] \upharpoonright A = \mathcal{SF}[\![Q]\!] \upharpoonright A$, where:

$$\mathcal{SF}[\![P]\!] \upharpoonright A = \{(tr \upharpoonright A, X \cap A) \mid (tr, X) \in \mathcal{SF}[\![P]\!]\}$$

and similarly for $Q$.

## 5.2 Definition

With these notational preliminaries in mind, a phrasing of non-interference [88] is as follows:

*Definition* 4 (Non-Interference for System $S$). For a CSP process $S$, and an alphabet of low symbols $LO \subseteq \alpha S$, the predicate $\mathsf{NI}_{LO}(S)$ is true iff:

$$\begin{aligned} \forall tr, tr' : traces(S).\ tr \approx_{LO} tr' \Rightarrow \\ ((S/tr) \upharpoonright LO = (S/tr') \upharpoonright LO) \end{aligned}$$

Where:

$$tr \approx_A tr' \Leftrightarrow tr \upharpoonright A = tr' \upharpoonright A$$

We say that the process $S$ exhibits *non-interference* with respect to the low alphabet $LO$ iff

$\mathsf{NI}_{LO}(S)$ is true.

In the stable failures model, the process equivalence relation

$$(S/tr) \upharpoonright LO \;=\; (S/tr') \upharpoonright LO$$

can be rewritten:

$$\mathcal{SF}[\![(S/tr)]\!] \upharpoonright LO \;=\; \mathcal{SF}[\![(S/tr')]\!] \upharpoonright LO$$

This definition considers all possible pairs of traces for $S$ that vary only by elements in the high alphabet (i.e., they are equal when projected to low). For each pair of traces, two experiments are considered: running $S$ over the elements in left trace, and running $S$ over the elements in the right trace. The two resulting processes must look equivalent from a "low" perspective. That is, they must accept all of the same traces (projected to low) and refuse all of the same refusal sets (projected to low).

### 5.2.1 Stability and Divergence

There are several complications. The first is the issue of whether or not the stable failures model is adequate. For instance, if a high process caused the kernel to diverge (i.e., *hang*), a low process could record such an occurrence on reboot, thereby leaking a bit (very slowly!) to low. By construction, the Flume kernel never diverges. One can check this property by examining each system call and verifying that only a finite number of internal events can occur before the process is ready to receive the next call. User-space process (e.g., $U_i$) can diverge, but their behavior matters little during a security analysis.

If divergence attacks were a practical concern, we could precisely capture divergent behavior with the more general Failures, Divergences, Infinite Traces (FDI) model [92]. We conjecture that Flume's non-interference results under the stable failures model also hold in the FDI model, but the proof mechanics are yet more complicated.

### 5.2.2 Time

The next complication involves time. The model for Flume does not fit in Hoare's original untimed CSP model, since the select system call requires an explicit timeout (via the $\triangle_t$ operator). Though Schneider develops a full notion of process equivalence in timed CSP [92], the mechanics are complex. Instead, we use a technique introduced by Ouaknine [74] and also suggested by Schneider [92]: convert our timed model into an untimed model with the introduction of the event *tock*, which represents a discrete unit of time's passage. In particular, Schneider provides the $\Psi$ function for mapping processes from timed CSP to discrete-event CSP with *tock*. For example:

$$\begin{aligned} \Psi(a \rightarrow Q) = \;& P_0 = a \rightarrow \Psi(Q) \\ & \Box\, tock \rightarrow P_0 \\ \Psi(WAIT\, n+1) \;=\; & tock \rightarrow \Psi(WAIT\, n) \end{aligned}$$

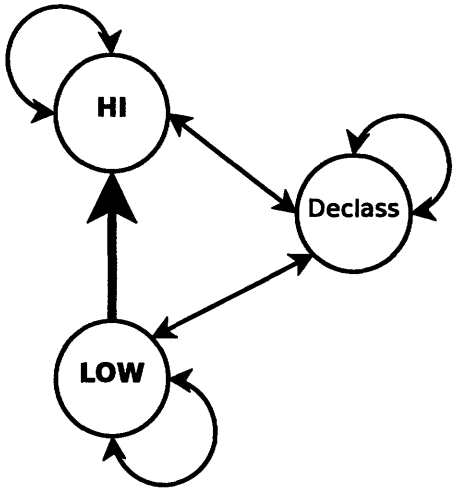


Figure 5-1: Intransitive Non-interference. Arrows depict allowed influence. All influences are allowed *except* high to low.

Without doing so explicitly, we assume that the $\Psi$ translation is applied to all states of the Flume model, and that the *tock* event is not hidden by any concealment operator.

### 5.2.3 Declassification

The third complication is *declassification*, or to use the terminology of the process-algebra literature, *intransitive non-interference*. That is, the system should allow certain flows of information from "high" processes to "low" processes, if that flow traverses the appropriate declassifier. Figure 5-1 provides a pictorial representation: the system allows low processes and the declassifier to influence all other processes, and the high processes to influence other high processes and declassifiers but *not* to influence low processes. However, in the transitive closure, all processes can influence all other processes, negating any desired security properties. Previous work assumes the existence of some global security policy, and modifies existing non-interference definitions to rule out flows not in the given policy [86].

In this thesis, we simplify the problem. Consider an export protection tag $t$, for which $t^+ \in \hat{O}$ and $t^- \notin \hat{O}$. We consider high symbols $HI_t$ as those that emanate from a process $p_i$ with $t \in S_i$. All other symbols are considered $LO_t$. Moreover, we consider only those processes that cannot declassify $t$. Let $N_t$ be the list of the process IDs of these processes:

$$N_t = \{j \mid t \notin O_j\}$$

Our proofs then cover non-interference results for $SYS_{N_t}$, all processes on the system that cannot declassify secret data tagged with tag $t$.

### 5.2.4 Model Refinement and Allocation of Global Identifiers

The model presented in Chapter 4 is almost fully-specified, with an important exception: the process *CHOOSE*:

$$CHOOSE_Y = ?(S, I) \rightarrow \sqcap_{y \in Y}(!y) \rightarrow STOP$$

The "nondeterministic internal choice" operator ($\sqcap$) implies that the model requires further *refinement.* The question becomes: how to allocate tags and process identifiers?

Consider an idea that does not work: *CHOOSE* picking from $Y$ sequentially, yielding the tag (or process ID) sequence $\langle 1, 2, 3, \ldots \rangle$. This allocation pattern allows high-throughput leaks of information from high to low. That is, the low process forks, retrieving a child ID $i$. Then the high process forks $k$ times, to communicate the value $k$ to low. The next time low forks, it gets process ID $i+k$, and subtracting $i$ recovers high's message. There are two problems: (1) low and high processes share the same process ID space; and (2) they can manipulate it in a predictable way.

In the naïve allocation scheme, the second weakness is exploitable even without the first. Consider the attack in which a high process communicates a "1" by allocating a new tag via create_tag(Add), and communicates a "0" by refraining from allocating. If a low process could guess which tag was allocated (call it $t$), it could then attempt to change its label to $S = \{t\}$. If the change succeeds, then the low process had access to $t^-$, meaning the high process allocated the tag. If the change fails, it follows the high process refrained from allocation. The key issue here is that the low process "guessed" the tag $t$ without the high process needing to communicate it. If such guesses were impossible (or very unlikely), the attack would fail.

Another idea—common to all DIFC kernels (c.f., Asbestos [21], HiStar [113] and the Flume implementation)—is random allocation from a large pool. The random allocation scheme addresses the second weakness—predictability—but not the first, and therefore fails to meet the formal definition for security. That is, operations like process forking and tag creation always have globally observable side affects: a previously unallocated resource becomes claimed.

Consider, as an example, this trace for the Flume system:

$$\begin{aligned} tr = \langle & i.b.(\{t\}, \{\}, \{\}, \{\}), \\ & i.s.(\{t\}, \{\}, \mathsf{fork}), \\ & j.b.(\{t\}, \{\}, \{\}, \{\}), \\ & i.s.(\{t\}, \{\}, j), \ldots \rangle \end{aligned}$$

A new process $i$ is born, with secrecy label $S_i = \{t\}$, and empty integrity and ownership. Thus, $i$'s actions fall into the $HI_t$ alphabet. Once $i$ starts, it forks a new process, which the kernel randomly picks as $j$. The child $j$ runs with secrecy $S_j = \{t\}$, inheriting its parent's secrecy label.

Projecting this trace onto the low alphabet yields the empty sequence ($tr \upharpoonright LO_t = \langle\rangle$). Thus, this trace should have no impact on the system from a low process $k$'s perspective. Unfortunately,

this is not the case. Before $tr$ occurred, $l$ could have forked off process $j$, meaning:

$$\begin{aligned} tr' = \langle & k.b.(\{\}, \{\}, \{\}, \{\}), \\ & k.s.(\{\}, \{\}, \mathsf{fork}), \\ & j.b.(\{\}, \{\}, \{\}, \{\}), \\ & k.s.(\{\}, \{\}, j), \ldots \rangle \end{aligned}$$

was also a valid trace for the system. But after $tr$ occurs, $tr'$ is no longer possible, since the process $j$ can only be born once. In other words, $tr \frown tr'$ is not a valid trace for the system but $tr'$ is by itself. This contradicts the definition of non-interference in the stable failures model of process equivalence.

Though random tag allocation does not meet the formal definition for non-interference, it still "feels" secure. Yes, high processes can theoretically interfere with low processes, but low processes will never observe that interference under conservative computation assumptions. That is, to observe that process $i$ forked process $j$, process $k$ would have to call fork an impractical number of times. One possible approach from here is to experiment with new, relaxed definitions on non-interference, though modeling cryptographic randomness has proven troublesome in the past [88].

To summarize, we have argued that due to the shared global capability pool $\hat{O}$ and the shared process ID pool $\hat{P}$, the allocation of these parameters must obey two properties: (1) partitioning; and (2) unpredictability. Our approach is to design a new allocation scheme that achieves both properties. We saw that certain schemes like random allocation achieve unpredictability. As for partitioning, we change the allocation scheme so that processes with different labels pick tags and processes IDs from different pools, meaning they can under no circumstance interfere with each other's choices. That is, the space of tags (and process IDs) is partitioned, and each $(S, I)$ pair picks tags (and process IDs) from its own partition.

To construct such an allocation scheme, we first define three parameters:

$$\begin{aligned} \alpha &\triangleq \text{the number of bits in a tag} \\ \beta &\triangleq \log_2(\text{maximum number of operations}) \\ \epsilon &\triangleq -\log_2(\text{acceptable failure probability}) \end{aligned}$$

As reasonable value for $\beta$ might be 80, meaning that no instance of the Flume system will attempt more that $2^{80}$ operations. Of course, allocating a tag or forking a new process is an operation, thus the assumption is that the system will allocate fewer than $2^{\beta}$ tags or process IDs. Similarly, it will express no more than $2^{\beta}$ different labels. A reasonable value for $\epsilon$ might be 100, meaning the system might fail catastrophically at any moment with probability no bigger than $2^{-100}$.

New we define a label serialization function, $s(\cdot)$. Given any label $L \subseteq \mathcal{T}$, $s(L)$ outputs a integer in $[0, 2^{\beta})$ that uniquely identifies $L$. The serialization can be predictable.

Next consider the family of all *injective* functions:

$$G : (\{0,1\}^{\beta}, \{0,1\}^{\beta}, \{0,1\}^{\beta}) \rightarrow \{0,1\}^{\alpha}$$

The Flume system, upon startup, picks an element $g \in G$ at random. When called upon to allocate a new tag or process ID, it returns $g(s(S), s(I), x)$, for some heretofore unused $x \in \{0,1\}^\beta$. The output is a tag in $\{0,1\}^\alpha$.

We can solve for how big $\alpha$ must be in terms of $\beta$ and $\epsilon$. Recall the first key property is *partitioning*, meaning functions in $G$ must be injective—their domains must fit inside their ranges: $2^{3\beta} \leq 2^\alpha$, or equivalently, $\alpha \geq 3\beta$. The second key property is *unpredictability*, meaning that the outcome of $g$ is not predictable. Since $g$ is chosen randomly from $G$, it will output elements in $\{0,1\}^\alpha$ in random order. After $2^\beta$ calls, $g$ outputs elements from a set sized $2^\alpha - 2^\beta$ at random. Since $\alpha \geq 3\beta$, this "restricted" range for $g$ still has well in excess of $2^{\alpha-1}$ elements. Failure occurs when a process can predict the output of $g$, which happens with probability no greater than $2^{\alpha-1}$. Thus, $\alpha - 1 \geq \epsilon$. Combining these two restrictions, $\alpha \geq \max(\epsilon + 1, 3\beta)$. Our settings $\beta = 80$ and $\epsilon = 100$ give $\alpha = 240$.

Thus, we assume that $\mathcal{T} = \mathcal{P} = \{0,1\}^\alpha$, for a sufficiently large $\alpha$. The kernel picks $g \in G$ at random upon startup. Then *CHOOSE* is refined as:

$$CHOOSE_Y = ?(S,I) \rightarrow \bigsqcap_{y \in \mathcal{G}(S,I,Y)} (!y) \rightarrow STOP$$

Where:

$$\mathcal{G}(S,I,Y) = \{ g(S,I,x) \mid x \in \mathcal{T} \wedge g(S,I,x) \in Y \}$$

Note that $\mathcal{G}(S,I,Y) \subseteq Y$, so the nature of the refinement is just to restrict the set of IDs that $CHOOSE_Y$ will ever output, based on the secrecy and integrity labels of the calling process.

## 5.3 Alphabets

We aim to show that $SYS_{N_t}$ fits the definition of non-interference given in Section 5.2. The first order of business is to define the alphabets $HI_t$ and $LO_t$, starting with $HI_t$:

$$\begin{aligned} HI_t \triangleq & \{ i.b.(S,I,\ldots) \mid i \in N_t \wedge S \subseteq \mathcal{T} \text{ s.t. } t \in S \} \cup \\ & \{ i.s.(S,I,\ldots) \mid i \in N_t \wedge S \subseteq \mathcal{T} \text{ s.t. } t \in S \} \end{aligned}$$

Now $LO_t$ is simply the complement of $HI_t$:

$$\begin{aligned} LO_t \triangleq & \{ i.b.(S,I,\ldots) \mid i \in N_t \wedge S \subseteq \mathcal{T} \text{ s.t. } t \notin S \} \cup \\ & \{ i.s.(S,I,\ldots) \mid i \in N_t \wedge S \subseteq \mathcal{T} \text{ s.t. } t \notin S \} \cup \\ & \{ tock \} \end{aligned}$$

These sets are trivially disjoint, and therefore they partition the possible alphabet for $SYS_{N_t}$, which we call $A$ for short:

$$A \triangleq \alpha SYS_{N_t} = HI_t \cup LO_t$$

The low set, $LO_t$ includes the event *tock* that marks the passing of time. In the discrete time model, these *tock* events can be arbitrarily interwoven in any trace.

The rest of the events in the Flume model (like communication through the switch, to the process or tag manager, etc.) are all hidden by the CSP-hiding operators, as given in Section 4.8. Thus, the exposed view of $SYS_{N_t}$ consists only of process births (i.e., $i.b$) and system call traces (i.e., $i.s$). For convenience, define the set of events that correspond to kernel process $i$'s incoming system calls, and a set of event that correspond to process $i$'s responses:

$$\begin{aligned} C_i \triangleq & \{i.s.(S, I, \mathsf{create\_tag}, w) \mid S, I \subset \mathcal{T} \wedge w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\}\} \cup \\ & \{i.s.(S, I, \mathsf{change\_label}, w, L) \mid S, I, L \subset \mathcal{T} \wedge w \in \{\mathsf{Integrity}, \mathsf{Secrecy}\}\} \cup \\ & \{i.s.(S, I, \mathsf{get\_label}, w) \mid w \in \{\mathsf{Integrity}, \mathsf{Secrecy}\}\} \cup \cdots \end{aligned}$$

And so on for all system calls. Similarly for return values from system calls:

$$\begin{aligned} R_i \triangleq & \{i.s.(S, I, t) \mid S, I \subset \mathcal{T} \wedge t \in \mathcal{T}\} \cup \\ & \{i.s.(S, I, r) \mid S, I \subset \mathcal{T} \wedge r \in \{\mathsf{Ok}, \mathsf{Error}\}\} \cup \\ & \{i.s.(S, I, L) \mid S, I, L \subset \mathcal{T}\} \cup \\ & \{i.s.(S, I, O) \mid S, I \subset \mathcal{T} \wedge O \subset \mathcal{O}\} \cup \\ & \{i.s.(S, I, p) \mid S, I \subset \mathcal{T} \wedge p \in \mathcal{P}\} \end{aligned}$$

The only visible events for process $i : K$ are system calls, system call replies and *tock*:

$$\alpha(i : K) = C_i \cup R_i \cup \{tock\}$$

A final notational convenience: we often describe the failures of a process $P$ projected onto the low alphabet $LO_t$ and abbreviate it:

$$\mathcal{L}_t[\![P]\!] \triangleq \mathcal{SF}[\![P]\!] \upharpoonright LO_t$$

## 5.4 Theorem and Proof

The main theorem is as follows:

*Theorem* 1 (Non-Interference in Flume). For any export-protection tag $t$, for any Flume instance $SYS_{N_t}$, and for any security parameter $\epsilon$, there exists an instantiation of *CHOOSE* such that $\Pr[\mathsf{NI}_{LO_t}(SYS_{N_t})] \geq 1 - \epsilon$.

We make several observations. First, note that $SYS_{N_t}$ is not a single CSP process but rather a family of processes, which vary from one another based on their user-space portions ($UPROCS_{N_t}$). The theorem must hold for all members of this family. Second, the theorem itself is probabilistic. As mentioned above, for any instance of $SYS_{N_t}$, there is a small chance that the output of *CHOOSE* is guessable, and in that case, the system may not exhibit the non-interference property. The best we can do is argue that the property fails with arbitrarily small probability.

**Proof** Consider any two traces $tr$ and $tr'$ such that $tr \approx_{LO_t} tr'$. The proof technique is induction over the length of the traces $tr$ and $tr'$. We invent a new function $\lambda(\cdot)$

$$\lambda(tr) \triangleq \#(tr \restriction LO_t)$$

that outputs the number of low events in a trace. Because $tr \approx tr'$, it follows that $\lambda(tr) = \lambda(tr')$. We first show the theorem holds for all traces $tr$ and $tr'$ such that $\lambda(tr) = \lambda(tr') = 0$. We then assume it holds for all traces with $\lambda(tr) = \lambda(tr') = k - 1$ and prove it holds for all traces with $\lambda(tr) = \lambda(tr') = k$.

**Base Case** For the base case, consider all $tr, tr' \in traces(SYS_{N_t})$ such that $\lambda(tr) = \lambda(tr') = 0$. In other words, $tr, tr' \in HI_t^*$.

At the system startup ($SYS_{N_t}$ after no transitions), all of the kernel process $i : K$ are waiting on a message of the form $i.b$ before they spring to life. Until such a message arrives, $i : K$ will refuse all events $C_i$ and $R_i$. The one exception is the process init, which is already waiting to accept incoming system calls when the system starts. By construction $S_{\mathsf{init}} = \{\}$ and $I_{\mathsf{init}} = \mathcal{T}$. Since $t \notin S_{\mathsf{init}}$, $C_{\mathsf{init}} \cup R_{\mathsf{init}} \subseteq LO_t$. Therefore, the system refuses all high events at startup, and $tr = \langle\rangle$ is the only trace of $SYS_{N_t}$ without low symbols (and for which $\lambda(tr) = 0$). For $tr = tr' = \langle\rangle$, the lemma trivially holds.

**Inductive Step** For the inductive step, assume the lemma holds for all traces $tr, tr'$ of $SYS_{N_t}$ such that $tr \approx_{LO_t} tr'$ and also $\lambda(tr) = \lambda(tr') = k - 1$. Now, we seek to show the lemma holds for all equivalent traces with one more low symbol.

Given an arbitrary trace $tr \in traces(SYS_{N_t})$ such that $\lambda(tr) = k$, write $tr$ in the form $tr = p \frown l \frown h$, where $p$ is prefix of $tr$, $l \in LO_t$ is a single low event, and $h \in HI_t^*$ are traces of high events. Similarly for $tr' \in traces(SYS_{N_t})$ where $tr \approx_{LO_t} tr'$: write $tr' = p' \frown l \frown h'$. It suffices to show that $\mathcal{L}_t[\![S/tr]\!] = \mathcal{L}_t[\![S/(p \frown l)]\!]$. If we have shown this equality for arbitrary $tr$, then the same applies for $S/tr'$, meaning $\mathcal{L}_t[\![S/tr']\!] = \mathcal{L}_t[\![S/(p' \frown l)]\!]$. By inductive hypothesis, $\mathcal{L}_t[\![S/p]\!] = \mathcal{L}_t[\![S/p']\!]$, and therefore $\mathcal{L}_t[\![S/(p \frown l)]\!] = \mathcal{L}_t[\![S/(p' \frown l)]\!]$. By transitivity, we have that $\mathcal{L}_t[\![S/tr]\!] = \mathcal{L}_t[\![S/tr']\!]$, which is what needs to be proven. Thus, the crux of the argument is to show that the high events of $tr$ do not affect low's view of the system; the second trace $tr'$ is immaterial.

We consider the event $l$ case-by-case over the different events in $SYS_{N_t}$:

- $l \in R_i$ for some $i$

  That is, $l$ is a return from a system call into user space. Because $l$ is a low event, $l$ is of the form $i.s.(S, I, \dots)$ where $t \notin S$. After this event, $i : K$ is in a state ready to receive a new system call ($i : K_{S,I,O}$). Because all events in $h$ are high events, none are system calls of the form $i.s.(S, I, \dots)$ with $t \notin S$, and therefore, none can force $i : K$ into a different state. In other words, the events $h$ can happen either before or after $l$; $SYS_{N_t}$ will accept

(and refuse) the same events after either ordering. That is:

$$\mathcal{L}_t[\![SYS_{N_t}/(p \frown l \frown h)]\!] = \mathcal{L}_t[\![SYS_{N_t}/(p \frown h \frown l)]\!].$$

We can apply the inductive hypothesis to deduce that:

$$\mathcal{L}_t[\![SYS_{N_t}/(p \frown h)]\!] = \mathcal{L}_t[\![SYS_{N_t}/p]\!]$$

Appending the same event $l$ to the tail of each trace gives:

$$\mathcal{L}_t[\![SYS_{N_t}/(p \frown h \frown l)]\!] = \mathcal{L}_t[\![SYS_{N_t}/(p \frown l)]\!]$$

and by transitivity:

$$\mathcal{L}_t[\![SYS_{N_t}/(p \frown l \frown h)]\!] = \mathcal{L}_t[\![SYS_{N_t}/(p \frown l)]\!]$$

which proves the claim for this case.

- $l = i.s.(S, I, \mathsf{create\_tag}, w)$ for some $i \in \mathcal{P}$, and some $w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\}$

  After accepting this event, the process $i : K$ can no longer accept system calls; it can only accept a response in the form $i.s.(S, I, t')$ for some tag $t'$, or *tock*. Since $l \in LO_t$, it follows that $t \notin S$ for both the system call and its eventual reply. The high events in $h$ could affect the return value to this system call (and therefore $\mathcal{SF}[\![S/tr]\!]$) if the space of $t$'s returned somehow depends on $h$, because $h$ changed the state of the shared tag manager. An inspection of the tag manager shows that its state only changes as a result of a call to $e = j.g.(S', I', \mathsf{create\_tag}, w)$ for some process $j$, and labels $S'$ and $I'$. Such a call would result in a tag such as $t' = g(S', I', x)$ being allocated, for some arbitrary $x$. Because $e \in h$ is a high event, $t \in S'$. Because $l$ is a low event, $t \notin S$. Thus, $S' \neq S$, and assuming $g$ is injective, it follows that $t' \neq t$, for all $x$. Therefore, events in $h$ cannot influence which tags $t'$ might be allocated as a result of a call to create_tag. We apply the same argument as above, that $h$ and $l$ can happen either before or after one another without changing the failures of the system. Hence, the claim holds in this case.

- $l = i.s.(S, I, \mathsf{change\_label}, w, L)$ for some $i \in \mathcal{P}$, $w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\}$ and $L \subseteq \mathcal{T}$.

  After accepting $l$, the process $i : K$ is expecting an event of the form $i.s.(S, I, r)$ for $r \in \{\mathsf{Ok}, \mathsf{Error}\}$, to indicate whether the label change succeeded or failed. It will transition to another internal state (and will behave differently in the future) on success. The only way an event in $h$ can influence this outcome is to alter the composition of $\hat{O}$, which the tag manager checks on $i$'s behalf by answering $i.g.(\mathsf{check+})$ and $i.g.(\mathsf{check\text{-}})$ within the *CHECK* subprocess.

  Consider the case in which $h$ contains an event $e$ such that $e = j.s.(S', I', \mathsf{create\_tag}, w)$, and $j$ is the high process that issued $e$ (that is, $t \in S'$). After $e$, the kernel might have

performed the internal events necessary to serving this system call, meaning a new tag $t'$ was allocated, and the tag manager switched to a new state reflecting $t'^{+} \in \hat{O}$ or $t'^{-} \in \hat{O}$. If $t' \in L$, then $h$'s occurrence allows $l$ to succeed, and $h$'s absence causes $l$ to fail. $t' \in L$ if and only if $\mathcal{L}_t[\![SYS_{N_t}/(p \frown l)]\!] \neq \mathcal{L}_t[\![SYS_{N_t}(p \frown l \frown h)]\!]$. However, we claim that $t'$ is a member of $L$ only if $U_i$ "predicted" the output of $g$, which it can do with negligible probability ($2^{-\epsilon}$). With extremely high probability, $L$ could only contain $t'$ if the event $e$ happened before the event $l$. But our inductive hypothesis has already ruled out this possibility.

- $l = i.s.(S, I, \mathsf{get\_label}, w)$ for some $w$.

  This call only outputs information about what state a kernel process is in; this state only updates as a result of low events $i.s.(S, I, \mathsf{change\_label})$. All events $e \in h$ do not fit this template since they are high events. Therefore, $h$ does not impact the result of system call $l$.

- $l = i.s.(S, I, \mathsf{get\_caps})$.

  There are three state transitions that can alter the reply to the get_caps system call: $i.s.(S, I, \mathsf{create\_tag}, w)$, $i.s.(S, I, \mathsf{drop\_caps}, L)$ or $i.s.(S, I, \mathsf{recv}, j)$. None of these calls are equal to an event in $h$, since they are low events and $h$ contains only high events.

- $l = i.s.(S, I, \mathsf{drop\_caps}, X)$ for some $X$.

  By definition of the *DROPCAPS* sub-process, a transition to a new $K_{S,I,O'}$ can follow a reply to $l$. If $e \in h$ can influence $O'$, then it can change $i : K$'s failures. However, $e$ cannot influence $O'$ since $O'$ is set to $O - X$ on a successful operation. If the event $e$ is to allocate a new tag $t'$, we can apply the same argument as above to see that $t'^{+} \notin O$ and $t'^{-} \notin O$, and therefore $e$ cannot affect $O'$.

- $l = i.s.(S, I, \mathsf{fork})$

  The only event $i : K$ will accept after $l$ (other than *tock* ) is $i.s.(S, I, k)$ where $k$ is the process ID of the newly-forked child. By definition of *CHOOSE* above, there exists some $x$ such that $k = g(S, I, x)$. If an event $e \in h$ causes a process ID to be chosen, it would be of the form $p = g(S', I', y)$, for some $y$, and some $S'$ such that $t \in S'$. That $l$ is a low symbol implies that $t \notin S$ and $S \neq S'$. If $g$ is injective then $k \neq p$. Therefore, event $e$ will never change the value $k$ that this kernel process might output next as its reply to the system call $l$.

  The other result of the fork system call is that now, a new process $k$ is running. That is, $k : K$ has moved out of the "birth state" and is now willing to accept incoming system calls in state $(k : K_{S,I,O})$. The same arguments as above apply here. Because $k$ was forked by a low process, it too is a low process, expecting only low symbols before it transitions to a new state. Therefore, the events in $h$ cannot affect its state machine.

- $l = i.s.(S, I, \mathsf{getpid})$

After accepting $l$, the process $i : K$ will only accept *tock* or $i.s.(S, I, i)$ in this state, so $h$ obviously has no effect.

- $l = i.s.(S, I, \mathsf{exit})$

  Regardless of $h$, a kernel process will only accept *tock* after exiting.

- $l = i.s.(S, I, \mathsf{send}, j, X, m)$ for some $j, X, m$.

  The outcome of the send operation depends only on whether $X \subseteq O$ or not. It therefore does not depend on $h$.

- $l = i.s.(S, I, \mathsf{recv}, j)$

  The event after $l$ that $i : K$ accepts is $i.s.(S, I, m)$ for some message $m$. It might also change to a different state if the process $j$ sent capabilities. The relevant possibility for $e \in h$ to consider is $e = j.s.(S', I', \mathsf{send}, i, \{t'^{+}\}, \langle\rangle)$, for some high process $j$ with $t \in S'$. The claim is that this message will never be enqueued at $i$ and therefore will not affect $i$'s next visible event. Say that process $j$ has ownership $O'$ and dual privileges $D'$. Because we assumed that $t^{-} \notin O \cup O' \cup \hat{O}$, $t$ cannot appear in either $D$ or $D'$. Also, because $i$ is a low process $t \notin S$. Therefore, $t \in S' - D'$ and $t \notin S \cup D$, which implies that $S' - D' \not\subseteq S \cup D$, and the kernel will not enqueue or deliver $j$'s message to $i$. Again, we have that $h$ does not affect the $i$'s possibilities for the next message it receives. The same argument applies to the final system call, select.

We have covered all of the relevant cases, and the theorem follows by induction. ∎

## 5.5 Practical Considerations

The construction of *CHOOSE*, based on a truly random function $g \in G$, is not practical for real implementations of Flume. The two requirements for $G$—partitioning (i.e., injectivity) and true unpredictability—must be relaxed in practice.

The first thought is to replace the random function family $G$ with a pseudo-random function family [38]. In this case, all aspects of the construction remain: the random selection of $g$ from its family, the serialization function $s(\cdot)$, and the input and outputs of the function $g$. In this construction, the hard bounds on true unpredictability are replaced with weakened bounds, reflecting computational assumptions for current hard problems.

Another implementation possibility is a "keyed-hash function" such as HMAC [6] in concert with a collision-resistant hash function like SHA-1 [28] or SHA-256 [29]. By definition:

$$\mathrm{HMAC}_k(x) = H(k \oplus \mathsf{opad}, H(k \oplus \mathsf{ipad}, x))$$

where $k$ is a secret key of sufficient length, opad and ipad are two fixed pads, and $H$ is a hash function like SHA-256. Thus, the kernel might pick a random (and secret) key $k$ on startup, and then compute new tags and process IDs with $\mathrm{HMAC}_k(S, I, x)$ for some counter variable $x$. This construction approximates both important properties. Assuming $H$ is collision-resistant,

$\text{HMAC}_k$ is also collision resistant, meaning an adversary cannot find $S, S', I, I', x, x'$ such that $\text{HMAC}_k(S, I, x) = \text{HMAC}_k(S', I', x')$ and the inputs differ at least in once place (i.e., $S \neq S'$ or $I \neq I'$ or $x \neq x'$). Thus, a high process with secrecy $\{t\}$ and a low process with secrecy $\{\}$ can only get the same tag (or process ID) if there is a collision in the hash function. Similarly, under standard computation assumptions, an adversary cannot predict a valid output of $\text{HMAC}_k(S, I, x)$ without knowing $k$.

The advantage of the keyed-hash function over the pseudo-random function is twofold: first the serialization function $s(\cdot)$ can be discarded; and second, tags can be smaller. Above, we suggested a reasonable length for tags might be 240 bits. If using HMAC with SHA-1, tag lengths are 160 bits. The cost of this reduction of $g$'s range is that $g$ is no longer injective; it is merely collision-resistant (i.e., injective under current computational assumptions).

The actual Flume implementation uses an even simpler approach. It picks $g$ at random from a family of pseudorandom functions, and outputs the sequence $g(1), g(2), g(3), \ldots$ for new tag values or process IDs. Of course, when $SYS_{N_t}$ is refined in this manner, many members of the $SYS_{N_t}$ family have the property $\Pr[\mathsf{NI}_{LO_t}(SYS_{N_t})] = 0$; for instance, a $SYS_{N_t}$ in which a high process allocates a tag, and then a low process allocates a tag (see Section 5.2.4). We leave for future work either the substitution of the HMAC function in the Flume implementation, or a formal argument that accommodates our actual approach. For now we conjecture that in practice this choice of $g$ does not negatively impact security.

## 5.6 Integrity

Though this chapter focuses on secrecy, the same arguments hold for integrity. Pick an integrity-protection tag $t$. Then the low symbols are those whose integrity tags contain $t$, and the high symbols are those that do not. The same proof shows that the high events do not interfere with the low.

# Chapter 6

# Fitting DIFC to Unix

In Chapters 3 and 4, we developed a model for a DIFC kernel, complete with the beginnings of a system call API. However, further challenges remain before we can build a Unix-like API from these primitives. To name a few:

1. A Unix kernel offers many ways for processes to communicate with one another: standard IPC (inter-process communication), the file system, virtual `mmap`ed memory, signals, System V IPC, TCP sockets, signals, and so on. The Flume model allows only one: sending an unreliable message from $p$ to $q$. A Unix-compatible Flume implementation therefore requires some mapping of all Unix communication mechanisms to a single DIFC primitive.

2. Every message sent between two processes entails a label check. Depending on implementation, this label check might be computationally expensive and slow down the system.

3. From the application designer's perspective, message sends can silently fail, greatly complicating debugging. Similarly, Unix-style flow-controlled pipes do not fit the Flume model as given in Chapter 4, thus programmers lose reliable IPC.

4. Definition 3 uses $\bar{D}_p$ to make message sends and receives maximally permissive, meaning a process that has capabilities always exercises them. Automatic exercise of privilege can lead to security bugs and is best avoided (c.f., the *confused deputy* problem [43]).

This section describes the *Flume system*, a refinement of the Flume *model* from Chapters 3 and 4. 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.

## 6.1 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* 5. A readable endpoint $e$ is *safe* iff

$$(S_e - S_p) \cup (I_p - I_e) \subseteq \bar{D}_p.$$

A writable endpoint $e$ is safe iff

$$(S_p - S_e) \cup (I_e - I_p) \subseteq \bar{D}_p.$$

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* 6. 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 6 that $e$ is writable, $f$ is readable, and $S_e \subseteq S_f$. Since $e$ and $f$ are safe, Definition 5 implies that $S_p - \bar{D}_p \subseteq S_e$ and $S_f \subseteq S_q \cup \bar{D}_q$. Combining the three observations yields:

$$S_p - \bar{D}_p \subseteq S_e \subseteq S_f \subseteq S_q \cup \bar{D}_q$$

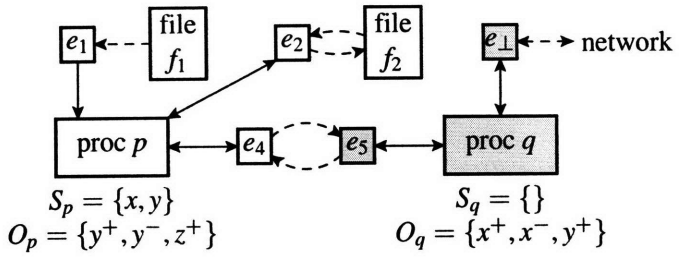


Figure 6-1: Processes $p$ and $q$. Assume $\hat{O} = \{\}$.

Thus, $S_p - \bar{D}_p \subseteq S_q \cup \bar{D}_q$, and the message between processes is safe for secrecy by Definition 3. A similar argument holds for integrity. ∎

## 6.2 Enforcing Safe Communication

For the Flume system to be secure in the model defined in Chapter 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.

**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 socket pair. 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 6. 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 5), 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 6-1, $p$ 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 7.3.

**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 7.4.1). 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).

Because Flume does not intercept individual file I/O operations, a process $p$ must hold such an endpoint $e_f$ at least until it closes the corresponding file. Moreover, all labels on file endpoints (such as $e_f$) are *immutable*: $p$ cannot change them under any circumstances. Because of label immutability, and because 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 6-1, 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 $S_p - \bar{D}_p = I_p - \bar{D}_p = \{\}$. That is, $p$ has the privileges required import and export all of its data.

For instance, process $q$ in Figure 6-1 has a network socket, and therefore gets an immutable endpoint $e_\perp$. This endpoint prevents $q$ from reading export-protected data it cannot export, since the assumption is that $q$ would leak the data. Thus, $q$ cannot raise $S_q = \{y\}$, as such as change would compromise $e_\perp$'s safety.

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.

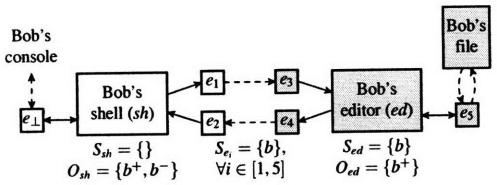


Figure 6-2: A configuration for Bob's shell and editor. Here, $\hat{O} = \{b^+\}$.

## 6.3 Examples

Endpoints help fill in the details of our earlier examples (from Section 3.3.4). For our secrecy example, Figure 6-2 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 $b^+ \in \hat{O}$, $b^- \in O_{sh}$ and therefore $b \in \bar{D}_{sh}$. *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.

Note that since $e_5$ is immutable, it prevents the editor from changing $S_{ed}$ to $\{a, b\}$, even though $a^+ \in \hat{O}$. This restriction makes sense; without it, Bob's editor could copy Alice's secret data into Bob's file.

In our secrecy example, Bob's shell process *sh* can communicate externally through standard input, output and error; therefore Flume attaches an immutable endpoint $e_\perp$ to *sh*. For Bob to read one of his own files, he must either change $S_{sh}$ to $\{b\}$, or establish a readable, immutable endpoint with secrecy $\{b\}$. Either configuration is possible (i.e., does not conflict with $e_\perp$), since $b^- \in O_{sh}$ and $b^+ \in \hat{O}$. If *sh* were to read one of Alice's files, it must likewise change $S_{sh}$ to $\{a\}$ or allocate a new readable endpoint with secrecy label $\{a\}$. But neither configuration is possible: a label of $S_{sh} = \{a\}$ would conflict with $e_\perp$ and an endpoint with secrecy $\{a\}$ is not safe given $O_{sh}$. Thus, Bob cannot view Alice's private data from his shell (i.e., export it).

In the shared-secrecy calendar example, Bob launches the process $q$ that examines Alice's calendar file. $q$ is disconnected from Bob's shell, and therefore does not have any endpoints when it starts up. $q$ can then freely set $S_q = \{a\}$, since $a^+ \in \hat{O}$ and $q$ has no endpoints. What if $q$ opened a writable file $f$ before changing $S_q$ to $\{a\}$? If $f$'s endpoint has secrecy $S_{e_f} = \{\}$, then $q$ would fail to raise $S_q$, since the label change would invalidate $S_{e_f}$. So $q$ cannot leak Alice's data to a file $f$ if $f$ is not export-protected. If $f$'s endpoint has secrecy $S_{e_f} = \{a\}$, then $q$ could raise $S_q$ to $\{a\}$ as before.

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 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.

# Chapter 7

# 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 [34] and Plash [94]. 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) [109] implements Flume's system call interposition (see Section 7.2). The OpenBSD implementation of Flume uses the *systrace* system call [80] instead, but we focus on the Linux implementation in this description.

Figure 7-1 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 7.2), a remote tag registry (see Section 7.4.3), and user space file servers (see Section 7.4.7). The Flume-aware C library redirects Unix system calls to the RM and also supports the new Flume calls shown in Figure 7-2. 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.

## 7.1 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.

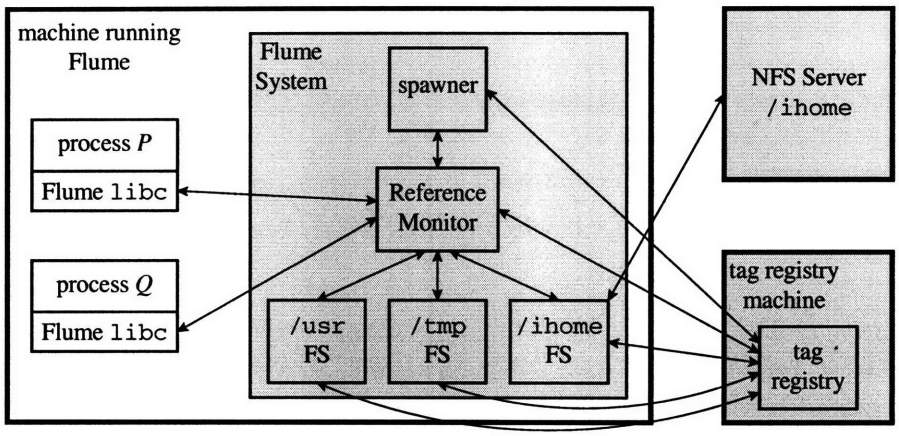


Figure 7-1: High-level design of the Flume implementation. The shaded boxes represent Flume's trusted computing base.

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., $O_p - \hat{O} = \{\}$). 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 6.2). Since a process's endpoints must all be safe, a process with $e_\perp$ can add a tag $t$ to its secrecy or integrity label only if it owns both $t^+$ and $t^-$. Thus, processes with endpoint $e_\perp$ cannot view secret data unless they are authorized to export it.

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. As they do so, standard Unix permissions prevent unprivileged, unconfined processes from reading the file system that Flume maintains. That is, files under Flume's control are owned by the user `flume` with access permissions like `0600` for plain files and `0700` for directories and binaries. Non-root users running as a user other than `flume` cannot access these files due to standard Unix access control checks.

## 7.2 Confinement, spawn and flume_fork

*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 7-4 provides a partial list of which calls fall into which categories. The goal here is for the RM to

- `label` **get_label**`({S,I})`
  Return the current process's $S$ or $I$ label.
- `capset` **get_caps**`()`
  For the current process $p$, return capability set $O_p$.
- `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 5). Return an error code on failure.
- `int` **drop_caps**`(capset` $O'$`)`
  Reduce the calling process's ownership to $O'$. 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 5), or if the endpoint is immutable. Still succeed even if the change stops endpoint flows (in the sense of Definition 6).
- `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 $\hat{O}$; in the second add $t^-$ to $\hat{O}$; 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 $O$.
- `pid` **flume_fork**`(int nfds, const int close_fds[])`
  Fork a copy of the current, confined process, and yield a confined child process. In the child, close the given file descriptors after forking the Unix process structure, but before subjecting the child process to scrutiny.

Figure 7-2: A partial list of new API calls in Flume.

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. Confined process come into existence by one of two means: via spawn or flume_fork.

### 7.2.1 spawn

Confined and unconfined processes alike can call spawn to make a new confined process. 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 7.4.1). System call interposition blocks other resource-granting system calls.

Without explicit access to the `fork` stage of the spawn operation, 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, opaque 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 $p$. Now $p$ and $q$ can communicate.

Processes calling spawn (and also flume_fork below) therefore depend on"pipe tokens" to communicate with their children, but the primitive is more general: a process $p$ can call flume_pipe or flume_socketpair at any time, and communicate the token to other processes via IPC, the file system, or any other means. Hence, pipe tokens must be randomly-chosen and unguessable: whichever process presents the token first will "win" the other side of $p$'s pipe or socketpair. If, by contrast, $q$ could guess $p$'s pipe-token, it could impersonate $p$'s intended counterparty, stealing or vandalizing important data.

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`.

Internally, 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 7.4.5); (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 run as an unprivileged user with whom other unprivileged users cannot interfere (along the same lines as Apache's www user). 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.

Finally, Linux notifies the spawner when a spawned process exits. The spawner reports this fact to the creating process via the reference monitor if labels allow communication between the exiting and creating process.

### 7.2.2 flume_fork

Confined processes can also fork other confined processes via flume_fork, an approximation of standard Unix `fork`. Fork is preferable to spawn in some cases for one major reason: performance. On the Linux machine used for our benchmarks (see Chapter 9), forking one Python process from another is five times faster than spawning that same new Python process anew. Such a discrepancy is exacerbated by Flume's system call overhead, which makes the many `open` calls involved with spawning a new Python process (and importing runtime libraries) all the more expensive. Thus, a busy system (like a Web server) might chose to spawn one Python process, importing all of the necessary libraries, then fork once for each incoming client request, rather than spawning each time.

The difficulty in forking in a DIFC context is *shared* file descriptors. As an example, consider the attack that Figure 7-3 depicts. The parent process $p$ intends to steal the data stored in the file $f$, even though $S_p = \{\}$ and $S_f = \{\}$. To pull off the heist, $p$ first launches a *source* process, whose task is simply to output a sequence of integers, like $\langle 1, 2, 3, \ldots \rangle$, to its standard output. The parent $p$ listens to the source on its standard input. It then forks a child $c$, so that after the fork, $p$ and $c$ share the pipe to the source. Next, $c$ raises its label to $S_c = \{t\}$, reads the contents of the file $f$ (in this case "5"), then reads that many integers from standard input. After waiting a sufficient time, $p$ reads from its standard input, retrieving the value $c$ wished to communicate to it, and therefore the value stored in $f$.

The shared resource $p$ and $c$ use to communicate in this attack is the stream of data coming

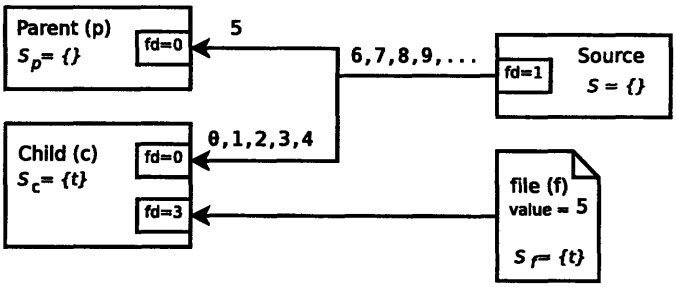


Figure 7-3: An attack against `fork` that allows file descriptor sharing between parent and child.

from the source process, which they can both read and therefore *modify* with their shared file descriptor (as noted by IX's authors [66]). Thus, a fork for confined processes cannot allow parents to share file descriptors with their children. And more obviously, `mmaped` memory regions established in the parent process and inherited by the child are verboten.

Given these constraints, forking among confined process proceeds as follows:

1. The parent calls flume_fork, providing as an argument the set of all file descriptors to close in the child.

2. In the library, the parent asks the reference monitor for a new control socket (eventually to give to the child). The parent also creates a temporary pipe.

3. The parent calls standard Unix `fork`. The parent closes the child's control socket, and the child's end of the temporary pipe. It then writes a byte to its end of the temporary pipe.

4. The child closes the parent's control socket, and the parent's end of the temporary pipe. It also closes all of the file descriptors passed in as arguments to flume_fork. It waits on its side of the pipe for an incoming byte. Once received, the child has exactly one file descriptor—its own control socket. Furthermore, it knows that it controls the only reference to that file descriptor, since it received the byte the parent sent *after* it closed its copy of that same file descriptor.

5. The child then calls into the reference monitor, asking for "unshared fate," i.e., the ability to set its labels independently of its parent's. Before the reference monitor grants this request, it calls into the kernel to scrutinize the child, ensuring:

   (a) The child has no *shared* `mmaped` memory regions

   (b) The child has only open file descriptor, its control socket.

   (c) The child's end of the control socket only has one reference, and that the other side is controlled by the reference monitor.

6. On success, the child gets independence and confinement; it can them reopen pipes using pipe tokens inherited in its address space.

7. The parent gets, as a return value, the process ID of the child process.

The goal of this protocol is to ensure that if parent and child can set labels independently (as in Figure 7-3's attack), then they share no communication channels at the time of the fork. The checks in Step 5 ensure no communication via file descriptors, and no communication via shared `mmaped` memory pages. One might be tempted to ensure the child has *no* `mmaped` pages; however, this restriction is impractical, since many components of the standard library (e.g., dynamic library loading, configuration file reading, dynamic memory allocation) use `mmap`. In all cases[1], those mappings are `MAP_PRIVATE` and therefore cannot be used to write to a child across a `fork`.

In Step 5, the kernel checks that the child has only a control socket open. If the child could inherit arbitrary file descriptors from its parent, the kernel checks would be considerably more complicated, forcing the Flume LSM to look deep into Linux data structures, which are bound to change over time. The "one-file policy" significantly simplifies the kernel checks required, yielding another application for flume_pipe and pipe tokens.

As Figure 7-4 shows, confined processes are still free to call the standard Unix `fork`. However, if they do, the reference monitor treats the parent and child as the same process. If the child changes label, the label change also affects the parent. Indeed, the parent and child can communicate with each other via `mmaped` memory or shared file descriptor ends, but because their labels cannot diverge, they cannot use the channels to move information against safety rules.

## 7.3 IPC

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. (Recall that the RM cannot reject such a change, since so doing would convey to $p$ information about $q$'s labels, or vice versa).

Flume takes special care to prevent unsafe information flows when the processes at either end of a pipe or socket have different labels. 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

[1]With the exception of `gconv` in `glibc 2.6.1`.

| 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), flume_fork† getcwd, getpid, kill, link, lstat, mkdir, open, symlink, readlink, rmdir, spawn†, stat, unlink, utimes, wait ... |
| | **Forbidden** |
| | bind(network socket), execve, getsid*, getpgrp*, getpgid*, getppid*, ptrace, setuid ... |

Figure 7-4: A partial list of 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.

$p$. 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. In practice, one-way communication is cumbersome and rarely used; see Section 10.1 for more details.

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.

## 7.4 Persistence

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.

### 7.4.1 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 reference monitor cannot then interrupt the process's reads and writes to the file if the process changes its label in a way that make that flow unsafe, as it does with pipes. Instead, the reference monitor relies on immutable endpoints to restrict the way the process can henceforward change its labels.

File opens work as described in Section 6.2, 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 an immutable *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: $t^- \in \hat{O}$, 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. The "stickiness" of these endpoints is indeed a shortcoming of the system, as it complicates application development. Future versions of Flume might do better file-descriptor accounting, allowing a process to drop an immutable endpoint after it closes the last reference to a file.

### 7.4.2 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 it, Flume offers a namespace logically filled with precreated directories for each $(S_p, I_p)$ pair. $p$ can write to directory of the form `/ihome/`$\mathsf{srl}(I_p)$`.`$\mathsf{srl}(S_p)$, where

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.

### 7.4.3 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.

Flume has a simple mechanism for sharing 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 7.4.7).

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. Such a facility is useful, for instance, in the management of Web sessions. The privileges $u$ uses during a Web session can be converted to such a token and then stored on $u$'s Web browser as an HTTP cookie, allowing $u$ to recover the necessary capabilities before each page load.

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.

### 7.4.4 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 7.4.3.

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 safety. 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.

### 7.4.5 Setlabel

Flume provides a *setlabel* facility, analogous to Unix's *setuid* or HiStar's gates, that is the best way for a process without privileges 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. A process whose $S$ and $I$ 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, granting the spawned process the capability referred to by the login token.

An example use for a setlabel process is a password checker. A process $p$ has a hash of a password for a user $u$, and wants to check if that hash matches $u$'s password in a secret password file. The password file is labeled $S_f = \{t\}$, where $t$ is an export protect tag. The password checker $q$ runs as a setlabel process. The setlabel file contains the path of the password checker binary, and also a login token for $t^-$. $p$ launches $q$, feeding it the user $u$ and the supposed hash of $u$'s password. $q$ reads in the password file, checks the hash, and outputs success or failure to $p$. This output declassifies one bit about the password file, and therefore requires the exercise of $t^-$.

Setlabel files can also specify a minimum integrity label, $I_s$. The RM only allows a process $p$ to execute such a setlabel file if $I_s \subseteq I_p$. This minimum integrity requirement helps defend the

setlabel process from surprises in its environment (such as a bad LD_LIBRARY_PATH).

### 7.4.6 Privileged Filters

Finally, in the application we've built, we have found a need for automatic endorsement and declassification of files; see Section 8.7 for a detailed motivation. A process can create a *filter* to replace "find label" ($L_{\text{find}}$) with a "replace label" ($L_{\text{repl}}$) 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 $L_{\text{find}}$, Flume replaces those tags with $L_{\text{repl}}$ before it decides whether to allow the process to open the file. A process can activate multiple filters, composing their effects.

### 7.4.7 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).

## 7.5 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 [57]. 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.

# Chapter 8

# Application

This section explores Flume's ability to enhance the security of off-the-shelf software. We first describe MoinMoin [68], 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 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.

## 8.1 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 [73] and MoinMoin's internal bug tracker [68] 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.

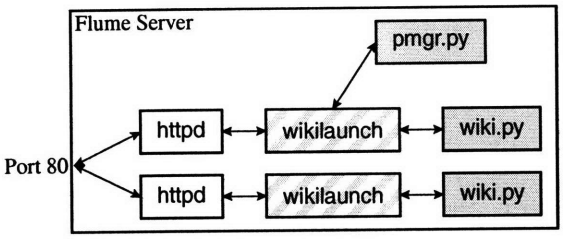


Figure 8-1: FlumeWiki application overview, showing two of many process pipelines. The top request is during a session login; the bottom request is for a subsequent logged-in request. Flume-oblivious processes are unshaded, unconfined processes are striped, and confined processes are shaded.

A site administrator may want to install a plug-in for some parts of the site, but not all of it. For example, the engineering company's Moin administrator may only trust Moin's base code to edit and render engineering documents, but she may want to allow plug-ins to run on other portions of the site. Currently, this policy is difficult to enforce because Python can dynamically load plug-ins at any time; a bug in Moin could cause it to load untrusted plug-ins accidentally.

## 8.2 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."

## 8.3 FlumeWiki Overview

Figure 8-1 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. 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

login 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.

## 8.4 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 $W = \{{w_x}^+\}$.

## 8.5 Acquiring and Granting Capabilities

When a user $u$ logs into FlumeWiki at the beginning of a session, she provides her username and password. wikilaunch the contacts the principal manager (pmgr.py) which verifies $u$'s password and creates a temporary session token (as described in Section 7.4.3) for $u$'s capability group $G_u$. wikilaunch saves this session token as a cookie on $u$'s Web browser and on subsequent requests, wikilaunch uses the cookie to claim $G_u$ from the RM. It then determines what page $u$ is requesting and what $S$ and $I$ labels to use when spawning wiki.py. Note that wikilaunch only receives capabilities that $u$ is supposed to have; it cannot accidentally grant wiki.py anything outside of $G_u$. Internally, the principal manager stores a hash of each user $u$'s password, read-protected by $r_u$. pmgr.py runs as a setlabel program with a capability group containing every users' $r_u$ tag.

## 8.6 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 $O_q = \hat{O}$, allowing the standard MoinMoin code in FlumeWiki transparent read access to files the user is allowed to read (see Figure 8-2). For a request that involves creating or modifying a page, wikilaunch looks at the

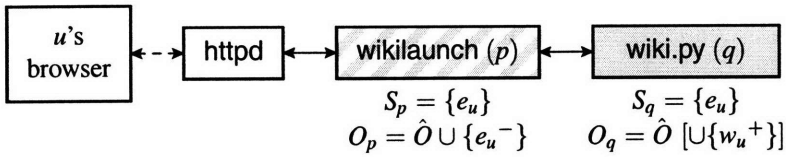


Figure 8-2: Label setup for a *read* or *write* request in FlumeWiki. wiki.py only gets capability $w_u^+$ if writing. The target page is export- and write-protected by user $u$.

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 8-2). 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 bi-directional 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.

## 8.7 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 7.4.6) 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.

## 8.8 Principal Management

FlumeWiki stores $u$'s private data including her email address and site preferences in the `/ihome/` file system with the labels: $S = \{r_u\}$ and write-protection $W = \{{w_u}^+\}$ which read and write protects it from other users. The principal management mechanisms are not specific to FlumeWiki and could be used in other similar systems.

A FlumeWiki installation has a special administrator account ($A$) whose $G_A$ contains a capability for each principal's $r'_p$. The administrator's powers are exercised by a "principal manager," a setlabel executable called `pmgr.py`, that runs with $I = \{i_A\}$ and with $O = G_A$. The integrity restriction prevents `pmgr.py` from accidentally referencing low-integrity shared libraries or Python libraries. The FlumeWiki user interface runs `pmgr.py` to perform tasks that require administrator privileges, which include `CreateUser`, `LoginUser`, `CreateUserGroup` and `InviteUserToGroup`.

`CreateUser` creates the tags mentioned in Section 8.4 and puts them in a newly created $G_u$. It adds the new user's $r'_u$ to $G_A$ so that the administrator will be able to read the user's `passwd.rp` but not the user's documents. Finally the principal manager creates the new user's home directory and `passwd.rp`.

When creating a new group $g$, the principal manager creates the standard set of tags and capabilities, and then grants access to $G_g$ (the capability group containing all of $g$'s capabilities) to whomever the group administrator is. The principal manager also creates another capability group $G'_g = \{{e_u}^-, {r_u}^-, {r_u}^+\}$ for *read only access* to $g$'s data. Through the principal manager, $g$'s administrator can extend invitations to other principals on the system to join group $g$. If $g$'s administrator wishes to grant $u$ read access to $g$, then the principal manager on his behalf creates a new login token for $G'_g$ and writes it to a read-protected file in $u$'s home directory. When $u$ logs in next, he can accept read-only membership into $g$ by adding $G'_g$ to his capability group $G_u$. The same process is followed for read/write access, using the capability group $G_g$ instead of $G'_g$. Note that since capabilities are transferable, any member of $g$ with read access to $g$ can grant this capability to other users on the system (and similarly for read/write access).

`LoginUser` is the implementation of the steps described in Section 8.5.

## 8.9 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.

# Chapter 9

# 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 35–286$\mu$s 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.

## 9.1 Security

The most important evaluation criterion for Flume is whether it improves the security of existing systems. Of the five recent ACL bypass vulnerabilities [73, 71], 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.

## 9.2 Interposition Overhead

To evaluate the performance overhead when Flume interposes on system calls, we measured the system call latencies shown in Figure 9-1. In all of these experiments, the server running Linux

| 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 9-1: System call and IPC microbenchmarks, and Flume overhead as a multiplier. Latencies are in $\mu$s 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.

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 35–286$\mu$s 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 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 40$\mu$s 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 `reads` and two `writes`. 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 9-1.

## 9.3 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 9-2 summarizes the results.

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 `opens` and 186 `stats`. 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 `stats`, 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 9-2 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.

## 9.4 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

| | **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 9-2: Latency and throughput for FlumeWiki and unmodified MoinMoin averaged over 10,000 requests.

centralized tag registry and FS file sharing supports another strategy for improving performance, namely clustering. We tested scalability on a "virtual" cluster, running the FlumeWiki read throughput experiment on the same server hardware, but with a varying number of single CPU virtual machines on top of a single 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 (18.8 req/sec) because of VM and NFS overhead.

## 9.5 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 [99] show a tenfold performance improvement when running MoinMoin as a FastCGI application [26] 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. Similarly, folding wikilaunch into the web server would avoid a `fork` and `exec` per incoming request.

# Chapter 10

# Discussion, Future Work and Conclusions

The thesis has made the case—via Flume—that DIFC's advantages 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 that are exploitable on a non-DIFC system. In this final chapter, we examine the Flume approach, asking questions such as: Is it really secure? Is it general? Is its complexity necessary? Is it solving a problem that actually matters? Reflecting upon general experience with Flume, and contrasting with experience on the Asbestos project (of which I am a proud member), we speculate on research directions for the future.

## 10.1 Programmability

MAC systems have a reputation for being difficult to program. A typical sticking point is *label creep*: all security labels monotonically increase, until no process on the system can perform a useful task [89]. The newer wave of statically-checked systems with decentralized information flow control (e.g., Jif) solve some of these problems but introduce new ones: they require security type annotations on variables and methods, and therefore increase the amount of thinking and typing that programmers must do; moreover, they demand many programs and libraries be deeply modified [44]. Asbestos and HiStar also eliminate label creep with decentralization of privilege, and without type-annotation overhead. But both require the system developers to maintain a new software stack (especially in Asbestos) and application programmers to learn a new programming model.

Does Flume improve programmability? Like Jif, Asbestos and HiStar, Flume offers decentralization of privilege, to combat label creep. But relative to those systems, Flume has better support for legacy software, legacy platforms, and tools that programmers typically use. In this sense, using a Flume-enabled machine is less of a "research-prototype" experience, and similar to using the underlying operating system (i.e., Linux): unconfined processes like standard

```
1 import flume.flumos as flmos
2 import flume

4 t = flmos.create_tag (flume.TAG_OPT_DEFAULT_ADD, "Alice")
5 flmos.change_label (flume.LABEL_O, flmos.Label ())
6 f = open ("/tmp/alice.dat", "w")
7 flmos.change_label (flume.LABEL_S, flmos.Label ([t]) )
```

Figure 10-1: An example of Python code that fails under Flume, due to violation of endpoint label safety.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "python2.5/site-packages/flume/flmos.py", line 960, in set_label
    raise_err ("set_label failed")
  File "python2.5/site-packages/flume/flmos.py", line 38, in raise_err
    raise flume.PermissionError, s
flume.PermissionError: "set_label failed; check_add_sub_all failure;
        could not subtract all of [0x53e5f900ec160c9] with an ownership
        label of []
     Check endpoint label (S) failed for EP '/tmp/alice.dat [rw]':
        don't have all capabilities for difference label
        [0x53e5f900ec160c9] where this EP label is []
        and the process's label is [0x53e5f900ec160c9]
      In set_process_label(label_type_t  = LABEL_S;, [0x53e5f900ec160c9])"
```

Figure 10-2: The error message that the code in Figure 10-1 produces.

daemons start up as normal; patches, upgrades, shells, X-windows, and even proprietary binary software work as before.

Of course the real question is: what is the programmer experience when developing, testing, debugging, and running confined code, or unconfined code that calls upon the Flume API. A programmability improvement relative to a system like Asbestos are the details conveyed in error messages resulting from system call failures. Many important Asbestos system calls (those derived from `sys_send`) cannot report errors at all for fear of information leaks. Label changes silently fail, as do message sends to other processes, etc. Flume cannot solve all of these problems, but in an important set of cases, it can.

Consider the actual Python program shown in Figure 10-1. Here, the program creates a new export-protection tag $t$ in line 4, discards the capability $t^-$ in line 5, opens a writable file (whose labels default to $S = I = \{\}$) in line 6, then attempts to raise its process label to $S = \{t\}$ in line 7. The label change in line 7 fails, because the writable, immutable endpoint on the file opened in line 6 has label $S = \{\}$, which is no longer safe. Flume can give a detailed error message at the site of the failed system call, as shown in Figure 10-2. The error explains the exact label, capability and endpoint configuration that caused the failure. The data conveyed

here is rich enough for a good programmer to get a sense of what the bug is. For those of lesser abilities, this messages will admittedly be vexing (and difficult to Google). The same error crops up in other cases, such as processes failing to change labels if the change would cut off IPC or signal retrieval. With endpoints, the report of the bug happens as close to the bug as possible. This experience contrasts markedly with application development on Asbestos, in which many errors were unreported and only manifested themselves much later in the program.

Another improvement in Flume (and HiStar) over Asbestos is flow-controlled IPC, as in standard Unix pipes and socketpairs. In Asbestos, communication between processes happens via unreliable messages. $p$ sends to $q$ without knowing how fast to send, or whether $q$ received the message, much like in UDP. Future Asbestos development plans call for reliable communication in a user-level library (like a TCP over UDP implementation), but such a facility did not exist when we were building Asbestos applications. In Flume, the assumption is that bidirectional flow-controlled communication is norm, and for this reason, endpoints allow two processes with different labels and sufficient privileges to act as if their labels are equal. Relative to our development experience with unreliable communication in Asbestos, our experience with reliable communication in Flume is extremely positive. Occasions for reliable communication abound in the MoinMoin Wiki example: between the the Web server and the launcher, the launcher and the application, and application and the file system, the launcher and the user manager, etc. To establish the same communications with unreliable messages would be tedious and error-prone. Though Flume processes with uneven labels can communicate unreliably, we have never found a need for this feature.

Flume has room for improvement. For instance, it ought to implement better tracking of open files, so that a process can close a final reference to a file and drop the corresponding endpoint. Certain certain programming tasks under Flume have proven difficult, such as implementing Web services with the flume_fork rather than the spawn primitive (c.f., Sections 7.2.1 and 7.2.2). Many bugs surrounded the closing and reopening all file descriptors (including standard error!) on either side of the call to Linux's `fork`. These implementation challenges prevented the measurement of flume_fork for this thesis, but preliminary results shows that performance improvements relative to spawn are substantial (on the order of 10x).

Debugging in general remains rudimentary. Most debugging now happens via print statement sent to standard error. Sometimes bugs require relaxation of information flow control rules, so that confined processes can still output error messages to the administrator (us), again via standard error. In the future, these debugging statements should flow through dedicated debugging devices, one per $(S, I)$-label combination. The problem becomes more complicated in the scenarios described below in Section 10.5, in which developers should not have access to debugging output, since their code is running over data that they are not authorized to see.

## 10.2 Security Compartment Granularity

A primary difference between language-level DIFC (e.g., Jif) and OS-level DIFC (e.g., Flume) is the size of security compartments. In the former, they can be as small as a bit; in the latter, they can be no smaller than a thread (in HiStar) or process (in Flume). The MoinMoin applica-

tion, and for that matter most CGI and FastCGI-based applications, can work with process-level label granularity. But some modern applications demand finer granularity, like Web application servers (e.g., JBoss [82]) and relational databases (e.g., PostgreSQL [76]). Such applications can find convenient expression in Jif, but on Flume would either need privilege for many tags, or operate at a very high (and therefore useless) secrecy level.

Perhaps a good general solution is a hybrid approach. Consider, for example, an implementation of FlumeWiki with a database backend. The database can be written with Jif, meaning information flow rules control movement of labeled data inside the process's address space, with declassification and endorsement occurring at a few isolated, privileged functions. From Flume's perspective, the database runs as a highly privileged component, with access to $t_u^-$ for all $u$. But the administrator relies on Jif's guarantees to reassure herself that the database uses those privileges responsibly. If the declassifiers and Jif's implementation are correct, she is not vulnerable to bugs in the large remainder of the database code. The rest of the system—wikilaunch and wiki.py—operate as previously described, with one label per address space.

## 10.3 Maintenance and Internals

We built Flume with an eye toward ease of future maintenance, keeping most code in application space. Some important maintenance tasks remain. The most onerous in our experience is updating Flume's patch to the Linux kernel (c.f., Section 7.2.1), since even minor kernel revisions can break the patch. Flume also demands a patch of `glibc` and `ld.so`, but this software has stayed much more stable over our two years maintaining Flume.

Some internal aspects of Flume's implementation proved trickier than we would have liked, such as of `wait` and `exit`. The state machine for this aspect of Unix is complex to begin with, but message delivery constraints induced by DIFC add further complexity. For instance, a parent $p$ at secrecy $S_p = \{\}$ gets an exit signal from a child $c$ at $S_c = \{t\}$ not when $c$ actually exits, but rather when $c$ changes its label. Many similar corner cases complicate the implementation and internal garbage collection of per-process resources. Also, in retrospect, factoring the Flume file servers into independent processes added extra implementation, debugging and performance overhead. Though it feels like the correct design decision in terms of privilege separation [91], future revisions of Flume's might fold these operations directly into the reference monitor.

## 10.4 Threat Model

A big question to consider when evaluating Flume or systems like it, is what is the *threat model*: who is attacking the system, how do they attack it, and what are their goals? The easiest threat to protect against are those introduced by well-intentioned but careless programmers, like the ACL-bypass vulnerabilities we noted in MoinMoin Wiki. Programmer bugs become more virulent when they allow arbitrary code execution (like a buffer overrun in C, a call to `system` with unescaped data in Perl, or a SQL injection attack); an adversary who exploits such a bug can control the Web server and the data it has access to. Even more difficult to defend against is an

adversary who is *invited* to run his code on the server or with access to sensitive site data (as in Facebook's application platform). In general, if a system intends to defend against a more severe attack, it must make a greater upheaval to longstanding operating system or programming techniques. We examine some of these trade-offs in the context of Flume and other ideas for Web security.

### 10.4.1 Programmer Bugs That Do Not Allow Arbitrary Code Execution

To protect only against bugs like the ACL-bypass vulnerabilities in MoinMoin, a system simpler than Flume might suffice. To secure MoinMoin, one can imagine building Flume's file I/O checks into a Python library. Whenever MoinMoin writes a file to disk, the library updates an application-level label on the file. Whenever MoinMoin reads a file from disk, the library reads in the label, and updates the label on the process. When MoinMoin outputs to the network, it tells the wikilaunch process what it thinks its label is, and wikilaunch applies the same sort of policies as in FlumeWiki. A more powerful technique is to build similar behavior into run-time interpreter. For instance, Perl introduced a *taint* facility years ago, which categorizes data into one of two integrity categories, and updates integrity labels as data flows through the run-time interpreter [8]. An expansion of this technique to cover generalized labels, file I/O and IPC might be useful in securing Perl-specific Web applications.

Relative to Flume, the library and interpreter techniques have several practical advantages. First, they are likely easier to implement; second, they are likely easier to program with; third they are more portable; and fourth they are likely incur a negligible performance penalty. These techniques would protect programmers against their own bugs, so long as those bugs do not allow arbitrary code to execute and disable the application's self-checking mechanism. Such an assumption would not hold for a language like C (prone to buffer overruns) or if the attacker could fork a new process that reads data files directly off the file system (and not through the tracking library).

These techniques also do not apply to systems composed of many different language technologies stitched together. For instance, large Web sites often consist of many components, built by many engineers with differing tastes, and therefore in multiple languages. Sometimes these sites require compiled C code when performance matters; other times, scripting languages suffice for rapid prototyping; and system administrators might prefer shell-scripting and pipelines while accessing sensitive site data. These circumstances require security controls at the common interface, which on many Unix systems is the system-call interface.

### 10.4.2 Virulent Programmer Bugs

Security becomes more complicated if programmers wish to protect themselves against more virulent bugs that allow arbitrary code execution, or if they wish to build a system out of multiple languages. Defense in this context leads to a system like Flume, that enforces security controls at the system-call level. This approach has several important advantages; security controls (1) cannot be disabled by a compromised application; (2) are available to all programs regardless of language; (3) can cover IPC and file I/O in all cases.

*Covert channels* emerge as a key complication in this threat model. A covert channel is means of inter-process communication not explicitly modeled (and therefore, not controlled). However, to exploit such a channel, an attacker must control at least two code paths on the system: one to encode and send the information and another to decode and receive it. Typically such communication is difficult or impossible without general control of one or more server processes (such as the attack in Section 3.4). Thus, covert channels are assumed not possible in Section 10.4.1 but become possible in the presence of more virulent programmer bugs.

A covert channel on Flume, for instance, is $p$ monopolizing CPU (or sitting idle) while $q$ queries the CPU load (perhaps by measuring latency between its instructions). $p$ can encode a "1" as CPU monopolization and a "0" as quiescence. Of course, Flume does not capture the notion of CPU scarcity, or that the CPU is shared among all processes. Therefore it neither models this channel nor provably prevents it. In general, whenever two processes $p$ and $q$ share a physical resource, they can communicate in this manner. Resources include: cache lines on the CPU, memory, memory bus bandwidth, hard disk space, hard disk arms, network bandwidth, etc. Flume does not protect against these covert channels, nor do other DIFC kernels like Asbestos and HiStar (though HiStar solves some resource-exhaustion channels).

The key issue to consider about covert channels is how fast they leak information, and how observable those leaks are. One can imagine that on a Flume machine with only two active processes ($p$ and $q$), $p$ can encode many bits as disk or cache access patterns, and have $q$ reliably observe them. However, machines in Web-based clusters are typically busy, with many active processes vying for control of the machine's resources. In such an environment, the covert channel between $p$ and $q$ becomes much noisier, and therefore, the rate of information transmission between them must drop. Without having any real experience with covert channels on real system, we conjecture that long-running covert channel attacks (due to noisy and slow channels) are likely to be noticed by site administrators in well-managed server environment. However, more experiments on real systems, under real workloads are needed to quantify these threats.

A final point about covert channels is that they are more complicated and far less convenient than the *overt* channels typically used to leak data from Web systems. In this sense, even a system like Flume that is susceptible to covert channels offers a substantial security improvement over the status quo.

### 10.4.3 Invited Malicious Code

The most difficult attack to defend against is the malware attack, in which the administrator actively invites malicious code onto the server and allows it to compute over sensitive data. Here, the adversary can run many processes, and pound on covert channels without raising an eyebrow. As discussed below, we believe that a Flume system susceptible to these attacks (and others) is still more secure than the Facebook-like architectures in use today.

## 10.5 Generality and Future Directions

MoinMoin Wiki is a compelling target application for Flume due to its generality. Wikis support file creation, link creation among files, namespace management, access control right management, version control, indexing, etc. In other words, wikis can be thought of as general file-systems with user-friendly interfaces, building blocks for many features available on the Web today. In this sense, the wikilaunch application might be more general than just MoinMoin Wiki application, and apply to other applications (like Photo-sharing, blogs or social-networking) with small modifications. The hope, eventually, is for an administrator to run a suite of interesting Web applications using only one wikilaunch declassifier, keeping his TCB fixed while expanding to new features.

One perspective on Web platforms like Facebook and OpenSocial is that they are generalizations of wikis. Like a wiki, Facebook allows users to upload files, images, movies and documents; it allows groups of users to contribute to the same document, say by all adding captions to the same picture; it also features access control policies, more powerful than MoinMoin's in some cases (such as "friend-of-a-friend" permissions). At a conceptual level, third-party Facebook applications are reminiscent of third-party MoinMoin plug-ins. In both cases, the plug-in code has access to the important underlying data; there are differences, though, such as where does the code actually run (on the same server as the main application or on a third-party server), who can add new modules, who can use new modules, etc.

One idea for reinventing extensible Web platforms, which we call "W5," is to implement them as generalized wikis: to allow contributors to upload both content and code [59]. W5's design follows from our current implementation of FlumeWiki. The wikilaunch script remains the important declassifier, but the wiki software (wiki.py) itself is replaced by arbitrary uploaded code. Even if uploaded code is malicious, it is still subject to DIFC rules. Therefore, users of the W5 system can experiment with new uploaded applications without fear of losing their data. Similar policies pertain to integrity protection: only high-integrity application code can overwrite high-integrity data stored on the W5 server. Reputation of code authors or editorial oversight can establish which pieces of the system get a high-integrity certification.

W5 presents challenges that Flume does not yet meet. As mentioned previously, some covert channels unclosed in Flume can allow information leaks. W5 must prevent against resource-exhaustion: attacks by malicious code meant to prevent good code from doing useful work. Web sites more general than MoinMoin might require a database backend that stores data in individually labeled rows. Several of these problems have related solutions in the literature: Google's App Engine [39], Amazon Web Services [1], and various virtual machine approaches provide means of isolation among mutually distrustful code modules written by different authors. The SeaView database applies traditional MAC rules [63]. And on a busy Web-server with many applications contending for resources, covert channels based on resource exhaustion might prove noisy and therefore slow.

W5 must also address browser-based attacks. For instance, imagine Charlie uploads a third-party application to W5, which Alice later uses. Charlie's application reads Alice's secrets from the file system, and due to the policy enforced by wikilaunch, cannot send that data back to

Charlie's browser. However, Charlie's code can send Alice's data to Alice's browser and then ask Alice's browser to send that data to Charlie's server. For instance, it could send HTML Alice, instructing her browser to load the image `http://charlie.com/`$s$, where $s$ is Alice's secret. Charlie can then monitor the access logs on `charlie.com` to recover $s$. Charlie can also use JavaScript or Ajax calls to achieve the same ends. Charlie has other attack possibilities: he can instruct Alice's browser to post her secret data back to the W5 site, but to a public forum. Or he can disrupt the integrity of Alice's data, by instructing her browser to overwrite her high-integrity server-side data.

The solution to these browser-based attacks is to consider the Web browser as part of the Web system, and to track information flow between the server and client. That is, Alice's browser runs with a secrecy and integrity label, ensuring that data movement from the W5 site, to Alice's browser and back to the W5 site obeys all of the same DIFC rules already present on the server. The SIF work achieves some of these properties, but under the assumption that the site user (e.g., Alice) and not the programmer (e.g., Charlie) is malicious [12]. W5 needs to employ either similar language-based approaches, or perhaps browser modifications to enable dataflow tracking. As it does so, it might be providing general solutions for XSS and XSRF attacks.

## 10.6 Conclusion

When researchers laid the groundwork for computer security in the sixties and seventies, they experimented with several styles of access controls, such as discretionary access control lists, or more military-style mandatory access control. As the PC rose to prominence in the eighties, it forced users and developers down the discretionary path (when using access control at all). The problem is that the discretionary style of access control does not work against modern threats: the prevalence of malware and the deteriorating state of Web security have proven so.

With Flume, this thesis aims to show that information flow tracking can work on popular systems, and that DIFC offers a strong solution to otherwise intractable security problems. With Web technologies maturing and aiming to be center stage for the next phase in computing, now is the perfect time to set developers on the right access-control trajectory. To this end, we hope that DIFC can transcend academic research and secure future mainstream computing systems.

# Appendix A

# More on Non-Interference

## A.1 Unwinding Lemma

Ryan and Schneider [88] allude to an "unwinding" result for non-interference, which states that if a CSP process $P$ shows a non-interference property for each of its states, then it shows the same property over all traces [87]. Unfortunately, best efforts to recover the publication, whether in paper or electronic form, have failed. We recreate a proof of the "easy" half of the unwinding result here, though the proof of Flume's non-interference does not require it.

*Lemma* 1. If $\forall a, a' \in \mathit{traces}(S)$ such that $a \approx_L a'$:

1. $\mathit{initials}(S/a) \cap L = \mathit{initials}(S/a') \cap L$
2. $\mathit{refusals}(S/a) \upharpoonright L = \mathit{refusals}(S/a') \upharpoonright L$

then $\forall b, b' \in \mathit{traces}(S)$ such that $b \approx_L b'$: $\mathcal{SF}[\![S/b]\!] \upharpoonright L = \mathcal{SF}[\![S/b']\!] \upharpoonright L$

In other words, if we can prove that after two equivalent traces, $S$ still accepts and rejects the same events, then $S$ exhibits non-interference.

**Proof** We show the equality of the two failure sets by proving each is included in the other. Take an arbitrary $f \in \mathcal{SF}[\![S/b]\!] \upharpoonright L$. The goal is to prove that $f \in \mathcal{SF}[\![S/b']\!] \upharpoonright L$. Write $f = (c, X)$. Thus, there exists some $f' = (c', X')$ such that $c = c' \upharpoonright L$, $X = X' \cap L$ and $f' \in \mathcal{SF}[\![S/b]\!]$, by definition of projection over failures.

First consider the set of refused events $X'$. Since $(c', X') \in \mathcal{SF}[\![S/b]\!]$, it follows that $X' \in \mathit{refusals}(S/(b \frown c'))$, and applying projections to both sides, $X' \cap L \in \mathit{refusals}(S/(b \frown c')) \upharpoonright L$, or equivalently, $X \in \mathit{refusals}(S/(b \frown c')) \upharpoonright L$. By assumption, $b \approx_L b'$. By definition of $c$ and $c'$, we also have that $c' \approx_L c$. Thus, their concatenations are also equivalent when projected over $L$. That is, $(b \frown c') \approx_L (b' \frown c)$. Thus, $X \in \mathit{refusals}(S/(b' \frown c)) \upharpoonright L$, since $\mathit{refusals}(S/(b' \frown c))$ = $\mathit{refusals}(S/(b \frown c'))$ by our assumption.

Next, consider the trace portion of $f'$, denoted $c'$. $(c', X') \in \mathcal{SF}[\![S/b]\!]$ implies that $b \frown c' \in$ *traces*$(S)$. Our goal is to show that $c \in$ *traces*$(S/b') \restriction L$. Write the trace $c'$ in terms of a sequence of events:

$$c' = \langle e_1, e_2, \ldots, e_n \rangle$$

For each $i$, we have that $e_i \in$ *initials*$(S/(b \frown \langle e_1, \ldots, e_{i-1} \rangle))$ by definition. If $e_i \in L$, then it follows that $e_i \in$ *initials*$(S/(b \frown \langle e_1, \ldots, e_{i-1} \rangle)) \cap L$. Using the same logic as above, $b \approx_L b'$ implies that $(b \frown \langle e_1, \ldots, e_{i-1} \rangle) \approx_L (b' \frown \langle e_1, \ldots, e_{i-1} \rangle)$, and hence:

$$\textit{initials}(S/(b \frown \langle e_1, \ldots, e_{i-1} \rangle)) \cap L = \textit{initials}(S/(b' \frown \langle e_1, \ldots, e_{i-1} \rangle)) \cap L$$

by our assumption. Thus, if $e_i \in L$, then $e_i \in$ *initials*$(S/(b' \frown \langle e_1, \ldots, e_{i-1} \rangle)) \cap L$. Take the maximal $i$ such that $e_i \in L$, and call it $i^*$. By the definition of traces, it follows that: $\langle e_1, \ldots, e_{i^*} \rangle \in$ *traces*$(S/b')$. And also, since the sequence $\langle e_1, \ldots, e_{i^*} \rangle$ includes all events from $c$, we have that $\langle e_1, \ldots, e_{i^*} \rangle \restriction L = c$, which shows that $c \in$ *traces*$(S/b')$.

Thus, $f \in \mathcal{SF}[\![S/b']\!] \restriction L$, and consequently, $\mathcal{SF}[\![S/b]\!] \restriction L \subseteq \mathcal{SF}[\![S/b']\!] \restriction L$. The other inclusion follows by symmetry. ∎

# Bibliography


[1] Amazon. Amazon Web Services. `http://aws.amazon.com`.

[2] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of19th ACM Symposium on Operating Systems Principles*, pages 90–105, Bolton Landing, Lake George, New York, October 2003.

[3] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *Proceedings of Sixteenth Systems Administrator Conference (LISA)*, Berkeley, CA, 2002.

[4] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report Technical Report 2547, Volume I, MITRE Corparation, Bedford, MA, March 1973.

[5] D. Elliott Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corparation, Bedford, MA, March 1976.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 1–15, August 1996.

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

[8] Gunther Birznieks. CGI/Perl Taint Mode FAQ, 1998. `http://gunther.web66.com/FAQS/taintmode.html`.

[9] Micah Brodsky, Petros Efstathopoulos, Frans Kaashoek, Eddie Kohler, Maxwell Krohn, David Mazières, Robert Morris, Steve VanDeBogart, and Alexander Yip. Toward secure services from untrusted developers. Technical Report TR-2007-041, MIT CSAIL, August 2007.

[10] CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests, 2000. `http://www.cert.org/advisories/CA-2000-02.html`.

[11] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web applications via automatic partitioning. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[12] Steven Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in Web applications. In *Proceedings of 16th USENIX Security Symposium*, Boston, MA, August 2007.

[13] Neil Chou, Robert Ledesma, Yuka Teraguchi, Dan Boneh, and John C. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2004.

[14] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of11th USENIX Security*, San Francisco, California, August 2002.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

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

[18] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings ofthe SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590, 2006.

[19] T. Dierks and E. Rescorla. The transport layer security (tls) protocol, version 1.1. Technical report, Network Working Group, April 2006.

[20] Chad R. Dougherty. Vulnerability note VU #80013: Multiple DNS implementations vulnerable to cache poisoning. Technical report, United States Computer Emergency Readiness Team, July 2008. `http://www.kb.cert.org/vuls/id/800113`.

[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[22] David Endler. The evolution of cross site scripting attacks. Technical report, iDEFENSE Labs, 2002.

[23] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[24] Facebook. Facebook developers wiki. `http://wiki.developers.facebook.com/index.php/API`.

[25] Facebook.com. `http://www.facebook.com`.

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

[27] Justin Fielding. UN website is defaced via SQL injection. *Tech Republic*, August 2007. `http://blogs.techrepublic.com.com/networking/?p=312`.

[28] FIPS 180-1. *Secure Hash Standard.* U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[29] FIPS 180-2. *Secure Hash Standard.* U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, August 2002.

[30] Django Software Foundation. Django.

[31] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of2000 IEEE Symposium on Security and Privacy*, pages 230–245, Oakland, CA, May 2000.

[32] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proceedings ofIEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[33] Stefan Frei, Thomas Dübendorfer, Gunter Ollmann, and Martin May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the "insecurity iceberg". Technical Report 288, ETH Zurich, January 2008.

[34] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2004.

[35] Jacques Gelinas. Virtual private servers and security contexts, January 2003. `http://linux-vserver.org`.

[36] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Research in Security and Privacy*, 1982.

[37] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, 1973.

[38] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):210–217, 1986.

[39] Google. Google App Engine. `http://code.google.com/appengine`.

[40] Google.com. Opensocial. `http://code.google.com/apis/opensocial/`.

[41] Andy Greenberg. Google's opensocial could invite trouble. *Forbes.com*, November 14 2007. `http://www.forbes.com/2007/11/13/open-social-google-tech-infrastructure-cx_ag_1114open.html`.

[42] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[43] Norman Hardy. The confused deputy: (or why capabilities might have been invented). 22(4), October 1988.

[44] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *Proceedings of22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[45] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. Integrating selinux with security-typed languages. In *Proceedings of the 3rd SELinux Symposium*, March 2007.

[46] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall International, Englewood Cliffs, New Jersey, 1985.

[47] A. Householder, K. Houle, and C. Dougherty. Computer attack trends challenge Internet security. *Computer*, 35(4):5–7, Apr 2002.

[48] Jeremy Jacob. On the derivation of secure components. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1989.

[49] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of14th Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, December 1993.

[50] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 1–10, September 2006.

[51] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of 2nd International System Administration and Netwroking Conference (SANE)*, Maastricht, NL, May 2000.

[52] Gregg Keizer. FAQ: The monster.com mess, August 2007. `http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9032518`.

[53] Key Logic. *The KeyKOS/KeySAFE System Design*, sec009-01 edition, March 1989. `http://www.agorics.com/Library/KeyKos/keysafe/Keysafe.html`.

[54] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing, Security Track*, April 2006.

[55] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of 11th USENIX Security*, August 2002.

[56] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of 10th Hot Topics in Operating Systems Symposium (HotOS-X)*, Santa Fe, New Mexico, June 2005.

[57] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[58] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[59] Maxwell Krohn, Alexander Yip, Micah Brodsky, Robert Morris, and Michael Walfish. A World Wide Web Without Walls. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networks (HotNets)*, Atlanta, GA, November 2007.

[60] Robert Lemos. Payroll site closes on security worries. *Cnet News.com*, February 2005. `http://news.com.com/2102-1029_3-5587859.html`.

[61] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 16–27, 2006.

[62] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of 2001 USENIX Annual Technical Conference*, San Diego, CA, June 2001. FREENIX track.

[63] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The SeaView Security Model. *IEEE Transactions on Software Engineering*, 16(6):593–607, 1990.

[64] Caroline McCarthy. Facebook dumps Secret Crush application over spyware claim. *Cnet News.com*, January 7 2008. http://news.cnet.com/8301-13577_3-9843175-36.html.

[65] Joe Mcdonald. China says web users top u.s. at 253 million. *Associated ProcesS*, July 25 2008.

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

[67] Mark Miller and Jonathan S. Shapiro. paradigm regained: Abstraction mechanisms for access control. 2003.

[68] The MoinMoin Wiki Engine, December 2006. http://moinmoin.wikiwikiweb.de/.

[69] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malô, France, October 1997.

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

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

[72] News10. Hacker accesses thousands of personal data files at CSU Chico, March 2005. http://www.news10.net/display_story.aspx?storyid=9784.

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

[74] J. Ouaknine. A framework for model-checking timed CSP. Technical report, Oxford University, 1999.

[75] Bryan Parno, Cynthia Kuo, , and Adrian Perrig. Phoolproof phishing prevention. In *Proceedings of the 10th International Conference on Financial Cryptography and Data Security*, Anguilla, British West Indies, February 2006.

[76] PostgreSQL. http://www.postgresql.org.

[77] Francois Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[78] Kevin Poulsen. Car shoppers' credit details exposed in bulk. *SecurityFocus*, September 2003. http://www.securityfocus.com/news/7067.

[79] Kevin Poulsen. Ftc investigates petco.com security hole. *SecurityFocus*, December 2003. `http://www.securityfocus.com/news/7581`.

[80] Niels Provos. Improving host security with system call policies. In *Proceedings of 12th USENIX Security Symposium*, Washington, DC, August 2003.

[81] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of First Workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, April 2007.

[82] Inc. Red Hat. JBoss enterprise middleware. `http://www.jboss.org`.

[83] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, pages 249–261, 1988.

[84] Ivan Ristić. Firefox 3 improves handling of invalid SSL certificates, April 2008. `http://blog.ivanristic.com/2008/04/firefox-3-ssl-i.html`.

[85] A. W. Roscoe. *A Theory and Practice of Concurrency*. Prentice Hall, London, UK, 1998.

[86] A.W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[87] Peter A. Ryan. A CSP formulation of non-interference and unwinding. *Cipher: IEEE Computer Society Technical Committee Newsletter on Security & Privacy*, pages 19–30, 1991.

[88] Peter A. Ryan and Steve A. Schneider. Process algebra and non-interference. *Journal of Computer Security*, (9):75–103, 2001.

[89] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[90] Jerome H. Saltzer. Protection and the control of information sharing in the multics system. *Communications of the ACM*, 17(7), July 1974.

[91] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[92] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons, LTD, Chichester, UK, 2000.

[93] Bruce Schneier. Two-factor authentication: too little, too late. *Communications of the ACM*, 48(4):136, 2005.

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

[95] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, October 2004.

[96] Jonathan S. Shapiro, Jonathan Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of17thACM Symposium on Operating Systems Principles*, Kiawah Island, SC, October 199.

[97] 37 Signals. Ruby on rails. `http://www.rubyonrails.org/`.

[98] 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`.

[99] Nir Soffer. MoinBenchmarks. `http://moinmoin.wikiwikiweb.de/MoinBenchmarks`.

[100] Chris Soghoian. Hackers target Facebook apps. March 27 2008. `http://news.cnet.com/8301-13739_3-9904331-46.html`.

[101] IBM Internet Security Systems. X-force®2008 mid-year trend statistics. Technical report, IBM, 2008. `http://www-935.ibm.com/services/us/iss/xforce/midyearreport`.

[102] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *Proceedings of 2006 Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, November 2006.

[103] Rebecca Trounson. Major breach of UCLA's computer files. *Los Angeles Times*, December 12 2006.

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

[105] Helen J. Wang, Xiaofeng Fan, Collin Jackson, and Jon Howell. Protection and communication abstractions for Web browsers in MashupOS. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[106] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of 2003 USENIX Annual Technical Conference*, pages 285–296, San Antonio, TX, June 2001.

[107] Dan Wendlandt and Ethan Jackson. Perspectives : Improving ssh-style host authentication with multi-path network probing. In *Proceedings of 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008.

[108] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[109] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[110] Edward Z. Yang. HTML purifier. `http://htmlpurifier.org`.

[111] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, Massachusetts, April 2007.

[112] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.

[113] Nickolai B. Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[114] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proceedings of 2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, August 2004.

[115] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, 2007.