PERVASIVE BINDING OF LABELS TO SYSTEM PROCESSES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Florian Buchholz

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2005

Für meine Eltern

ACKNOWLEDGMENTS

There is a large number of people without whom my work at Purdue would not have been possible. Special thanks go to my family – my father Eberhard, my mother Erika, my sister Susanne, and my niece Tekla – for always being there for me, their love and support. Many thanks also to my "American family", Vivian and Kip Kistler, Yuliya and Jason Kistler, and Kiera and Josh Dubach. You adopted me to your family and always welcomed me with open arms.

I would like to thank my committee for their support and insightful feedback, especially my advisor Gene Spafford (Spaf) for his guidance, help and support. Special thanks also go to my former advisor Clay Shields, with whom I developed the early ideas for my work.

I have made many friends during my time at Purdue, and they provided me with much needed distractions from my studies, shoulders to lean on, and much happiness. A very special thank you goes to Marina Bykova for dancing with me and being a good friend, and Jim Early for taking many paths together and being as great a friend as one could want. My former and current office mates Tom Daniels, Ben Kuperman, and Brian Carrier also have become good friends and I appreciate our stimulating discussions as well as the stimulating beverages we consumed. Scott Robinson, my former room mate deserves a thank you for putting up with me, and he and Tia have become good friends to me. I would like to thank Ed Finkler for our quests to discover good music in this State of Indiana. Thanks to Courtney Falk for his collaboration. Thanks also go to the "New Year" crew: Sara and Kevin Miller, Mark Harbaugh, Jeff and Sarah Wagner, Kyle Ham, Sarah "Sprite" Daley, and Stephanie Popper; the "office" crew (current and past): Krista Bennett, Abhilasha Spantzel, Sundararaman Jeyaraman, Rajeev Gopalakrishna, Paul Williams, Mahesh Tripunitara, Sofie Nystrøm, Anya Berdichevskaya, Mat Baarman, Keith Turner,

Hao Wu, and Chris Telfer; the “dance” crew: Adrienne Foster, Frank Scharf, Corrie Vandervlugt, Dave Zage, and Tyler Hershberger; the “beer” and “wine” crews.

Last but not least my thanks go out to CERIAS and the Computer Sciences Department. Many thanks to the CERIAS sponsors for their financial support and collaboration. A big thank you to the hard-working staff at CERIAS who made my life so much simpler: Adam Hammer, Ed Cates, Marlene Walls, Mary Jo Maslin, Carol Dyrek, Debbie Frantz, Laurie Floyd, Randy Bond, Jennifer Kurtz, and Tersa Bennett. Many thanks also to the staff in the CS graduate office, Dr. Gorman, Amy Ingram, and Susan Deno.

There are many more who have touched my life at Purdue in a positive way, and to all of you I extend my deepest gratitude. Thank you very much!

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT x

1 Introduction 1

1.1 Background and Problem Statement 1

1.2 Thesis Statement 3

1.3 Document Organization 4

2 Background and Related Work 5

2.1 System Information 5

2.1.1 Log Files 6

2.1.2 File System Metadata 8

2.1.3 Digital Forensics 11

2.2 Information Flow Analysis 14

2.2.1 Information Flow Policies 14

2.2.2 Static Information Flow Analysis 18

2.2.3 Dynamic Analysis 18

2.3 Network Traceback 19

2.3.1 Packet Source Determination 20

2.3.2 Correlating Streams 25

3 Adding Audit Information 28

3.1 Types of Information 28

3.1.1 The Relevance of Metadata for Forensics 30

3.2 Desired Information 32

3.2.1 Who Did It? 33

Page

3.2.2 Where Did That Come From? . . . 36
3.2.3 When Did What Happen? . . . 39
3.2.4 How Did It Happen? . . . 42
3.2.5 What Was Done to the File? . . . 43
3.2.6 User Influence and Location Information . . . 43
4 A Model for Label Propagation Based on Causality . . . 45
4.1 Causality . . . 45
4.1.1 Labels . . . 47
4.2 A General Model of Label Propagation Between Principals Based on Causality . . . 48
4.2.1 Covert Channels . . . 57
4.3 Space Analysis . . . 58
4.3.1 Enforcing Space Constraints . . . 63
4.4 Properties of the Propagation Model . . . 65
4.5 Case Studies . . . 72
4.5.1 User Influence Labels . . . 72
4.5.2 Host Causality Labels . . . 74
4.5.3 Network Location Traceback Labels . . . 75
4.5.4 Military Classification Labels . . . 76
5 Implementation . . . 80
5.1 Subsystems Affected by Label Propagation . . . 81
5.1.1 Shared Resources . . . 82
5.1.2 Interprocess Communication . . . 84
5.1.3 Operations and System Calls . . . 85
5.2 Data Structures and Operations . . . 93
5.3 IPC: Sockets . . . 96
5.4 Shared Resources: Files . . . 100
5.5 Results . . . 103

Page

5.5.1 User Influence 103
5.5.2 Location Information 106
5.5.3 Remote System Compromise 107
5.5.4 Performance Overhead 109
6 Conclusions 116
LIST OF REFERENCES 127
Appendix A: Detailed Performance Results 135
VITA 139

LIST OF TABLES

Table		Page
5.1	Processor and process tests – times in μs	112
5.2	File system tests – times in μs	112
5.3	Network latency tests – times in μs	114
5.4	I/O bandwidth tests – in MB/s	114
1	`FreeBSD` kernel results	135
2	`Label-0` kernel results	136
3	`Label-s` kernel results	137
4	`Label-l` kernel results	138

LIST OF FIGURES

Figure		Page
2.1	Distributed denial of service model	21
2.2	A sample chain of remote connections	26
2.3	Incoming and outgoing network traffic can no longer be related because data is transformed.	27
3.1	The contents of File 2 are influenced by User A	35
3.2	Given the origin of the involved entities, what is the origin of the new file?	38
4.1	Information flow for each operation and the final label sets	55
5.1	Data flowing through a channel	88
5.2	The main kernel data structure for labels	94
5.3	FreeBSD kernel functions for socket I/O	98
5.4	Output from the user influence test from both user sessions	105
5.5	Output from the location information case study	107
5.6	An `ssh` session with location labels	108

ABSTRACT

Buchholz, Florian. Ph.D., Purdue University, August, 2005. Pervasive Binding of Labels to System Processes. Major Professor: Eugene H. Spafford.

It is desirable to be able to gather more forensically valuable audit data from computing systems than is currently done or possible. This is useful for the reconstruction of events that took place on the system for the purpose of digital forensic investigations. In this document, we analyze what kind of information is desired and what is lacking in computing systems. We then propose a mechanism that allows arbitrary information from a system to be propagated based on causality influenced by information flow. We further discuss how to implement such a mechanism for the FreeBSD operating system and present a proof-of-concept implementation that has little overhead compared to the system without label propagation.

1 INTRODUCTION

It is desirable to be able to gather more forensically valuable audit data from computing systems than is currently done or currently possible. This is useful for the reconstruction of events that took place on the system for the purpose of digital forensic investigations. In this document, we analyze what kind of information is desired and what is lacking in computing systems. We then propose a mechanism that allows arbitrary information from a system to be propagated based on causality influenced by information flow.

1.1 Background and Problem Statement

Security mechanisms on computing systems, such as intrusion detection systems, access control, and audit facilities rely on the information that is available on the system on which they are deployed. The information that is available on computing systems about the events that occur, however, did not evolve from the need for good security data but rather from the need to manage the available shared resources of the system among its users. If the need arises to investigate an incident, as part of a forensics investigation or incident response, the amount of information that is actually stored on a permanent basis is often further reduced, making it difficult to draw sound conclusions from them. Most of the effort to date in the digital forensics community has been in the retrieval and analysis of existing information from computing systems. Little has been done to increase the quantity and quality of the forensic information on computing systems.

An operating system's main function is to administer the limited available resources to the programs that request them. Thus, much of the information a system keeps about its processes and objects is related directly to the task of administering

those resources. A large part of this information is kept for reasons of access control. Other important security concerns do not play a prominent role. While processes usually carry a user identifier, it may be unclear whether this user is truly responsible for the actions the process performs. Furthermore, there is no notion at all about location, or origin, of a system's processes and objects. From where was a session initiated? Where did a file come from?

Third party extensions exist that add more information for the purpose of access control [106] but also for detecting policy violations [56, 118], adding a sense of location [30, 106] or more general security audit mechanisms [60, 101]. Logging facilities such as `syslog` for UNIX and the Event Viewer mechanism for the Windows platform may record extra information about events as they occur. However, most of these extensions lie outside the system itself, which means that they may be subject to tampering. Event logs need to be correlated to establish any causal relationship between events. This is at best a tedious task, but may even turn out to be impossible in certain situations because not enough information is recorded or propagated.

When adding information to a system, the goal is to preserve event relationships. From a security perspective, one wants to determine answers to questions such as "who is responsible for events?" or where the entities that caused the events are located in the world (or network). To achieve this, it is not sufficient to simply have one user identifier per process or file, or introduce a location field. This is because a principal acting within a system is influenced by other principals or by the contents of objects such as files. This causality is governed by the information flow among principals and objects on the system. A principal communicating with another principal may influence/force/ask/trick/signal the latter into performing some sort of action. The same is true for the content of an object that a principal accesses. This dissertation presents a model that allows the system to bind arbitrary information in the form of a label to its principals. Labels are then propagated to other principals and objects on the system as information is exchanged between them. Depending

on the nature of such a label, e.g. user identity or location information, valuable audit data can be created on a system. This is especially useful for digital forensics, intrusion detection, network traceback, and access control.

The information output by a principal can be described as its observable actions. In many cases, these outputs are triggered in response to the principal's inputs, i.e. the information entering the principal. Thus, we may say that certain inputs *cause* certain outputs of a principal. We will give a more detailed discussion of causality in Section 4.1.

In our work, labels are propagated based on information flow between subjects. This includes cases where one principal (or more) controls the actions of another. Our approach differs from traditional information flow analysis methods in the way that we do not attempt to determine how information actually is exchanged but rather how information could have been exchanged. We will focus on two categories of information that may be desired during an investigation but current operating systems cannot supply. These two categories are *user influence* and *location information* (host causality [12]).

Location information is especially important for the area of network traceback [10], where the goal is to trace attackers back through the network to the ultimate source of an attack. As we will show in Chapter 2, existing research only takes into consideration data obtained from a network perspective, neglecting traffic transformations that can be performed on a host to obfuscate any relations among network packets or streams.

1.2 Thesis Statement

It is possible to add significantly useful audit information to a system with little computational overhead by binding labels that convey information such as user identity or location information to principals on the system and propagate those based on how information flows between principals and objects.

1.3 Document Organization

This document is organized into six chapters, including this introductory chapter. Chapter 2 gives an overview of the problem and discusses related work. Chapter 3 addresses the question of what kind of audit information is desired on computing systems to aid in forensic investigations. This reflects work regarding the relation between audit data and file system metadata for forensic purposes [13]. In Chapter 4 we present a model that can be used to obtain some of the information that is desired but currently cannot be collected [14]. Chapter 5 discusses how to implement the model for the FreeBSD platform and show an actual proof-of-concept implementation and results. We give concluding remarks, discuss the limitations of our approach and outline future work in Chapter 6.

2 BACKGROUND AND RELATED WORK

Computing systems are generally not designed with security in mind. While operating systems exist that conform to the Orange Book's [31] security divisions of B-2 (Multics, VSLAN, Xenix), B-3 (XTS-300), and A-1 (Boeing MLS LAN, Gemini Trusted Network Processor, Honeywell SCOMP), these are not widely distributed and limited in their usability. Other approaches such as Microsoft's efforts to secure its Windows XP platform and the SELinux project [64] are intended to enhance security on the system, but inherit the design from their "ancestor" operating systems. Because of the lack of systems designed for security, the amount and quality of audit data that is useful for digital forensics, network traceback, or intrusion detection is small. This chapter gives an overview of what kind of information is inherently available by current computing systems, and what research has been done to improve the amount, quality, and integrity of audit data. We further give an overview of network traceback research and explain what data is lacking in that area. Finally, we discuss past research in information flow analysis to establish how our approach differs from previous work.

2.1 System Information

When gathering audit data on a system, we are interested in information that describes the events that took place within the system. This data can then be used to detect policy violations, profile system behavior, or to reconstruct what events occurred and how, if the need arises. Audit information can either be present at runtime, or it can be preserved in permanent objects. The information that is kept in the system's long-term storage can either be recorded explicitly in the form of audit logs, or it can be a byproduct of a system object's meta information (e.g. file system

metadata). In the latter case the meta information might not have been stored with the intent of creating audit data about the system, but the very nature of describing some of the attributes about the object may lead to conclusions regarding events that involved the object itself.

When a computer system is running, and while non-persistent objects reside on the system, the amount of available audit information is at its largest. Running processes on a system carry with them information about the programs they are executing, the users who executed those programs, open files and sockets, etc.

Valuable audit information may be lost when network connections or file descriptors are closed, processes terminate, or data that is not needed anymore for the immediate execution of programs is discarded. In some cases information may persist in the system memory for a certain amount of time [36], but eventually the information will fade away. For this reason, certain information is stored on a more permanent basis. This is either done in log files, or implicitly by the system through meta information associated with long-term storage objects.

In the following we will give a brief overview of the sources of audit information on a system and what kind of information is available.

2.1.1 Log Files

The most obvious location to preserve audit data on a system is by storing them in the system's long-term storage objects (files) as *log files*. These log files can originate from the system itself (e.g. user login events, access control violations, firewall data, or changes in the system's configuration), or applications can supply log information about the events they can witness. In either case, the information is stored in files, which inherently brings with it the danger of tampering or deletion. Sometimes logs may be protected by cryptographic mechanisms [92] or by the more common techniques of being written to write-only media or being sent to a more secure central logging facility. In general, however, this is not commonly done.

Many operating systems provide some sort of logging facility for system events. For the UNIX-like operating systems, this is the `syslog(3)` facility and for the more recent versions of the Windows operating systems the Event Viewer system keeps a *Security log*, a *System log*, and an *Application log*. Programs outside the system may also choose to add log entries to those facilities as is frequently done by intrusion detection systems, firewalls and network daemon processes.

`Tcpwrappers` [106] is a tool that allows a system administrator to set up access control policies for accepting network connections for well-known services based on where those connections originate. Apart from authenticating network connections `tcpwrappers` can also be used to collect extensive log information about the connections, whether they are accepted or rejected.

Tripwire is a tool developed by Kim and Spafford [56]. Tripwire monitors changes in the contents and metadata of a list of file as well as the creation and deletion of new files, and compares the observed behavior against a system policy.

Denning and MacDoran proposed an access control mechanism based on GPS location information [30]. The GPS information is used to grant or deny access to system services, but one can easily imagine extending the approach to logging the information about those accesses.

Zamboni describes how to embed small modifications into system and application code to act as sensors that record the actions they observe [118]. The information is logged and then analyzed to detect known intrusions. Previously unknown intrusions that exhibit patterns similar to the known intrusions can be detected that way, as well.

A more thorough analysis of audit data on systems is provided by Kuperman [60]. His work classifies audit sources into categories of how they may be utilized, such as intrusion detection and computer forensics. The problem of how events logged by different sources on a system correlate, and how this may not be captured by the individual audit logs is not discussed, however.

Apart from their vulnerability to tampering, log files are recorded at the time when the event occurred, but it might be unclear at that point how certain events relate to each other. To receive a more accurate picture of past events, log file entries have to be correlated, which is a tedious and complex undertaking. Furthermore, sometimes correlation is impossible because it may not be decidable which event(s) might have caused other events to take place. This might be because insufficient information is recorded, but also because information gathered by one program cannot be accessed or interpreted by a different one. Having the information of both sources available together to describe an event would yield more precise conclusions. For example, `tcpwrappers` might record the information about a session from a remote location while Tripwire detects an access violation of one of its policies in the system during the same time interval. When analyzing the log entries it remains unclear whether the access violation was caused as a result of the remote session.

2.1.2 File System Metadata

A valuable source of information about past events on a system is the long-term storage of the system. In many cases, additional data is associated with long-term objects, from which conclusions of past system events may be derived.

Most computing systems have some type of long-lived data storage that may be examined for evidence. The usual organization of this storage is comprised of files, directories, and metadata. For the remainder of this document we will assume such an organization. We define *metadata* as all the data in the file system that describes the layout and attributes of the regular files and directories. This includes attributes such as timestamps, access control information, and file size, and also information on how to locate and assemble a file or directory in the file system. This latter information contains pointers to data blocks, or even entire blocks used as internal nodes of lookup data structures such as B-trees.

File system metadata was not originally designed to be used for the purpose of reconstructing events that occurred on the system. Anderson [3] was the first to utilize such data for threat monitoring. He proposed to utilize System Management Facilities (SMF) records. These records were commonly used by mainframe server operating systems, such as IBM's OS/360.

In the 1960s most computing tasks were performed on mainframe computers, with OS/360 one of the dominating operating systems. Information stored on the servers' disks described the entire batch job of a user. The data for the jobs came from punch cards or tape media. The batch job information was kept in *records*, which described different aspects about the job, some describing the user data (which can be seen as a file). This included file type, minimum and maximum size, creation, access, and modification times, but also information about the job itself such as running times, duration and resources utilized. Compared to the metadata of current systems, there was more information available for the purpose of analyzing system events. This information is still maintained on systems at this date, but usually not recorded or only in a temporary fashion such as the *proc* file system for UNIX.

Multics was the first operating system that supplied a hierarchical file system, which is generally considered as the ancestor of most common file systems. Daley and Neumann describe in the Multics file system design paper [23] the need for users to store their data within the computing environment itself as opposed to storage media such as cards and tape. The user would have complete control and ownership of his data as well as the metadata. They formulated the following design objectives: "Little-used information must percolate to devices with longer access times, to allow ample space on faster devices for more frequently used files. Furthermore, information must be easy to access when required, it must be safe from accidents and maliciousness, and it should be accessible to other users on an easily controllable basis when desired." [23] To determine how frequently information was used, they proposed an access timestamp. The need for modification and creation times came from the file system's backup system, which would commit newly created and

modified files to tape backup. This is the original motivation for the use of the MAC ("Modified, Accessed, Changed") times. To be able to allow other users to access files, they proposed the inclusion of an access control list plus permissions (modes) for each file. All remaining metadata had to do with the actual on-disk layout of a file.

UNIX was introduced in the early 1970s [85] and its file system was strongly influenced by Multics. The metadata for a file was stored in an *inode* and it contained the file's location and size, its type (directory or file), the three timestamps, and the access control information. The latter was comprised of the user and group identifier and protection bits as all modern UNIX variants and derivatives use them.

MS-DOS emerged in the early 1980s. Its file system, FAT [69], keeps track of the file type, size, location, and the timestamps. The space reserved for timestamps varies between 2 and 4 bytes, which results in differences in granularity. For example, the access time is only measured in days. Because DOS did not have any notion of a user, no user or permissions information is stored with FAT. The Windows operating system at first inherited the FAT file system, but when the limitations of FAT became too much of a problem, NTFS was introduced. NTFS carries detailed user and permission information as well as modified, accessed, created, and changed timestamps.

Much previous work has been done in the area of versioning file systems. Versioning file systems store past versions of files in the file system and also the metadata associated with those versions. However, the primary focus here lies in data recovery and undoing of write operations. Systems including AFS, Plan-9 [80], and WAFL allow for setting of checkpoints for files on a periodic basis. The Cedar file system [44] as well as the RSX, VMS, and TOPS-20 operating systems create new versions of a file for each modification, but have limitations as to how many copies of a file may exist and simple heuristics to decide what versions to delete after that. The Elephant file system [90] also provides the ability to keep a long-term or even complete history of a file. However, the long-term history is only achieved by retaining user-defined

landmark versions and space considerations are not discussed for the complete history option. Furthermore, the versions kept of the files are focused on content only, ignoring metadata such as times of access and modification.

2.1.3 Digital Forensics

The area of digital forensics is concerned with the investigation of an incident after it has happened. For this purpose, *digital evidence* is gathered from the system and used to support hypotheses a forensic investigator may have about the incident. Such an investigation may be as simple as locating incriminating material, but in its most complex case, a reconstruction of all the past events on the system may be desired. In this section, we present the current state of digital forensics and discuss where shortcomings are, before we go into more detail about desired information for forensic investigations in Chapter 3.

Casey and Palmer define *forensic* as "... a characteristic of evidence that satisfies its suitability for admission as fact and its ability to persuade based upon proof (or high statistical confidence)." [20]. When applying this definition to digital forensics, one can see that the area consists of contributors from a wide spectrum of disciplines and backgrounds. Apart from computer science, digital forensics is relevant to practitioners of disciplines such as law, law enforcement, politics, or standardization bodies. Literature in the field of digital forensics thus has much material to cover and needs to address a diverse audience including digital crime scene technicians, digital evidence examiners, digital investigators [20], as well as lawyers, attorneys, judges, politicians, developers, and researchers.

Much of the current literature and guidelines for digital forensics focuses primarily on data retrieval and availability of information on existing systems. Given the diverse target audience and the different levels of expertise in computing, one of the main objectives is to teach practitioners the basic procedures of evidence retrieval and analysis on computing systems. Naturally, future research in the field of digital

forensics cannot be addressed in as much detail as desired by the research community, although more recent publications also discuss research. Most of the current work explains how to recover data from a system in one form or the other. In respect to data about system events, some of the work also discusses the forensic value and/or quality of the information that is found. By *forensic value* we understand the possibility to draw conclusions about events on the system from the data. For example, timestamps have a high value from an event reconstruction perspective because they allow an ordering of file operations into a timeline. This is provided that the timestamps have not been tampered with and that the system's clock is correct. Access control information on its own, however, holds less value from an event reconstruction point of view, because it generally only reflects static system policies, and does not provide information about individual events. The information that can be derived from access control information leads to a (group of) user(s) that may have had access to an object on the system. At this point further evidence (e.g. in the form of timestamps or login data) is needed to draw any conclusions. Under *forensic quality* we understand how trustworthy the information is. Is it easy to tamper with the information on the system? For example, on some operating systems a file's access and modification timestamps can be arbitrarily set by its owner.

For some instances of forensic literature, the discussion of evidence retrieval is limited to a description of where important system files are located and how to use tools that recover deleted files [22, 113]. Metadata is not discussed at all or only in the form of timestamps [113], and no critical discussion is given about the value of the forensic information. Other publications focus in great detail on the issue of information hiding and retrieval without mentioning file system metadata or issues such as how to obtain time, user, or location information [15].

Some of the current forensics literature actually addresses the value of file system metadata in the form of MAC times and user information [19, 59]. However, a critical discussion about the quality of the information is lacking. At some part of

the discussion timestamps are presented as a powerful means to reconstruct events. However, either no critical discussion is given [19], or the whole value of timestamp information is undermined by statements such as: "Altering the modify and access times in an inode is simple, but not every suspect knows how to do it" [59]. The notion of an "owner" of a file is mentioned [59] but the term "owner" is not explained and may lead to incorrect assumptions about the relationship between a user and a file. A more thorough discussion about event reconstruction is given by Casey [20]. The actual techniques in terms of functional, relational, and temporal analysis are described on a higher level than what information a system may (reliably) provide. However, the author makes it clear that timestamps may be altered and discusses techniques to detect the tampering or deduce the correct times of events. Carrier and Spafford [18] use the term *characteristics* of a digital object, the set of data and metadata associated with the object, in their event reconstruction model. This reflects the need for reliable metadata information for event reconstruction. They do not, however, discuss what the nature of these characteristics could or should be. Mohay et al. dedicate an entire chapter to research directions and future development [70]. Topics such as data mining, text categorization, authorship analysis, steganography, and cryptography are covered. In their section on evidence extraction they address the difficulty associating collected data from various sources with events on the system. They give a framework to correlate existing data on a system, whereas the purpose of our work is to analyze what data can be added and how its forensic quality can be maintained.

All of the surveyed literature only describes the information that can be obtained from existing systems and this is their intended purpose. File recovery is the main focus, but a few documents elaborate on the value of timestamps or user information. In our survey of literature in the field we encountered no discussion about the requirements of future systems with respect to digital forensics and what type of meta information beyond MAC times and user information is desired. The discussion shows that the digital forensics community is aware of what tasks need to be

performed and also aware of the fragility of digital evidence. What is lacking is an analysis of what information is necessary to perform those tasks or make them easier to perform, and further what kind of desired information can actually be obtained from a computing system.

2.2 Information Flow Analysis

Keeping track of extra information in a computing system about events at the location where they occur is typically merely a problem of allocating storage for the information, when recording it. Recording the event or the nature of the event at its source is not a problem (e.g., a user modifying a file or a process receiving data from a specific remote location). However, events may influence other events and as processing of data potentially results in new events, the roles of the earlier events get lost because the information is no longer available to the entities observing the new events. If we want to keep track of which events influenced others, we need to examine how information flows within the system.

The area of information flow analysis is concerned with determining how information is propagated within a system. The paths of how information flows describe a causal relationship: if information flows from A to C via B, then A has caused B to communicate with C. Some research in information flow analysis is concerned in how to restrict information flow between subjects and objects, while other research tries to determine how the information actually does (or will) flow.

2.2.1 Information Flow Policies

Information flow policies describe how information is supposed to be accessed or modified on a system. This usually implies a partitioning of the resources and subjects of a system into different classes, that form a hierarchy or lattice. Information flow policies are concerned with describing how access to and propagation of data on the system is allowed. Usually, the enforcement of the policies are left to access

control mechanisms and the analysis of the actual information flow of the system is left to be performed by other techniques.

The Bell-LaPadula Model

The Bell-LaPadula Model [4] is based on military classification of data. The model has subjects, objects and security classes, the last having some sort of ordering associated with it. Subjects possess *security clearances* and objects *security classifications*.

The following two properties define secure flow of information in the model: a subject can only read an object's content if the subject's security class is at least as high as the object's, and contents of objects may be written only to objects of at least as high a security class. The first property, called the *Simple Security Condition*, makes sure that subjects cannot directly access objects for which they are not cleared. The second property, called *Star Property*, further makes sure that a subject with clearance to access an object, that is classified at a particular class, does not generate a new object with the same contents as the original object but with a lower classification.

The Bell-LaPadula Model ensures that no unauthorized access of the objects of a system takes place. It thus protects the confidentiality of data on a system.

The lattice model introduced by Denning [27] is an extension of the Bell-LaPadula model. Here, security classes together with a class combining operator $\oplus$, a greatest lower bound operator $\otimes$ and a flow relation between classes form a lattice, which is used for access control between processes and objects.

The Biba Model

The information flow model defined by Biba [7] focuses on data integrity rather than confidentiality. The model consists of subjects, objects, and ordered *integrity levels*.

The following rules maintain the integrity (trustworthiness) of objects: a subject can read an object only if the subject possesses the same or a lower integrity level as the object; a subject can write to an object only if it possesses the same or a higher integrity level as the object; a subject may execute another subject (program) only if it possesses the same or a higher integrity level as the subject (program) being executed. The first rule makes sure a subject can only access information that is at least as trustworthy as the subject itself. The second rule ensures that the subject can only modify objects that are not more trustworthy than the subject itself. The third rule ensures that a subject may not make other subjects, that possess a greater integrity level, act on its behalf.

The Chinese Wall Model

The Chinese Wall Model [8] addresses both confidentiality and integrity on a system. While the model was developed to guide information flow within a business environment, it may also be applied to other systems. In the model resources on a system a grouped into both *company datasets*, (CDs) and *conflict of interest* classes (COIs). A CD contains all the data in the system that belongs together, e.g. objects belonging to the same company or project. A COI is a collection of datasets whose information must not be shared with other members in the class. Under the Chinese Wall Model, a subject S may access an object only if the only other objects that S had previously accessed are in the same CD as the object, or if none of the objects S had previously accessed is in the same COI as the object. A subject may also access an object if the object is sanitized. A subject S may modify an object if the subject has read access to the object, and if the object is in the same CD as all the objects S can access.

For the Chinese Wall Model to work, a set needs to be maintained for each subject, that contains all the objects that were ever accessed by the subject. There

is a concept sanitizing of an object, after which the object may be removed from the subject's access set.

Denning's Information Flow Model

Denning formally defines an information flow model $FM = (N, P, SC, \oplus, \rightarrow)$, where N is the set of objects on a system, P the set of processes, and SC a set of security classes. The $\oplus$ operation is a binary *class-combining* operation that defines the class of the result of (binary) operations between members of the security classes. E.g. if $a \in A$ and $b \in B$ then the result of any binary function on a and b belongs to security class $A \oplus B$. The $\rightarrow$ operation is a flow relation between security classes. It defines the permitted information flow. If information is allowed to flow from class A to class B, then $A \rightarrow B$.

An information flow FM is *secure* if none of the sequences of operations on a system violates the $\rightarrow$ relation. Furthermore, SC, $\rightarrow$, and $\oplus$ form a universally bounded lattice of security classes.

Summary

The models we have discussed above merely formalize the system policy in regard to how information is supposed to flow. It is up to individual systems to make sure they adhere to the model. This may be a difficult task when considering the trade-off between security and usability on a system. In particular, to make a system practical, concessions have to be made in terms of confidentiality and integrity. In a practical environment subjects need to have more rights than they are allowed under the Bell-LaPadula and Biba models. Nevertheless an observer might be interested in how actual information flow has taken place to deduce if undesired accesses or modifications have occurred. Information flow analysis addresses this problem. Here, either the possible or actual information flow of programs is analyzed to either enforce

information flow according to the model or to detect violations of the information flow policy. In the following, we will discuss a number of such approaches.

2.2.2 Static Information Flow Analysis

A substantial amount of research has been done in the area of static information flow analysis [89]. The primary focus lies in assuring data confidentiality and integrity when using certain programs on a system. One approach is to use techniques from type systems for controlling information flow. Security identifiers are attached to variables and expressions and used to verify the information flow at compile time [45, 46, 75, 93, 108]. Other approaches use semantic-based security models [52, 107], analyzing end-to-end program behavior, often related to some sort of noninterference [43, 66] policy.

Denning [28] and Denning and Denning [29] discuss compiler-based mechanisms that verify information flow against a security policy. With this, one is able to certify that given programs do not violate the security policy.

Static information flow analysis is a powerful method to determine how information is propagated by which principals. However, all the programs running on a system need to be analyzed to verify that they adhere to the information flow policy. Even though advances are made in the automation of the tedious work, analyzing programs is still a time-consuming and expensive task that increases with a program's complexity. Furthermore, one can only be certain about those programs on a system that have been analyzed, which takes away the ability to execute general-purpose programs. This limitation is acceptable in many scenarios where knowledge of the exact information dissemination in a system is crucial.

2.2.3 Dynamic Analysis

Some work exists that is intended to track runtime information flow of programs [110]. Here it is proposed to incorporate Denning's compiler-based informa-

tion flow concepts [28, 29] into the Java virtual machine to keep track of every user identifier associated with the running program and then enforcing access control at the time when system calls need to be made. A weakness of this approach is that the user identifiers are vulnerable to tampering when kept in user space. Also, no implementation or results have yet been published.

The Data Mark Machine developed by Fenton [37] associates a security class label with every variable on the system and is able to analyze information flow at execution time. However, it is a highly abstracted machine not suitable to monitor information flow on a more complex system with the types of channels we will describe in this document.

Other work utilizes virtual machines that record certain checkpoints of the system state they are emulating, thus allowing a post-event analysis of how information has disseminated through the system. This work is either motivated by intrusion analysis [34, 42, 57] or operating system and program debugging [58, 111, 112].

The information flow analysis techniques we discuss here find their application in specialized environments. To have a complete understanding of how information flows, all the programs running on the system need to be analyzed. The alternative to this is to limit the execution of programs to an environment where their actions can be monitored. Thus far, research in dynamic information flow analysis has utilized program interpreters of virtual machines for that purpose, which comes at a high performance cost. In this document we will present an alternative that allows us to track information across a system dynamically without the performance penalty incurred by virtual machines or interpreted programs.

2.3 Network Traceback

A special case of tracking information through computing systems is network traceback. The research area of network traceback is concerned with locating the true location (in a network sense) of an attacker. This may be hosts that perform

denial-of-service attacks, or it may be a host from where a user is logged in directly and performs malicious actions, either within that host or to other hosts through the network. Within network traceback, there are currently two major areas with different objectives and approaches for solutions. One area tries to determine the source of individual datagrams that take their paths through the network. The other area attempts to correlate streams of network traffic, which are observed at different locations within the network. The following gives an overview of past research performed in both areas.

2.3.1 Packet Source Determination

The Internet Protocol Suite (TCP/IPv4) [81], which is used throughout the Internet and various intranets does not provide a mechanism to authenticate datagrams. The only field within the IP datagram header that gives an indication about the origin of a datagram is the *source* address field, which is a 32-bit long Internet address. However, routing of IP datagrams is typically only performed based on the *destination* field within the datagram, and the source field is rarely inspected by a router. Both IPv6 [26] and IPSEC [55] provide source authentication, but are not widely used in TCP/IP networking.

In normal operation, a host receiving packets can determine their source by direct examination of the source address field in the IP packet header. Unfortunately, this address is easy to falsify, making it simple for attackers to send packets that have their source effectively hidden. This is more common for one-way communication, such as the UDP and ICMP packets used in denial-of-service attacks, but has been used in attacks using TCP streams in which the TCP sequence numbers are guessable [6, 72]. There has been significant research in how to locate the source of such packets, primarily motivated by distributed denial-of-service (DDoS) attacks in early February of 2000.

Generally, DoS attacks work by consuming a limited amount of a certain resource at the victim. This could be bandwidth, CPU time, or memory. The objective is to consume as much of that particular resource so that normal operation is no longer possible. A DDoS attack usually focuses on consuming network bandwidth and uses multiple clients distributed over the network to perform DoS attacks. The software and tools to perform DDoS attacks are widely available and easy to use. The attack involves a series of master and client programs running on compromised hosts throughout the network. A client program is used to generate as much network traffic as possible and send it to the host. The master program is used to coordinate the clients so that all of them start and end their attacks roughly at the same time. Once the masters and clients have been set up, a master then can be used to direct the client to send large amounts of network traffic to a single host on the network, resulting in that single host to be overburdened with the amount of traffic it receives. Sometimes, there are multiple master machines, which, in return, are controlled by a controlling host. This hierarchical approach allows for an easy configuration and synchronized execution of an attack from a multitude of compromised hosts from a single machine. Figure 2.1 shows a simple DDoS model.

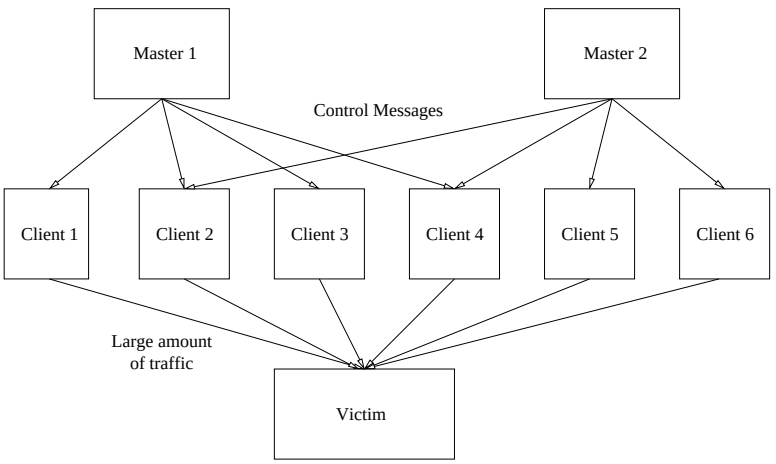

Figure 2.1. Distributed denial of service model

IP source address spoofing is not necessarily a requirement for a DoS or DDoS attack, but usually spoofing is used to hide real addresses of the hosts used for the attack. This could be because the attacker is a user on that host, or because the attacker does not want to risk losing a compromised host that he could use for later attacks.

While it is generally recommended that routers be configured to perform ingress or egress routing [38], it is clear from continuing denial-of-service attacks [71] that this is not widely done. There have been other methods proposed to perform filtering to limit the effect of such attacks [50, 78, 116].

Router Approaches

Some methods add or collect information at routers to allow traceback of DoS traffic. Rowe et al. developed the Intrusion Detection and Isolation Protocol (IDIP) [88], where an IDS that recognizes an attack queries IDIP enabled devices (including the router) about the source of the attack. The queries then cascade up until the origin of the attack is located.

CenterTrack [100] proposes the use of special tracking routers within an autonomous system that receive "interesting" datagrams directly from the edge routers. At the tracking router, those datagrams may then be examined and handled accordingly.

Cisco provides a public guideline as to how to perform tracing of DoS traffic by analyzing router traces at each hop [21]. They propose compiling access list entries that match the attack traffic to produce log output for manual inspection. Naturally, this will only work while the attack is in progress.

Snoeren et al. have developed a Source Path Isolation Engine (SPIE) [94], which enables traceback of individual packets. This is accomplished by a router that computes and stores digests of the packets it forwards. Starting from the target of a

network attack, one can then create an attack graph by querying the data collected at each router.

An approach that proposes the deployment of *hardened networks* across the Internet was introduced by Zhang and Dasgupta [119]. Here, the border routers and subsequent routers within an autonomous system (AS) mark the traffic they receive by digitally signing them. Thus a DoS victim within the AS may reconstruct the path the traffic took within the system and it is possible to filter the traffic at the border router. Also, a cooperation between different ASs can lead to even further tracking of the immediate source of the attack and contain it at that point.

Packet Marking

Other methods to determine the source of a DDoS attack add markings to the packets to probabilistically allow determination of the source given sufficient packets.

Savage et al. propose a marking scheme for IP datagrams [91]. The authors propose two main marking schemes, *node sampling* and *edge sampling*. Here, IP datagrams are probabilistically marked with a single router's IP address. A router further down the path toward the destination may overwrite an existing marking and thus the path a datagram takes has to be derived from the distributions of markings from the incoming datagrams.

To solve the problem of where to store the marking information, the authors developed a reduction scheme using XORs and hashes. Instead of sending the unencoded edge information, unique fragments of a larger identifying number are used to mark the datagram. This comes at the cost of a slight deprecation of speed of convergence and robustness, but allows storage in the space available. This technique was further refined by Song et al., proposing an advanced marking scheme [95]. Their work mainly focuses on improving the marking scheme so that fewer packets are needed and overall computation time for the reconstruction path is reduced. Instead of the router's IP addresses, only hashes of those addresses are encoded. To address

the problem of false positives in Savage et al.'s scheme when there is more than a single DoS attacker, different sets of hash functions are used at each router to keep the probability low that two routers can compute the same hash value for the edge information. Furthermore, an authentication scheme for the markings was added.

Dean et al. propose a different approach to encode the packet markings [25]. They encode paths between nodes by calculating the result of a polynomial with the IP addresses of the routers and a multiplier that is passed along in the packet as inputs. There is a trade-off between the number of packets needed to solve the polynomial for the routers' IP addresses and the amount of storage space needed for the marking for each packet.

Doeppner et al. use IP's *record route* option for the purpose of "router stamping" [32]. Here, routers record their IP address with a certain probability into a random slot available for the record route option in the IP header. With enough attack packets available, a path to the source may be reconstructed. Adler describes the tradeoff for general packet marking schemes between the size of the markings and the number of attack packets needed to perform traceback [2].

Park and Lee show that there is a possibility that an attacker may inject false markings into the network traffic, potentially leading an investigator to a false source of traffic [77]. They show that there is a trade-off between the ability to trace back the attack and the severity of the attack.

Control Messages

Several Internet Drafts [5, 114] address the use of ICMP messages for traceback purposes. A new type of ICMP message is proposed, called a *Traceback message*, that is sent by an Internet router. The message itself contains the previous and next hop of the datagram from the router's perspective as well as a timestamp and part of the datagram that caused the ICMP message to be sent.

ICMP Traceback messages are caused by a datagram forwarded by the router with a certain probability. The authors suggest a probability of about 1/20,000. The message datagram is sent to the destination specified by the datagram that caused the message to be sent with a time to live (TTL) of 255. With enough traffic going to a particular destination, this scheme allows the destination host to reconstruct the path traveled by the datagram using the TTL field as a distance measure. The link information about the previous and next hops make it possible to reconstruct a complete traceback chain to the source of large traffic volumes. Furthermore, the drafts explore several authentication options, to prevent an attacker from sending out bogus Traceback messages.

Much of the work presented above allows an investigator to locate the immediate source of a DDoS attack. In many cases, the only benefit of this discovery lies in the ability to contain the attack. Important questions, such as from where the DDoS zombie was controlled and by whom remain unanswered. Even when fully investigating the host that is running the DDoS zombie it is unlikely that an answer can be found, as we discuss in Chapter 3

2.3.2 Correlating Streams

The other area of current network traceback research tries to correlate streams of TCP connections observed at different points in the network architecture. The motivation behind this work is that attackers often log in through a chain of compromised hosts to launch an attack from the end of that connection chain. This chain of hosts will hide the actual location of the attacker, as the victim sees only the last hop in the chain. Law enforcement agencies investigating a break-in might be forced to cancel their efforts when the trace points to a host out of the jurisdiction of that agency.

The chain of TCP connections often consists of a series of interactive sessions on the hosts utilizing remote login protocols such as `telnet` `ssh`, or `rlogin`. From

each remotely accessed host, the chain is extended by logging in to the next one. Figure 2.2 shows an example of a connection chain.

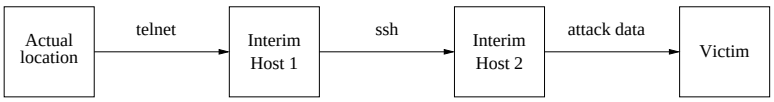

Figure 2.2. A sample chain of remote connections

While early attempts by Jung et al. [53] have proven difficult to implement and not practical (for a discussion on this, see Buchholz et al. [10]), recent research focuses on what can be be deduced from information obtained from various fixed checkpoints in the network.

Up to now, research addressing determination of the source of a connection chain has mainly focused on correlating streams of TCP connections observed at different points in the network. The initial work in matching streams constructed *thumbprints* of each stream based on content [96]. While this technique could effectively match streams, it would be ineffective in compressed or encrypted streams that are commonly used. Other work compared the rate of sequence number increase in TCP streams as a matching mechanism, which can work as long as the data is not compressed at different hops and does not see excessive network delay [117]. Another technique, which relies solely on the timing of packets in a stream, is effective against encrypted or compressed streams of interactive user data [120]. This work was originally intended for intrusion detection purposes but was also proposed as an effective method for finding the source of connection chains. While performing stream matching might be effective in some cases, such methods rely on examining network information, and might be vulnerable to the same methods that can be used to defeat network intrusion detection systems [82]. More recent work has examined the effectiveness of attackers attempting to defeat stream matching by adding delay or

additional packets to the data stream, but did not propose a method of directly matching streams [33].

Daniels describes a general reference model for origin concealment of network data elements regardless of whether the motivation lies in packet source authentication or stream correlation [24]. He defines *internal* and *external* monitors with respect to a node (a host in the network) and describes algorithms for a passive origin identification based on what the monitors observe. It is assumed that the internal monitor possesses the capability to observe the relation between inputs and outputs without specifying how this might be accomplished. The work we present in this document will aid in addressing this problem.

Any attempt to correlate data streams entering and leaving a host strictly from a network perspective can potentially be foiled by measures on the host to obfuscate any relations between such streams. Through a sequence of arbitrary transformations regarding the properties of the data streams, a stream of incoming data can be modified in such a manner that when data leaves the host, it is impossible in the general case to say if there was a causal relationship between the two. Such transformations include modification to the data itself, but also alterations in size and timing information. Figure 2.3 illustrates those transformations.

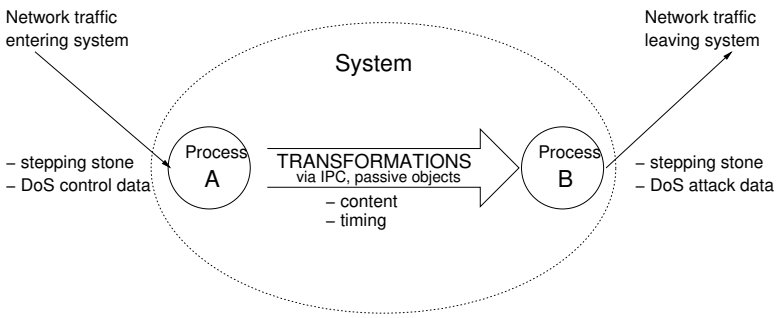

Figure 2.3. Incoming and outgoing network traffic can no longer be related because data is transformed.

3 ADDING AUDIT INFORMATION

In this chapter we will discuss what kind of information is desired when performing forensic investigations and event reconstruction [13]. In some of the cases we will merely give suggestions as how to collect and store the information as it is readily available but usually not recorded. We will also address the need for two types of information currently not available with any operating system – user influence and location information – and thus motivate the work described in the remainder of this dissertation.

3.1 Types of Information

A recent gap analysis study for computer forensics tools by the Institute for Security Technology Studies [49] shows that for Category 1 of the National Needs Assessment – Preliminary Investigation and Data Collection – the section for operating systems is well-covered with existing tools that address the needs [48]. This could lead to the conclusion that data collection for operating systems is a solved problem, but this is not the case. The reason for this is that all the gap analysis as well as other studies [48, 74, 86] are only addressing the retrieval of information that is currently recorded explicitly for those operating systems. Apparently, the question: "What information would you like to have available as a forensic investigator?" has not been sufficiently addressed in the forensic research community. Part of the problem has been addressed by Kuperman [60], but the actual quality of the information retrieved on current systems and its usefulness for forensic purposes has not been addressed, nor has the feasibility of obtaining and storing the desired information been examined.

We can categorize desired information into the following groups:

1. Information that is available to the system and recorded on non-volatile media.
2. Information that is available to the system but is not recorded.
3. Information that is not currently available to the system but could be made available.
4. Information that is impossible to be obtained by a computing system.

A forensic investigator will normally only have access to the first kind of information, or, if a live system analysis is performed, to the second kind. Most of the literature and development in the field is only concerned with the information that is actually present in the first category. We hold the opinion that it is the duty of future research to explore in what manner more forensically relevant information can be provided to an examiner in a feasible fashion. Considering the design of future systems, it is useful to first evaluate what the desired information is, whether and how it can be obtained by a computing system, and if it can be stored in a reasonable fashion. This way, some or all information from the second and third categories could be moved to the first one, plus we will gain an understanding of what is possible.

The exact kind of information and the scope of its storage we are discussing in this dissertation will differ from system to system. One can hardly require all operating system vendors to start modifying their products to record more data for forensics, or force users to enable such logging. However, there are many cases where extra information is desirable, be it to be able to show due diligence, or to more quickly discover if a system was compromised or accessed in an unauthorized fashion.

In the following, we shall examine which extra information is desirable from a forensics point of view, analyze how the currently existing information satisfies those wishes and how feasible it is to obtain and store information that is currently not collected on a system.

3.1.1 The Relevance of Metadata for Forensics

If it were possible to record a system's state – register values, memory, timers, network events, interrupt information, etc. – for every single clock step, one could use that information to deterministically replay all events that took place on the system. The answers to most questions an investigator might have could be answered, albeit in a tedious and time-consuming fashion. Even if it were possible to record all that information it still would not be feasible as the amount of time necessary to record the information on a storage device slows the system down several orders of magnitude [35]. As this approach is not feasible, we have to utilize snapshots of the system's state instead. A snapshot reflects the system state at a given discrete point in time. In addition to knowing the actual state of the system for those points in time, one might be able to draw conclusions about the state changes that occurred between two given snapshots. Such an approach using external logging of processes, files, filenames, and system calls, was implemented by King and Chen [57]. That approach was further refined to allow a virtual machine layer between the operating system and the user space to record select snapshots of the system state that can be used to replay events on the system [58]. That approach was designed for debugging operating systems as the penalty for utilizing a virtual machine and the space overhead might not be suitable for production type systems.

Taking a snapshot of the entire system's state or large parts thereof on a frequent basis might be feasible for critical systems or debugging. In the general case, limited storage capacity and performance considerations prohibit this practice. For this reason we need to consider a further reduction of information quantity and frequency of recording, preferably through an already existing mechanism on the system. The hope is that through an audit trail of individual changes to parts of the system (small deltas in the system state) we obtain sufficient information to understand the changes in the system's state leading up to the current one.

Files play an important role in the operation of most computing systems. Usually the operating system itself as well as the boot mechanism are comprised of files. Program executables, configuration data and startup scripts, user information, as well as application data are stored in files. Therefore looking at files gives a rough view of information flow, file accesses can show what programs were executed when, and file modifications show what was altered on a system. The operations on the system's files are only a subset of the system state. However, for the reasons discussed above, they can yield answers to many questions of interest to a forensic examiner. Furthermore, recording meta information about a file's operations as they occur is a mechanism that introduces little computational overhead. Thus a file's metadata seems a logical place to record our subset of the system's state. The metadata associated with a file can be seen as the characteristics of a digital object as discussed by Carrier and Spafford [18].

Information recorded by system or user programs may aid the forensics investigator during the analysis. System logging facilities such as *syslog* or shell history files and third party programs such as Tripwire [56] or tcpwrappers [106] provide valuable data for a forensic investigation. However, programs running outside of the system's kernel space may not have access to the necessary information. Plus, all the information is stored in files, which are subject to deletion or tampering. Other approaches that modify the kernel [11, 17] either do not store the added information on a permanent basis, or do so in log files, as well. Other approaches, such as Sun's Basic Security Module for Solaris [101] store extensive audit information in sequential log files using an *audit token* to relate records to each other. The audit records can become quite complicated and large in size, and space management can become complex [39]. Furthermore, the information is not stored directly at the object of interest (i.e. the file) but rather operations on files have to be reconstructed from all of the audit records on the system. By storing the desired information directly as file system metadata we gain the following benefits:

- The information is automatically collected and stored by the system: all the information that is available to the system is available to be recorded.
- The information is automatically collected without the penalty for setting up external logging mechanisms.
- The information is directly stored with the object of interest. It is not necessary to correlate various system logs to obtain the desired information.
- Tampering with the information is not as simple as tampering with a file. If raw disk access is not allowed by the operating system, the recorded information is protected from all users. Even if raw disk access is allowed a malicious user still has to navigate the file system to get to the information. When modifying or deleting it he needs to be careful not to destroy any data that is crucial to the successful operation of the system.[1]

3.2 Desired Information

As can be seen from the overview given in Chapter 2, current operating systems and file systems were generally not designed with digital forensics in mind. On most Unix-like systems, the metadata associated with a file that holds forensically usable information is only 22 bytes, of which 16 are timestamps. In this section we will discuss the types of information that is desired from a forensics point of view.

When performing a forensic investigation on a computing system an investigator needs to reconstruct as many events and actions that took place on the system as are necessary to draw unambiguous conclusions. This may be as basic as locating contraband material on a system and determining when and how it got there, but even that is not a simple task. In its most complex form such an investigation will attempt to reconstruct all events that took place on a system. This could be to

[1] This may easily occur due to race conditions on an active system.

investigate a break-in or crimes committed by an insider. The main questions a forensic investigator has to ask are: *who, what, when, how, where* and *why*.

The *who* question is concerned with what user is (or which users are) responsible for certain actions on the system. *What* addresses what actions actually were performed on the system, *when* over which time interval they took place, and *how* in what manner those actions were executed. The *where* question is to determine both where the responsible users were located when they initiated the actions as well as where the data on the system, i. e. files, came from. Finally, the *why* question is concerned with the motives that lie behind the actions. As a computing system cannot know the intentions of its users the answer to this question is one that the investigator must infer from the answers to all the other ones.

3.2.1 Who Did It?

The question of who is responsible for certain actions or the existence of data can be important in the course of an investigation. This holds especially true for systems with a large number of active users, such as a server in a thin-client environment or systems that offer network portal services. Most systems utilize some sort of authentication mechanism – usually a login procedure requiring a user name and a password – to bind a user identifier and often a group identifier to a process or a session.

The user and group identifiers for processes and files are also commonly referred to as the "owners" of those instances. It is thus tempting to classify everything that bears such an identifier as the result of that particular user's action. However, the original purpose of those identifiers in operating systems lies in access control, not true ownership. The only thing that may be deduced by looking at the identifiers is that a process bearing a particular identifier has been granted the permissions to an object that is associated with that identifier. Thus, in most of the computing systems known to this date it is difficult or even impossible to determine the user id

of the subject that is truly responsible for actions. The information might be gained by correlating other information, such as login times or typical user activities, with the times at which the file operations occurred, but this process is tedious and may not lead to the correct conclusions as we discuss below. If the correct information is stored directly with the file, no correlation work will be necessary.

From a digital forensics perspective the question of who "owns" a file is irrelevant. We want to know who created, modified, accessed, and deleted it. So who performed those operations on a file? In many cases the user id of the file will be equivalent with the identity of the user responsible. There are exceptions, though. For example, a file may be created by User A who then changes the user id of the file to User B. In Unix, this may be done with the `chown` command. Such commands are used to transfer permission rights for an object from one user to the next, but once executed any notion of the creator of a file is lost to the system. For the remaining operations (modify, access, delete) it would make sense to look for the responsible users within the set of users that hold the proper permissions for that file. This set may be quite large (up to any user on the system), and also when the permissions or user and group identifiers change, the information deduced would be incorrect.

Simply introducing new fields that associate user and group identifiers with the various timestamps (MAC times) can solve the problems addressed above. However, this will not be enough to solve other ones where a user's actions affect a file. For example consider two processes, one controlled by User A and the other controlled by User B as illustrated in Figure 3.1. Process A reads data from File 1. The two processes communicate using interprocess communication (IPC) and User A supplies the data it read from the file to B's process, which writes the data into File 2. The creator of the file clearly is User B. However, User A also played an important role in its creation and content. Current systems do not have the ability to detect User A's effect on the file.

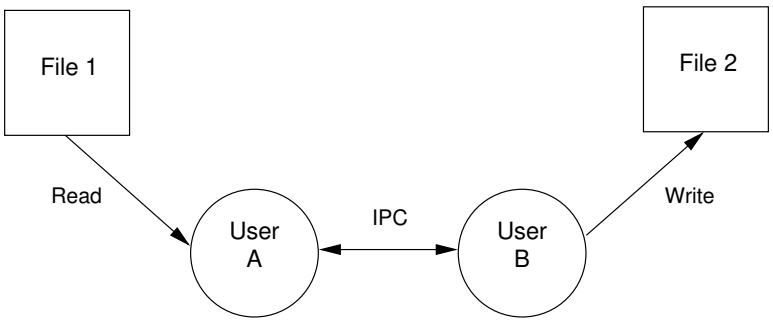

Figure 3.1. The contents of File 2 are influenced by User A

If we extend the example to more processes and users that play a role in the creation of the file we can observe that a fixed size field to hold user information for a file is not sufficient (unless it is large enough to hold information about all users). Also, if multiple users' processes communicate with each other, how can the system tell which ones played a role in a file operation and which ones did not? It turns out that this problem is actually undecidable for the general case. As the system is generally unaware of the information flow within a process, the only thing that can be done is observe its inputs (interprocess communication in this case) and outputs (the creation of the file). Deciding which input caused an output to occur is equivalent to solving the Halting problem. The Halting problem [102, 103] states that, given a Turing machine M and its input x, it is undecidable in the general case to determine whether M halts on x.

While this problem cannot be solved, there are two ways to manage it. The first is to only allow programs to run on the system whose information flow has been determined through methods such as static analysis. In this case it is known what data from which input affects the output and the system has this information readily available. But while we obtain the correct desired information, we lose the ability to run any general program because of the requirement to perform information flow analysis for each program we allow to run. In addition to the fact that information flow analysis can be time-consuming, for some programs such an analysis might not

be possible because of randomness or race conditions. The second way to avoid the problem is using an approximation. Because we cannot tell which exact inputs affect the output, we can simply assume that all of them had an effect. This approach will result in some incorrect extra information being kept, but it also assures that no correct information is discarded. From a digital forensics perspective, this is a good approach for two reasons: if information about a particular user is not associated with a file, we can be sure that that user did not have anything to do with the file's operation; also, information that is present and might be false is better than no information being present at all, especially if the amount of false information is kept small. We will focus on obtaining this kind of information in the second part of this dissertation.

By the discussion above we can see that the "who" information actually falls into Category 4 of our classification: it is impossible in the general case to obtain the correct information. We do believe, however, that the approximations mentioned in this section are good enough in most cases and thus justify the recording of this information.

3.2.2 Where Did That Come From?

There are many cases where it is desirable to know from where a particular file on a system originated. The problem of defining *location* is not trivial, however. Apart from the TCP/IP based network location information we used as origin information in previous work [11, 12], location information may take on many different forms. The GPS location information proposed by Denning and MacDoran [30] may be used as a location identifier. Location information may also be as simple as "local vs. remote", where "local" means information coming from a source or device physically connected to a system, such as a keyboard, mouse, or scanner. Information entering the system from a network through system services, such as `ssh`, `telnet`, `ftp`, `http`, or `rpc` would then be classified as "remote". Another example of location information

could be the structure of an organization. The organization may be split into different offices, the offices into departments and departments into smaller divisions. This forms a location hierarchy that also may be used to form location identifiers and groups. An Internet draft proposed by Leach and Salz [63] discusses Globally Unique Identifiers (GUIDs). A GUID has a fixed size of 128 bits and contains time and node (network location) information. The generation algorithms specified in the draft ensure that collisions among GUIDs occur only with a low probability.

Location information on a system regarding from where processes or files originated or were influenced is desirable because it can be used for system analysis (forensics, intrusion detection), access control (enabling policies based on location information), and the network traceback research discussed in Section 2.3

When locating contraband material, location information may lead further down the chain of distribution, which may result in follow-up investigations. When unknown files are found on a system, information about where the file came from may give clues as to what kind of file it could be, plus – if the file is malicious code – the information could be used trying to locate the author. Furthermore, some files may immediately be classified as benign on a system if their origin is from a trustworthy source. For example, this might be the operating system vendor's installation media for system binaries. With current systems, no such origin information is available. In some cases the origin of files may be deduced by correlating log information – such as mount times or web logs – with file timestamps. This information will only be available for origins that are logged on the system, plus some file systems do not have reliable creation timestamps and the information may be lost.

Where do files on a system come from? A user can create one by typing on the console, they can be read from storage media such as CD-ROMs or floppy disks, they can be downloaded from a network, they can be transfered from devices such as digital cameras or scanners, etc. Current computing systems have no notion of the origin of their files (or processes). The reason for this might lie in the fact that the systems from which the more recent ones descended were mostly self-contained

and isolated units with their only inputs coming from a console or card readers. However, the problem of determining a file's origin is not as simple as the above examples might suggest. By definition a computer computes data from other data. The fact that a system produces new data makes what we mean by origin of a file (or data in general) more complicated.

Consider the following example: A user had typed and saved a C-source-code file on the console. He later logs in from a remote location and compiles it with a compiler that was installed from the operating system vendor's distribution CD. He also links in a library that was downloaded from an open source web site (see Figure 3.2). Taking into account the origins of the files and processes that played a role in the creation of the resulting executable, what should its origin be? Ideally we want to capture the origin of all data involved in the generation of the new data.

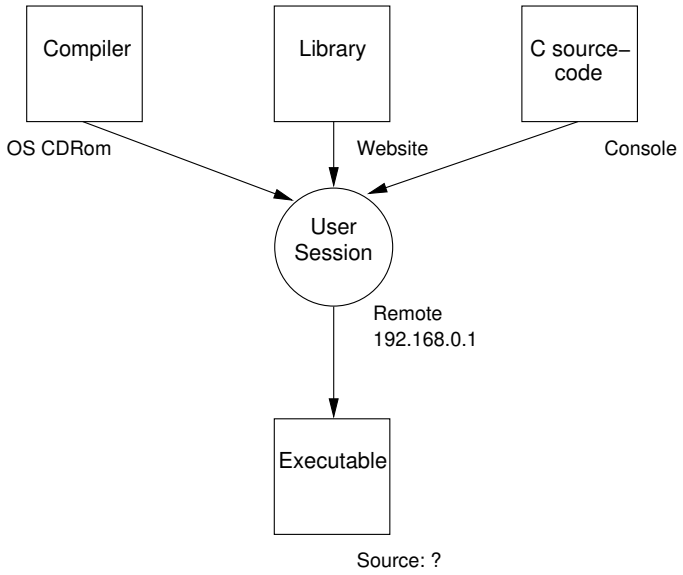

Figure 3.2. Given the origin of the involved entities, what is the origin of the new file?

A specialized compiler that can take origin into account and knows which sources are used in the creation of a file could correctly keep track of all origin information.

For general unknown programs that produce files, the same problem as with user influence applies: It is an undecidable problem. However, the same approximate solutions that can be used for the user influence problem can also be applied here.

As with the "who" information, the answer to the "where" question also falls into Category 4 of our classification. Maintaining origin information is similar to the problem of keeping track of who influenced whose actions on a system and the work in the second half of this document will also apply to origin information. Given that there is no location or origin information kept at all in existing systems, we believe that adding support for keeping and recording it will add substantial support for forensic investigations, especially in those cases where tracing back to an author of code or creator of contraband material is of importance.

3.2.3 When Did What Happen?

All commonly used file systems associate certain timestamps with their files and directories. The original purpose of timestamps lie in archiving and backup of files, but many tools serving different purposes, such as *find* [104] or *make* [105], utilize them.

The timestamps available for the ext2 file system [16] are modification, access, change, and deletion times. The first three are commonly referred to as *MAC times*. NTFS has an altered (A), read (R), MFT changed (M), and creation time (C). As can be seen, different file systems have different kinds of timestamps available. Furthermore, confusion may arise from the different naming of the metadata fields. What is MAC in ext2 would be ARM in NTFS when using the Windows terminology. There is no deletion time in NTFS and there is no explicit creation time in ext2. This latter fact is a common misunderstanding about the file system that emerged from UFS: the "C" in MAC time is often thought of as the creation time. Even the Linux source code refers to the *ctime* field in ext2 as "creation time." This is incorrect, as the original term is *change time*. This timestamp field is updated every

time there is a change in the inode information (a change in the metadata itself). When a file is created, the ctime is set to the time of creation because in a sense the metadata has been changed. If the file's metadata changes after that the ctime is updated again. A simple change in permissions via *chmod*, or user or group via *chown* or *chgrp* is sufficient for updating the ctime field. In most cases the ctime will be equivalent to the creation time, but there is no guarantee for that and a forensics investigator needs to be aware of that fact [41].

For the Unix and Unix-like operating systems the POSIX standard [47] defines the timestamps, their meaning and when they are updated. The required timestamp fields under POSIX are modification, access, and change times. An overview of which Unix system calls change what timestamps can be found throughout the literature (e.g., see Stevens [97]). The specific time when an update has to occur is ambiguous, however. The standard specifies that the timestamp fields need to be marked for an update and that "[a]n implementation may update fields that are marked for update immediately, or it may update such fields periodically."

While some of the names for timestamp fields are unambiguous, others have room for interpretation. A creation and a deletion time are straightforward in their meaning, but what exactly is meant by access, read, modification, and alteration time? Simply because a metadata field is named a certain way does not mean that it really conveys the information suggested by that name. When is a file considered accessed or read?

As the semantics of the timestamps are not specified by any standard they are open to interpretation and different operating systems' implementations can show different behavior as to when the timestamps are updated. This may be sufficient for using timestamps with tools such as *make* and *find* or for backup purposes. From a forensics point of view a clear definition of what the timestamps mean is of much greater importance. Furthermore, different operating systems or file systems may keep different kinds of timestamps. As we have shown above, Unix-like operating

systems usually do not keep a real creation time, whereas Windows using NTFS does.

The above examples of using *chmod* or *chown* to update a file's ctime show a side-effect of how timestamps may be tampered with using access control operations that affect the file's metadata. There are other commands, such as *touch* in the Unix world that allow a user to arbitrarily modify the timestamps. To obtain audit data of good quality in the sense we defined earlier, creation, modification, and access timestamps should not be subject to alteration by any user. If modifiable timestamps are needed for certain tools or procedures such as backups, then they should be separate from the timestamps we use for forensics.

All current file systems have in common that only the latest respective time is kept. This is understandable for reasons of space constraints, but not satisfying from a forensics perspective. Modern journaling file systems, such as etx3 and the Reiser file system [83] record more timestamps as part of the journal. However, one cannot choose which timestamps are kept – only the metadata of the latest operations are preserved. Ideally, all operations on a file should be recorded. A creation and a deletion time will only require one field, as the operation is only performed once per file. Operations that modify or access a file occur quite frequently, however, and there is no upper limit in the amount of space required to record these occurrences.

The GUIDs mentioned in the previous section also contain time information. Depending on what kind of location information one is interested in the GUID could replace time and location metadata fields. In this case the GUID is not bound to the file itself but rather to the individual actions to the file. However, as the above discussion shows, files may have many individual sources and events from the same sources may occur at different times. In these cases a coupling of location and time may not be desired. If the system does support GUIDs, though, the file metadata would be a good place to store it and use it for propagation purposes.

The "when" information falls into Categories 1 and 2 of our classification: some of it is recorded already, whereas the rest is available to the system but not recorded.

For a forensic investigator it would be beneficial not having to worry about what timestamp means what on a particular system and which timestamps are available. For this, standardized semantics of timestamps as well as a standard set of timestamps available are needed. The necessary information is present on all systems. While recording all of it might be infeasible, currently it is not possible to record the information at all, even if desired. A more fine-grained set of recording options as discussed below may help to adjust any particular system's requirements.

3.2.4 How Did It Happen?

The manner how an operation was performed on a file can be of importance to an investigator. By "how" we mean what controlling agent or executable program was used to perform the operations. It should make a difference for the course of an investigation whether a program named *explorer.exe* was used in the creation of contraband material on a system, or if the program was *backdoor.exe* instead. These programs can be seen as agents of the process's user to perform tasks on the system. The role of user agent may be further delegated to other programs, creating a chain of agents. For example, *command.exe* may invoke *explorer.exe*, which in turn may invoke *winamp.exe* to access an MP3 file on the system.

Having access to the entire chain of user agents for an operation on a file obviously can be valuable information in an investigation to reconstruct events on a system. Apart from scenarios described in the above example it also enables an investigator to identify automated system service routines that manipulate files such as system cleanup, log rotations, file indexing, etc.

It is necessary to record the chain of user agents because there are cases where ambiguities occur even with complete access and modification information available for a file and all executables on the system. This may happen, when several executables are accessed by the same process prior to performing the file operation.

In addition to simply referencing the name of (or a pointer to) an executable in the chain, it may also be desirable to know what the executable's version or patch-level is. For this purpose, a cryptographic checksum could be included in the chain information so that the value can later be verified against a white-list of known executables.

This kind of "how" information is currently not available on the common computing systems, but could easily be made accessible by keeping a stack of agent information, given that there is sufficient space available to keep the information.[2] Each time a user agent invokes another one, the information is pushed onto the stack and each time it finishes the information is removed. Thus, this information falls into Category 3 of our classification.

3.2.5 What Was Done to the File?

In addition to the metadata described in the previous sections the actual nature of the modification to a file is also important. Ideally, the entire chain of modifications from a file's creation through its current state should be available.

Alternatively to storing the actual modifications of a file, only the hash values of the different versions can be kept. This way it is at least possible to identify version changes for well known files, such as kernel versions and upgrades or patches to program binaries.

The "what" information is readily available on current systems but changes in files usually are not recorded. This falls into Category 2 of our classification. In general, for this type of information the same considerations are true as for the "when" data: it is probably infeasible to record everything, but even if desired the information can currently not be recorded at all.

[2]This is similar to the space management issues we address in Chapter 4 for labels.

3.2.6 User Influence and Location Information

The types of information we discuss in the previous section are, for the most part, readily available on the system and only need to be stored appropriately. This is most certainly true for the *when* and *what* information. The information required to answer the *how* question can be gathered by keeping track of an program-calling hierarchy and recording the agent chains accordingly. It should not be difficult to amend a system to include this behavior.

As we have shown, the *who* and *where* questions are more difficult to answer. The information that we require is not available on current computing systems. For the remainder of our work we will discuss an approach that can help obtaining the desired information: user influence and location/origin information. We will not discuss any further the problem of storing all the data we discuss in this chapter as part of file system metadata, as this is well beyond the scope of this document.

4 A MODEL FOR LABEL PROPAGATION BASED ON CAUSALITY

Motivated by the lack of ability to track user influence and location information through a computing system, in this chapter we introduce a model that allows the propagation of arbitrary labels based on how information flows on the system. This is a generalization, moving from user and location information to general purpose labels as the propagation mechanism works for arbitrary information that we associate with entities on a system [14]. First we define the term *causality* with respect to information flow on a system, then we introduce the general model, address space management issues, discuss properties of our model and then present case studies.

4.1 Causality

Consider a system that is comprised of active principals and passive objects. We define a *principal* as the active agent on a system that performs actions and interacts with other principals. A principal may act as an agent of a human being, on behalf of other principals, or the system itself. Principals can create other principals; create, access, and modify passive objects; and exchange information with other principals through communication channels. We use the term *subject* to denote either a principal or an object.

Principals have inputs and outputs for the purpose of interacting with other principals and accessing passive objects. Observable changes in a principal's state are defined by its outputs. Such a change can be caused by many factors: implicit measures within the principal itself, input from the system, input from another principal, input from an object, or any combination of the above. There might be other changes of state for a principal. These may include hardware failure, electrical surges, or cosmic radiation. As these are non-deterministic, unpredictable events,

those types of change of a principal's state will be ignored by our model. In the case that such an event triggers an output, the model will incorrectly blame one of the principal's inputs for the output (or none if there had not been any observable inputs).

We define a given input to be a *cause* for an observable change in state of a principal (and thus the cause for an output), if a change in state observed in the output is different based on whether or not the input is provided. This is consistent with the principle of non-interference [43, 66].

When all the internals of a principal are known and deterministic, it is possible to analyze exactly which inputs cause what outputs as we mentioned in Section 2.2 However, generally this is not the case. If we view a principal as a black box and only observe its inputs and outputs, even in the simple case when there are only two inputs and one output, it is undecidable [102, 103] to determine which input (if any) caused the output, for the general case. This can easily be proven by reducing the following well-known undecidable problem to our case: given a Turing machine M as well as the two strings x and y, determining if $M(x) = y$ is undecidable [76]. Also, Rice's theorem [84], which states that, for any non-trivial property of partial functions, the question of whether a given algorithm computes a partial function with this property is undecidable, can be applied to this.

To avoid the undecidability issue, we utilize a pessimistic heuristic: if an output can be observed for a principal at time t, we consider all previous inputs of time $t_i \leq t$ as potentially having caused the output. Thus any information exchange between principals – direct or indirect – has a potential effect on successive outputs of a principal. This approach will yield false positives as certain inputs may not have been the cause of an output. However, this ensures that any input that did cause an output will be considered.

4.1.1 Labels

In the following, we will present a model to follow information flow between principals based on causality. For this purpose we introduce an operation that binds a *label* to a principal. By label we mean an arbitrary string of bytes whose interpretation depends on the given application of the model. Labels are bounded in size and may either be ordered or unordered. Labels are then *propagated* to principals and objects based on causality: if an input causes an output, the label of the input's source needs to be propagated to the input's target. By propagation we mean some function of the label and any existing labels of the target (the target's *label set*), resulting in a new label set for the target.

This approach differs from the information flow analysis methods we discussed in Section 2.2. It is a dynamic solution without the overhead incurred with the techniques previously discussed. We achieve this at the penalty of being imprecise, meaning we do not track the exact information flow but rather all possible ones. The dynamic approach gives us the advantage of being able to execute arbitrary programs on a system without having to analyze their information flow first. There is no need for an interpreter-based runtime environment or virtual machines, which incur a performance penalty. Also, no special hardware is needed to track information flow. Using labels and propagating them dynamically at run time means that we perform a forward propagation of meta-information. As a result, we can observe the information immediately with the principals and objects on a system, without having to perform any reconstruction steps as required in other work, where the operations of a system are recorded and later analyzed [57]. With our approach it is possible to not only determine labels sets of any given principals and objects, but also to determine if a given label is present at what entities without having to reconstruct the information flow explicitly for all entities on the system.

The ability to bind arbitrary labels to principals allows us to address a variety of scenarios where it is important to track not only how information propagates on

a system but also what kind of information. It further allows us to focus on only the information that is relevant for the scenario. Such use of labels allows us to help solve the problems described in Chapters 2 and 3 as we will demonstrate in the case studies in Section 4.5.

4.2 A General Model of Label Propagation Between Principals Based on Causality

The model described below is used to propagate labels according to information exchange between active subjects (principals) in a system. A label needs to be propagated from one principal to another if information is exchanged between the two of them. The idea is that if one principal causes the information flow of another principal, then the former's labels should be propagated to the latter.

A single bit of exchanged information may be sufficient to control further information flow for a principal. Therefore, not only transfer of data itself, but also success or failure of certain operations between principals must be considered as information exchange, as they count as inputs for the principals.

Information exchange between principals is performed through communication channels that are established between the two participants. As the communication between principals does not necessarily have to be synchronized there may be an arbitrary time delay between one participant's *write* and the other's *read* operation. In a sense, channels are abstract passive objects that act like FIFO queues from which principals read and write. Channels in this model are uni-directional, meaning that only one principal may write to the channel and the other can only read from the channel.

Information between subjects may also be exchanged indirectly through storage objects (objects). By storage objects we mean shared objects on a system that are used to store data either temporarily or on a long-term basis. As objects may be used to transfer information, labels from principals also need to be associated with storage objects that are modified or created by them, and be propagated to principals

accessing those objects. Here, the concept of information exchange is less restrictive than above, as only a modification of the object or actual data transfer from the object needs to be considered. The mere existence of an object may also be used to exchange information between principals one bit at a time: one principal can create or destroy an object while a second principal tests for the object's existence. If the two principals synchronize their operations, they can exchange information this way. This is the type of channel Lampson defines as a *storage channel* [61,62]. Therefore, a successful open operation to an object for read or write operations must also be considered information exchange. However, it is not necessary to propagate all labels associated with the object, rather, it is sufficient to propagate the label set of the object's creator at the time the object was created. This is because only the creator of the object can control the storage channel described above.

We define the following sets, which are all countable and unbounded[1]:

L: set of labels

P: set of principals

O: set of storage objects

$C \subseteq \{P \times \{P \cup O\}\} \cup \{\{P \cup O\} \times P\}$: set of ordered pairs $\langle i, j \rangle \in C$ if and only if a communication channels exists between $i \in P$ and $j \in \{P \cup O\}$ or $i \in \{P \cup O\}$ and $j \in P$.

label(): $O \cup P \rightarrow 2^L$, a function that given a principal or an object returns a subset of L that is called the *label set* of the principal or object. The function label(ϕ) will always return the empty set $\emptyset$.

clabel(): $O \rightarrow 2^L$, a function that given an object returns a subset of L that is called the *creator label set* of the object. This function denotes the label set of the object's creating principal at the time of creation.

[1]Subsequent sub-models may have bounded label sets

Channels between two principals are uni-directional and function according to the consumer-producer model as a FIFO queue. Instead of actual data items, it is sufficient for this model to require only that the label set associated with the data is contained in the channel. For this purpose, we define two operations on channels:

enqueue(c, l) adds a label set $l \in 2^L$ to the FIFO queue of channel $c \in C$.

dequeue(c) returns and removes the next label set $l \in 2^L$ from the FIFO queue for $c \in C$. If the queue is empty, the empty set is returned.

Each channel $c \in C$ possesses a capacity cap(c), which denotes the number of items the channel can hold. The *enqueue* operation for a channel will fail if its capacity has been reached. This in return will cause the operation that caused the *enqueue* operation to fail as well. For channels between a principal and an object the data transfer is simpler and can be viewed as atomic and instantaneous.

P, O, and C are unordered sets, whereas L, and the sets returned by label(), and clabel() may be ordered. The latter sets are ordered iff L is ordered. In the latter case label and clabel's ranges are no longer 2^L, but rather the set of sorted subsets of L.

We further have a mapping on label sets, $update : 2^L \times 2^L \rightarrow 2^L$. The mapping determines how label sets are updated as two processes exchange information. This function must be defined for sub-models derived from this model.

Below we define the operations on the sets described above (P, O, C, and L). An operation consists of two parts: an optional precondition and an action part. If there is a precondition associated with an operation, it must be fulfilled for the operation to succeed. Otherwise, the operation fails. Operations without a precondition will always succeed. Only a principal may perform an operation. To avoid a cumbersome notation the principal that performs the operation can either be implied from the operation itself, or, if necessary, is explicitly mentioned.

create(p_1, p_2) Principal p_1 creates principal p_2. The label set of p_1 needs to be inherited by p_2:

$$\begin{aligned} P &:= P \cup \{p_2\} \\ \text{label}(p_2) &:= \text{label}(p_1) \end{aligned}$$

create(p, o) Principal p creates object o. The object's label set as well as the creator labels set need to be inherited from p:

$$\begin{aligned} O &:= O \cup \{o\} \\ \text{clabel}(o) &:= \text{label}(o) := \text{label}(p) \end{aligned}$$

open(p_1, p_2) The channel $\langle p_1, p_2 \rangle$ is opened between principals p_1 and p_2. The channel has the direction from p_1 to p_2, meaning that p_1 can perform write operations and p_2 can perform read operations on the channel. Whether the operation of opening the channel succeeded or not can already be viewed as the exchange of 1 bit of information: success or failure. Therefore, the label sets of both principals need to be updated at this point:

$$\begin{aligned} C &:= C \cup \{\langle p_1, p_2 \rangle\} \\ \text{label}(p_1) &:= \text{label}(p_2) := \text{update}(\text{label}(p_1), \text{label}(p_2)) \end{aligned}$$

open(p, o) The channel $\langle p, o \rangle$ is opened between principal p and object o. P can write to the object. A successful open indicates that o actually exists, so the object's creator label set needs to be updated with the principal's:

$$\begin{aligned} C &:= C \cup \{\langle p, o \rangle\} \\ \text{label}(p) &:= \text{update}(\text{label}(p), \text{clabel}(o)) \end{aligned}$$

open(o, p) This is analogous to the previous operation, using channel $\langle o, p \rangle$ and p having read access instead.

write(p_1, p_2, n) Principal p_1 writes n data items to the channel $\langle p_1, p_2 \rangle$, i.e. it consists of n enqueue operations. In this case the channel $\langle p_1, p_2 \rangle$ needs to be open.

Precondition: $\langle p_1, p_2 \rangle \in C$

repeat n times:

$$\text{enqueue}(\langle p_1, p_2 \rangle, \text{label}(p_1))$$

write(p, o) Principal p writes data to object o. In this case the channel $\langle p, o \rangle$ needs to be open. Because o is receiving information, o's label set needs to be updated with p's:

Precondition: $\langle p, o \rangle \in C$

$$\text{label}(o) \;:=\; \text{update}(\text{label}(p), \text{label}(o))$$

read(p_2, p_1, n) Principal p_1 reads and removes n data items from channel $\langle p_2, p_1 \rangle$, i.e. it performs n successive dequeue operations.. In this case the channel $\langle p_2, p_1 \rangle$ needs to be open. Because p_1 is receiving information, p_1's label set needs to be updated with the ones read from the channel:

Precondition: $\langle p_2, p_1 \rangle \in C$

repeat n times:

$$\text{label}(p_1) \;:=\; \text{update}(\text{label}(p_1), \text{dequeue}(\langle p_2, p_1 \rangle))$$

read(o, p) Principal p reads data from object o. In this case the channel $\langle o, p \rangle$ needs to be open. Because p is receiving information, p's label set needs to be updated with o's:

Precondition: $\langle o, p \rangle \in C$

$$\text{label}(p) := \text{update}(\text{label}(p), \text{label}(o))$$

close(p_1, p_2) The channel $\langle p_1, p_2 \rangle$ between principals p_1 and p_2 is closed. Both principals may interpret this event, so this can be seen as a 1-bit information exchange. Both principals' label sets need to be updated with each other's label sets:

Precondition: $\langle p_1, p_2 \rangle \in C$

$$\begin{aligned} C &:= C - \langle p_1, p_2 \rangle \\ \text{label}(p_1) &:= \text{label}(p_2) := \text{update}(\text{label}(p_1), \text{label}(p_2)) \end{aligned}$$

close(p, o) The channel $\langle p, o \rangle$ between principal p and object o is closed. No information is exchanged.

Precondition: $\langle p, o \rangle \in C$

$$C := C - \langle p, o \rangle$$

close(o, p) This is analogous to the previous operation, using channel $\langle o, p \rangle$ instead.

addlabel(p, l) Label l is bound to principal p. L needs to be added to any existing labels in p's label set:

$$\text{label}(p) := \text{label}(p) \cup \{l\}$$

destroy(p) Principal p is destroyed. All open channels involving p are closed.

$$\begin{aligned} &P := P - p \\ &\{\langle x, y \rangle \in C | x = p \vee y = p\} : \text{close}(x, y) \end{aligned}$$

destroy(o) Object o is destroyed. All open channels involving o are closed.

$$\begin{aligned} &O := O - o \\ &\{\langle x, y \rangle \in C | x = o \vee y = o\} : \text{close}(x, y) \end{aligned}$$

A sequence of operations is an ordered list of operations, in the order they occur. At any given discrete time interval t_i exactly one operation is allowed and that operation is considered atomic. To extend the model, it may be necessary to define composite operations from the basic operations given above. For example, the opening of a bi-directional channel between principals p_1 and p_2 may be defined as $\{\text{open}(p_1, p_2), \text{open}(p_2, p_1)\}$. This new operation is also atomic. There is also an initial state of the system, which at a minimum consists of $P = \{p_0\}$ and $\text{label}(p_0) = \{\}$.

Example:

Let $P = \{p_1, p_2, p_3, p_4\}$, with $\text{label}(p_1) = \{A\}$ and $\text{label}(p_3) = \{B\}$, $O = \{o_1\}$, $\text{clabel}(o_1) = \text{label}(o_1) = \{C\}$, and update $= \cup$. The following shows a sequence of operations and its effect on the label sets:

$\mathbf{op_1}$: $\text{create}(p_1, o_2)$	$\text{clabel}(o_2) = \{A\}$
$\mathbf{op_2}$: $\text{open}(p_1, o_2)$	
$\mathbf{op_3}$: $\text{write}(p_1, o_2)$	$\text{clabel}(o_2) = \{A\}$
$\mathbf{op_4}$: $\text{close}(p_1, o_2)$	
$\mathbf{op_5}$: $\text{open}(o_2, p_2)$	$\text{label}(p_2) = \{A\}$
$\mathbf{op_6}$: $\text{read}(o_2, p_2)$	
$\mathbf{op_7}$: $\text{close}(o_2, p_2)$	
$\mathbf{op_8}$: $\text{destroy}(p_1, o_2)$	
$\mathbf{op_9}$: $\text{create}(p_2, p_5)$	$\text{label}(p_5) = \{A\}$
$\mathbf{op_{10}}$: $\text{open}(o_1, p_5)$	$\text{label}(p_5) = \{A, C\}$
$\mathbf{op_{11}}$: $\text{read}(o_1, p_5)$	
$\mathbf{op_{12}}$: $\text{close}(o_1, p_5)$	
$\mathbf{op_{13}}$: $\text{open}(p_3, p_5)$	$\text{label}(p_5) = \{A, B, C\}$
$\mathbf{op_{14}}$: $\text{write}(p_3, p_5, n)$	

op$_{15}$: read(p_3, p_5, n)

op$_{16}$: close(p_3, p_5)

op$_{17}$: open(p_5, p_4) label$(p_4) = \{A, B, C\}$

op$_{18}$: write(p_5, p_4, n')

op$_{19}$: read(p_5, p_4, n')

Principal p_1 creates object o_2, and writes some data to it. Principal p_2 then reads data from o_2, after which o_2 is destroyed by p_1. P_2 subsequently creates principal p_5, who then reads data from o_1. A communication channel to write data from principal p_3 to p_5 is then opened and data is being transmitted. After that, another communication channel is opened, this time between p_5 and p_4, and p_4 is receiving data from p_5. At the end of these operations, both p_4 and p_5 carry labels from p_1 (A), p_3 (B), and o_1 (C). P_2 carries the label from p_1. Figure 4.1 illustrates how information flows in this example.

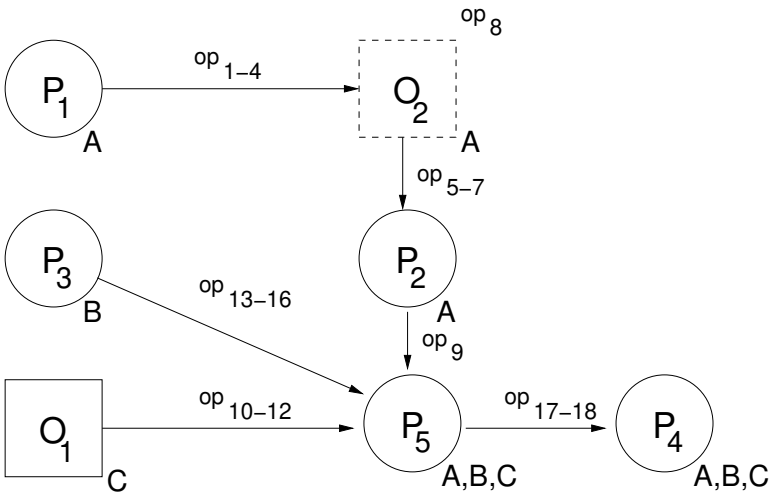

Figure 4.1. Information flow for each operation and the final label sets

On a computing system, the above example could illustrate the following scenario: The labels A, B, and C are location information and p_1 is the process of a malicious

user logged in from location A and p_2 is the process of a local user on the system who has privileged access rights. In operations op_1 through op_4 the user creates a script that when executed will create a new process and start the program stored in object o_1, which was created earlier by a user from location C. The user then dupes p_2 into executing this script (let's say the path of the superuser contains "." as its first entry and the script has a name of a common command and the superuser executed the command while in the directory that contains the script). The access of the script by the superuser is shown in operations op_5 through op_7 and the spawning of a new process with the access rights of the superuser and execution of the program in operations op_9 through op_{12}. In the meantime the user for p_1 gets rid of the script with op_8.

During operations op_{13} through op_{16}, another process, p_3, that earlier received some network traffic from location B sends some control information to the new process p_5, who in return (in op_{17} and op_{18}) opens a communication channel to p_4 (lets assume p_4 actually lies outside the computing system) to perform some sort of attack that required privileged access rights (sending bad routing information for example).

When the attack is detected and investigated the labels associated with p_5 list all the locations that played a role in the attack and can be investigated further. It could be possible that p_5 is terminated after the completion of the attack, but system logging policies such as "log all network traffic and process information from privileged ports from processes that have a non-empty label set" can record the necessary information for post-incident analysis. Note that anything that p_2 does after being duped into executing the script is now "tainted" with a label A from p_1 and and the system can be considered compromised after operations op_5 through op_7. Any child processes and objects created as a result of that need to be investigated as well.

4.2.1 Covert Channels

Principals may exchange information via the use of covert channels. Covert channels have been largely discussed in previous research. They can be divided into *storage* and *timing* channels as discussed by Lampson [62] and Kemmerer [54]. For communication taking place between a sender and a receiver through a covert channel, both must have access to an attribute of a shared resource and the sender either the ability to cause a change in the attribute (storage) or the capability of modulating the receiver's response time for detecting a change in the attribute (timing). Covert channels can be identified through the shared resource matrix as defined by Kemmerer [54] or via source code information flow analysis techniques discussed in Section 2.2.2.

Most of the research in covert channels focuses on systems with different security levels (e.g. `low` and `high`). The goal is to detect and restrict information flow between security levels (from high to low) through the covert channel. For such purposes covert channels between subjects that also have legitimate channels available to communicate can be declared not harmful and thus be ignored. Also, Moskowitz and Kang [73] propose a mechanism called the "pump" that limits the bandwidth of possible covert channels that exist in low-to-high communication. For our model the bandwidth of a covert channel is of no interest because a single bit of exchanged information between two principals may cause further information exchanges to other subjects and thus labels need to be propagated.

The model we describe above takes into account some possible covert channels, other forms have to be excluded from our model as a system cannot reliably decide whether or not communication actually takes place. Covert channels that use the success or failure of establishing a communication channel, use lock availability, existence of files, or encoded information in the actual data exchange are handled by the model. Also, probabilistic covert channels [89] that convey information by changing the probability distribution of observable data are handled because it does

not matter if there is more information encoded in information flow we already detect. However, covert channels that use a system's *availability* of resources, such as CPU, total memory, power, or bandwidth to encode messages for information exchange cannot be detected by this model. Such covert channels lie outside of the scope of this work and will not be considered further.

4.3 Space Analysis

The general model from the previous section is not concerned with any space limitations a system might have regarding labels associated with principals and storage objects. There is sufficient space to store infinitely many labels with principals and objects, and channels have unlimited capacity. This is not a practical assumption. The need for space restrictions for derived sub-models depends on two factors: the nature of the label set L and the definition of the label-updating function *update*.

A fixed or bounded number of elements for L automatically implies a bound for the maximum space requirements for each principal and object. Let $|l|$ be the size of label l in bits. Then we can define $|L| := \sum_{l \in L} |l|$ for any label set L. If L is bounded, the maximum space needed for each principal and object is potentially $|L|$. For example, if one wants to determine if information flow was caused locally or remote, or from both possibilities (in whatever sense), then we have $L = \text{local}, \text{remote}$, update $= \cup$, and $|L| = 2$. Only two bits are needed to store the labels at each principal and object. Another example of this kind would be user ids on a computing system, whose number is usually bounded by system limitations. However, even when L is bounded, the set might be so large that allocating $|L|$ amount of space for every principal and object is infeasible. We can extend the previous example and instead of simply specifying "remote" we want to record a TCP connection consisting of foreign IP address and port. The label set L is still bounded, as it consists of the cross product of possible IP addresses and TCP ports. However, there are

$2^{32} \times 2^{16} = 2^{48}$ possible labels, which is too large an amount of space to be allocated for each and every subject.

Sometimes, even though the label set may be large, it might still be possible to allocate space for all the labels each subject may have to hold. This happens if the label-updating function *update* is not an increasing function. For example, assume we are binding color labels to our principals. If, instead of unioning the label sets, *update* computes the color that is a result of mixing the colors involved, only one single label needs to be stored with each subject. So even for 32-bit colors, resulting in 2^{32} possible labels, only 32 bits of space are needed.

If L is an ordered set, then the label-updating function may also consist of operations such as *min* and *max*, which may further reduce the space requirements for the subjects.

In the general case, however, we cannot assume that space requirements are (reasonably) bounded and we therefore need to have a mechanism to manage the available space. There are two factors to consider for space management: how to distribute the available space and what to do if no further space is available. Note that for channels there is an implicit space limitation caused by their inherent capacity. An operation will fail if the operation on the channel fails because of insufficient capacity.

For the actual space management we can combine subjects together into groups and then allocate a fixed amount of space for each group. It is possible to imagine every combination of principals and objects grouped together and this is equivalent to partitioning a set of size n, where $n = |P \cup O|$. There are B_n number of ways to partition such a set, where B_n is the n-th Bell number [87,109] and $B_n = \lceil e^{-1} \sum_{m=1}^{2n} \frac{m^n}{m!} \rceil$.

In the following we will describe how to adapt the above model to take space considerations into account. In general, space limitations only affect the preconditions of our operations, so we will not repeat all the operations but rather concentrate on the relevant parts. Let $S \in \mathbb{N}$ be the total amount of space available for labels and $\wp(P \cup O)$ be the set of all possible partitions of $P \cup O$. For a specific

partition $\text{part}_i \in \wp(P \cup O)$, where $0 \leq i < B_{|P \cup O|}$, we can assume without loss of generality that part_i contains $k > 0$ subsets $g_1, \ldots, g_k$ of $\{P \cup O\}$. We call these subsets *resource groups*. Let $S_1, \ldots, S_k \in \mathbb{N}$ be the space available for labels allotted to groups $g_1, \ldots, g_k$ and $\sum_{i=1}^{k} S_i \leq S$. Furthermore, we have a mapping function $\mathit{group} : P \cup O \rightarrow g_1, \ldots, g_k$, which, given an element from $P \cup O$ returns the resource group it belongs to. Finally, there is a function $\mathit{util} : g_1, \ldots, g_k \rightarrow \mathbb{N}$, which returns the utilized space for a given group. Technically, the updating of the *util* function needs to be performed in the action part of our operations. We therefore assume that it is updated implicitly so that we can still present only the preconditions and not have to worry about cluttering up the notation. In the following we describe the preconditions for space management for our operations:

create(p_c, s): When a new principal or a new object s is created by principal p_c, one of two things may happen with regard to the space resource groups:

1. The subject s is assigned to an existing resource group g_x. In this case g_x needs to have enough available label space to hold p_c's label set:

$$\begin{aligned} \text{util}(g_x) + |\text{label}(p_c)| &\leq S_x \\ g_x &:= g_x \cup \{s\} \end{aligned}$$

2. A new resource group g_{k+1} with available space S_{k+1} is created for the new subject s. In this case there needs to be sufficient space left to allow the creation of the resource group:

$$\begin{aligned} |\text{label}(p_c)| \leq S_{k+1} &\leq S - \sum_{i=1}^{k} S_i \\ g_{k+1} &:= \{s\} \end{aligned}$$

open(p_1, p_2): As the label sets of both principals are modified, there needs to be sufficient space in both p_1 and p_2's resource groups to hold the space difference

between the old and the new label sets. Let $g_x = \text{group}(p_1)$ and $g_y = \text{group}(p_2)$ and $x \neq y$:

$$\begin{aligned} |\text{update}(\text{label}(p_1), \text{label}(p_2))| - |\text{label}(p_1)| &\leq S_x - \text{util}(g_x) \\ |\text{update}(\text{label}(p_1), \text{label}(p_2))| - |\text{label}(p_2)| &\leq S_y - \text{util}(g_y) \end{aligned}$$

If $\text{group}(p_1) = \text{group}(p_2) = g_z$, then:

$$2|\text{update}(\text{label}(p_1), \text{label}(p_2))| - |\text{label}(p_1)| - |\text{label}(p_2)| \leq S_z - \text{util}(g_z)$$

open(p, o) and open(o, p): The object's label set is not affected by this operation. P's label set is updated with o's creator label set. Let $g_x = \text{group}(p)$:

$$|\text{update}(\text{label}(p), \text{clabel}(o))| - |\text{label}(p)| \leq S_x - \text{util}(g_x)$$

write(p_1, p_2): No label sets are modified by this operation. Only the channel $\langle p_1, p_2 \rangle$'s capacity to hold $\text{label}(p_1)$ imposes an added precondition to this operation.

write(p, o) Only o's label set is modified and the resource group $g_x = \text{group}(o)$ needs to have sufficient available space for the resulting label set:

$$|\text{update}(\text{label}(p), \text{label}(o))| - |\text{label}(o)| \leq S_x - \text{util}(g_x)$$

read(p_2, p_1, n) The label set of p_1 is updated with the first n label sets contained in channel $\langle p_2, p_1 \rangle$. The resource group $g_x = \text{group}(p_1)$ needs to have sufficient available space for the resulting label set after the n updates. Let $\text{lset}_j(\langle p_2, p_1 \rangle)$ be an auxiliary function that returns the jth label set in the channel queue without dequeuing it, and l a label set initially set to $\text{label}(p_1)$.

$$\begin{aligned} &\text{for } i = 1 \text{ to } n: \\ &\qquad l = \text{update}(l, \text{lset}_i(\langle p_2, p_1 \rangle)) \\ &|l| - |\text{label}(p_1)| \leq S_x - \text{util}(g_x) \end{aligned}$$

read(o, p) Only p's label set is modified and the resource group $g_x = \text{group}(p)$ needs to have sufficient available space for the resulting label set:

$$|f(\text{label}(p), \text{label}(o))| - |\text{label}(p)| \quad \leq \quad S_x - \text{util}(g_x)$$

close(p_1, p_2) This operation has the same space preconditions as the open(p_1, p_2) command.

close(p, o) and close(o, p) No label sets are modified.

addlabel(p, l) The resource group $g_x = \text{group}(p)$ needs to have sufficient available space to hold the resulting label set:

$$|\text{update}(\text{label}(p), l)| - |\text{label}(p)| \quad \leq \quad S_x - \text{util}(g_x)$$

destroy(p) All the preconditions for closing the channels containing p must be met.

destroy(o) No label sets are modified.

The above model for space management applies to any particular partitioning of the subject set and their resulting resource groups. Two special cases stand out as they simplify space management considerably: having a unique resource group for each individual subject (*local model*) and having one global resource group for all subjects (*global model*).

Local Models:

In this group of models we have $k = |\{P \cup O\}|$. This means that each group of $g_1 \ldots g_k$ contains exactly one principal or object. This means that new subjects may only be created as long as $\sum_{i=1}^{k} S_i \leq S$ and that the space S_j belonging to group g_j is only available to the one subject.

If storage space for labels is restricted on a per principal basis, it has the advantage that only operations involving principals with many labels are affected by

whatever means are taken to deal with a restriction violation. This may result in efficient ways to identify and contain "misbehaving" principals that try to obscure their label sets. However, the local restriction may unfairly penalize certain principals that accumulate many labels in their normal course of operation. We call a model that follows this approach a *local* model with respect to label space constraints.

Global Models:

In this group of models, we have $k = 1$, and $g_1 = \{P \cup O\}$. Furthermore it seems logical to assume that $S_1 = S$. All the available space is made available to all subjects on an equal basis.

Principals that accumulate many labels during their normal course of operation are no longer penalized more than any other principal. However, a single misbehaving or even a normal process may now exhaust the entire resource pool for label storage, affecting all other principals' operations as well. Depending on how the space restrictions are enforced this could easily lead to denial-of-service attacks or label-washing. We call a model that follows this approach a *global* model with respect to label space constraints.

4.3.1 Enforcing Space Constraints

Simply defining space constraints is not sufficient for describing a comprehensive space management model as there are different ways to enforce the constraints. If an operation violates one or more space constraints, there are two ways of dealing with it: deny the operation or make space.

Strict Models:

If an operation is denied (i.e., it will fail) when its action would result in a violation of the label space constraints, it ensures the correct binding of labels to

principals at all times. As labels are never dropped, or "unbound" from a principal it follows that a principal was never bound to other labels at earlier points in time. We call a model that follows this approach a *strict* model with respect to label space constraints.

For certain applications of label propagation using a strict model is essential for its correct behavior.

Loose Models:

To ensure that any given label set size stays below or at the size limitation, a function is needed that reduces the label set size should an operation exceed that limit. Taking this approach guarantees that operations will not fail because of label size limitations. However, because the label set size will be reduced, labels, and thus information will be lost. We call a model that follows this approach a *loose* model with respect to label space constraints.

Depending on the definition of the label-updating function *update* and the label set L, there might be an upper bound on the possible label set size, either for each principal or for the entire system. If this upper bound is reasonable in size, a strict model with that upper bound as the label space constraint can be utilized. This will result in no labels being lost and no operations will fail because of label size limitations.

The actual nature of the function that reduces the label set size can be manifold, and it is beyond the scope of this document to address all cases. Possible variants might be to delete the "oldest" labels, delete the "least important" labels, combine specific labels into more general ones, etc.

Note that this reduction of label set size need not only involve only the resource groups belonging to the subjects involved at the operation. It can be quite possible that labels may be removed from "less important" resource groups and the space sets $S_1, \ldots, S_k$ be adjusted dynamically.

Another measure to free up label space is to create label hierarchies: well-known labels that occur frequently together among many subjects of the system may be grouped together into one meta-label. After the grouping only the meta-label needs to be propagted and the mapping of labels to the meta-label kept once in the system. As a disadvantage, all labels associated with the meta-label are now propagated where in some cases only individual ones would have been.

Finally, any combination of utilizing strict and loose enforcement could be utilized. Operations can be denied in certain cases, where in others it might be sufficient to make space even at the cost of losing information. The exact enforcement strategy depends on the application for the label propagation.

4.4 Properties of the Propagation Model

For the remainder of this document, we require that the label-updating function *update* preserves the labels. An example of this is the set union operation, i.e., update $= \cup$. In general, we need to have $s = \text{update}(t, l) \Rightarrow l \in s;\ \forall t \in 2^L, l \in L$.

This allows us to address the problems from Chapters 2 and 3, because the labels are not modified or dropped as part of the updating process. Furthermore, we define a special set $P_g \subset P$ of principals, which contains the principals that may generate labels.

Such a group P_g of principals implies that each label is uniquely identifiable as having originated from a particular principal or a group of principals from P_g. That means there is a mapping $originated : L \rightarrow 2^{P_g}$ that takes a label and returns the principal that created the label or the group of principals from which the label could have originated. For example, if we have a group of principals `httpd, telnetd, ftpd, sshd`, which are responsible for communicating with principals from other systems and they generate labels identifying those systems, then any label found within the system that is an identifier of a foreign system must have been generated by one of those four principals.

We shall now define what we consider information exchange between principals. Intuitively, two principals exchange information at or after a given time when data flows from one to the other. If intermediates such as other principals or objects have been used to exchange the information, then information is exchanged between the source and the intermediaries as well as the intermediaries and the target. The successful opening of channels constitutes a 1 bit information exchange. In case of checking the existence of an object for the purpose of information exchange is only relevant for the creator of the object and the principal who tests the existence. Principals who may have written to the object in the meantime do not matter as they played no role in the existence of the object. Writing data into a channel is only relevant if that very data is also read. That means that if data is already present in the channel prior to a write operation w, that data needs to be read from the channel before a reading principal is affected by w.

Because of our conservative approach when tracking information flow, we can not be sure that information was actually exchanged between principals. However, information could have been exchanged. Therefore, we shall formally define a *potential information exchange path* $IE_n(p_1, p_2)$ between two principals $p_1, p_2 \in P$ as a sequence of $n \geq 1$ operations from our model

$$\mathrm{op}_1(p_1, s_1) \circ \mathrm{op}_2(s_1, s_2) \circ \ldots \circ \mathrm{op}_n(s_n, p_2)$$

where the $s_i \in \{P \cup O\}$ are the subjects for the operations. If the operation is a read or write operation between principals, then, of course, we have $\mathrm{op}_i(s_i, s_{i+1}, q)$. This means that, if we see an operation $\mathrm{op}(x, y)$ as having a "direction" from source x to target y, the source of each operation is the target of the next one. For the *open* operation between principals we also allow the swapping of the parameters: i.e. the sequence $... \circ \mathrm{op}_k(s_k, s_{k+1}) \circ \mathrm{open}(s_{k+2}, s_{k+1}) \circ ...$ still results in a valid potential information exchange path.

Furthermore, we consider the destroy operation as the sequence of close operations that are triggered by it. Let $subIE_n(p_1, p_2, k)$ be the subsequence of $IE_n(p_1, p_2)$ consisting of the first $k \leq n$ operations.

Some operations may only be in the path if they are matched with other operations in a manner so that information is propagated from one subject to the next. A write operation needs to be matched with sufficient read operations, an open operation of a channel between principals is only relevant for the information exchange if the opening principal was created by an operation in the path, and an open operation of a channel involving an object is only relevant if the object was created by an operation in the path. In general these operations may exist by themselves, but without their matching operations will not contribute to the information exchange and therefore need not be in the path. We will elaborate on the individual match-ups in the following:

If $\text{write}(x, y, w_i) \in \text{subIE}_n(p_1, p_2, k)$, then we further require that there exists a non-empty sequence of m read operations $\text{read}(x, y, r_j) \in \text{subIE}_n(p_1, p_2, k'), 0 < j \leq m$, where $k' > k$ and $\sum_{j=1}^{m} r_j > c_i$. The number c_i is the number of items already contained in the channel $\langle x, y \rangle$ prior to $\text{write}(x, y, w_i)$. We require this so that at least some of the data written to the channel by the write operation is actually read from the channel by the read() operations contained in the path, and at least the last read() operation occurring after the write().

Similarly, if $\text{read}(x, y, r_i) \in \text{subIE}_n(p_1, p_2, k)$, then we require that the operation $\text{write}(x, y, w_j) \in \text{subIE}_n(p_1, p_2, k')$ and furthermore that there is a sequence of read() operations $\text{read}(x, y, r_q) \in \text{subIE}_n(p_1, p_2, k'')$, where $k' < k'' < k$ and $\sum r_q + w_i > c_j$. This means that at least some data read from the channel by the read() operation has to have been written to the channel by some prior read() operations contained in the path.

In summary, if either a read or a write operation is contained in the information exchange path, the sequence of operations must contain the following, where channel $\langle x, y \rangle$ contains c items, $\sum_{i=1}^{m} r_i > c$, and $m > 0$:

$$\ldots \circ \mathbf{write}(x, y, w) \circ \ldots \circ \mathbf{read}(x, y, r_1) \circ \ldots \circ \mathbf{read}(x, y, r_2) \circ \ldots \circ \mathbf{read}(x, y, r_m) \circ \ldots$$

If $\text{open}(x, o)$ or $\text{open}(o, x) \in \text{subIE}_n(p_1, p_2, k)$ and $o \in O$, then we require that $\text{create}(y, o) \in \text{subIE}_n(p_1, p_2, k')$, where $k' < k$. Even though an open operation is

necessary for subsequent read operations the opening of a reading channel to an object is only relevant for information exchange when the object was created by an operation in the path. Otherwise, the open operation must not be contained in the path. This means that the object must have been created by an operation in the information exchange path prior to the open operation.

We say p_1 *and* p_2 *potentially exchange information* iff there exists a potential information exchange path between p_1 and p_2.

Given the definitions above, the general information flow model has the following two properties:

1. If information is exchanged between principals $p_1 \in P_g$ and $p_2 \notin P_g$, and label $l \in \text{label}(p_1)$ prior to the information exchange, then $l \in \text{label}(p_2)$ after the information exchange.

2. If principal $p_2 \notin P_g$ and label $l \in \text{label}(p_2)$, then information was potentially exchanged between p_2 and a principal $p_1 \in \text{originated}(l)$.

We will prove the two properties using proof by induction

1. We need to show $\exists IE_n(p_1, p_2) \Rightarrow l \in \text{label}(p_2)$. We do this by a proof of induction over the length $n > 0$ of the information exchange path $IE_n(p_1, p_2)$ of operations:

   Base case: $n = 1$

   A potential information exchange path $IE_1(p_1, p_2)$ only exists when there exists any of the following operations:

   create(p_1, p_2) We have $l \in \text{label}(p_1)$ and $\text{label}(p_2) = \text{label}(p_1)$. It thus follows that $l \in \text{label}(p_2)$.

   open(p_1, p_2) and close(p_1, p_2) We have

   $$l \in \text{label}(p_1) \text{and} \text{label}(p_2) = \text{update}(\text{label}(p_1), \text{label}(p_2))$$

   By the nature of update() it follows that $l \in \text{label}(p_2)$.

Read and write operations are not part of the base case, as they must occur as write(a, b, x)/read(b, c, y) pairs/groups.

Induction hypothesis: assume $\exists IE_k(p_1, x) \Rightarrow l \in \text{label}(x)$ is true $\forall 1 \leq k < n$. Let op_{k+1} be the last operation in $IE_{k+1}(p_1, p_2)$, i.e., $IE_{k+1}(p_1, p_2) = IE_k(p_1, x) \circ \text{op}_{k+1}$. We have the following cases:

(a) If op_{k+1} is of the form create$(x, p_2), x \in P$, then from the induction hypothesis we know that $l \in \text{label}(x)$. The create() operation copies the label set of the creator to the created principal and we have $\text{label}(p_2) = \text{label}(x)$ from which follows that $l \in \text{label}(p_2)$.

(b) If op_{k+1} is of the form open(x, p_2), open(p_2, x), close(x, p_2), or close(p_2, x), $x \in P$, then we have $l \in \text{label}(x)$ by the induction hypothesis. Furthermore, from op_{k+1} we have

$$\text{label}(x) = \text{label}(p_2) = \text{update}(\text{label}(x), \text{label}(p_2))$$

Because of the property of update it directly follows that $l \in \text{label}(p_2)$.

(c) If op_{k+1} is of the form read$(y, p_2, r), y \in P$, then by definition $\exists \text{op}_i = \text{write}(y, p_2, w) \in \text{subIE}_{k+1}(p_1, p_2, k'), k' < k + 1$. This means that there exists an information exchange path $IE_{k'}(p_1, y)$ and by induction hypothesis $l \in \text{label}(y)$.

There is also possibly a sequence of read operations that read a total number of r' items from the channel $\langle y, p_2 \rangle$ after the write operation and prior to op_{k+1}. If $r' \geq c_i + w$, then all the information from the write operation has already been dequeued from the channel. This implies that there exists a shorter information exchange path $IE_{k''}(p_1, p_2), k'' < k + 1$ and from the induction hypothesis it directly follows that $l \in \text{label}(p_2)$. Otherwise, we have $r' + r > w$. As $l \in \text{label}(y)$ prior to op_i at least one of the label sets queued in $\langle y, p_2 \rangle$ contains l. The operation read(y, p_2, r) leads to $\text{label}(p_2) = \text{update}(\text{label}(p_2), \text{dequeue}(\langle y, p_2 \rangle))$ r times and thus $l \in \text{label}(p_2)$.

(d) If op_{k+1} is of the form $\text{read}(o, p_2), o \in O$, then by definition $\exists \text{write}(x, o) \in \text{subIE}_{k+1}(p_1, p_2, k'), k' < k+1$. This means that there exists an information exchange path $IE_{k'}(p_1, x)$ and by induction hypothesis $l \in \text{label}(x)$. The write operation results in $\text{label}(o) = \text{update}(\text{label}(x), \text{label}(o))$ and thus $l \in \text{label}(o)$. From the read operation we have

$$\text{label}(p_2) = \text{update}(\text{label}(o), \text{label}(p_2))$$

and it follows that $l \in \text{label}(p_2)$.

(e) If op_{k+1} is of the form $\text{open}(o, p_2), o \in O$, then by definition $\exists \text{create}(x, o) \in \text{subIE}_{k+1}(p_1, p_2, k'), k' < k+1$. This means that there exists an information exchange path $IE_{k'}(p_1, x)$ and by induction hypothesis $l \in \text{label}(x)$. The create operation results in $\text{clabel}(o) = \text{label}(x)$ and thus $l \in \text{clabel}(o)$. From the open operation we have $\text{label}(p_2) = \text{update}(\text{clabel}(o), \text{label}(p_2))$ and it follows that $l \in \text{label}(p_2)$.

□

2. We need to show $l \in \text{label}(p_2) \Rightarrow \exists IE_{n'}(p_1, p_2) \subset \text{OP}_n, p_1 \in \text{originated}(l), n' > 0$. We do this by a proof by induction over the sequence OP_n of $n \geq n'$ total operations from our model from the start of the information exchange.

Base case: $n = 1$

A principal can obtain a label only through the addlabel(), open(), close(), and read() operations. Because $p_2 \notin P_g$ the addlabel() operation was not the cause for $l \in \text{label}(p_2)$. Because $n = 1$, a read() operation also was not the cause as it requires a previous write(). To obtain l with one single operation, the operation had involve a principal $p_1 \in P_g$. The label l was propagated to p_2 as a result of one of the assignments $\text{label}(p_2) = \text{label}(p_1)$ (create) and $\text{label}(p_2) = \text{update}(\text{label}(p_1), \text{label}(p_2))$ (open, close between principals). A write() operation does not directly affect a principal's label set. Hence the label must have been obtained via one of the following operations: $\text{create}(p_1, p_2)$,

open(p_1, p_2), or close(p_1, p_2). Each of these operations is a valid information exchange path $IE_1(p_1, p_2)$.

Induction hypothesis: assume $l \in \text{label}(p_2) \Rightarrow \exists IE_{n'}(p_1, p_2) \subset \text{OP}_k, p_1 \in \text{originated}(l)$ is true $\forall 1 \leq k < n$

As $p_2 \notin P_g$, the label l must have been obtained by the latest operation performed in the model and it must have involved p_2. If l was obtained through an earlier operation then there exists a shorter potential information exchange path, and it directly follows from the induction hypothesis that $\exists IE_{n'}(p_1, p_2), p_1 \in \text{originated}(l)$. We have the following cases:

(a) If the last operation $\text{op}_{k+1} \in \text{OP}_{k+1}$ was one of create(x, p_2), open(x, p_2), close(x, p_2), or close(p_2, x), and l was obtained by p_2 as the result of this operation, it follows that $l \in \text{label}(x)$. By induction hypothesis $\exists IE_{n'}(p_1, x)$ and we have $IE_{n'+1}(p_1, p_2) = IE_{n'}(p_1, x) \circ \text{op}_{k+1}$.

(b) If op_{k+1} was of the form read(x, p_2, r), and l was obtained by this operation, then l must have been part of one of the r number of label sets contained in the channel queue. This means that there must have been an operation $\text{op}_{k'} = \text{write}(x, p_2, w) \in OP_k$ prior to op_{k+1}, $w > 0$, and $l \in \text{label}(x)$. Furthermore, the number of items c contained in $\langle x, p_2 \rangle$ prior to $\text{op}_{k'}$ plus w has to be greater than the number of items read from the channel from possible read operations in between $\text{op}_{k'}$ and op_{k+1}. If this is not the case, l was propagated to p_2 by some prior read operation already and it directly follows from the induction hypothesis that $\exists IE_{n'}(p_1, p_2)$, where $p_1 \in \text{originated}(l)$. Otherwise, by induction hypothesis $\exists IE_{n''}(p_1, x)$. The existence of $\text{op}_{k'}$ together with op_{k+1} fulfill the definition of an information exchange path $IE_{n''+1}(p_1, p_2) = IE_{n''}(p_1, x) \circ \text{op}_{k+1}$.

(c) If op_{k+1} was of the form read(o, p_2), it implies that $l \in \text{label}(o)$ and thus there must have been an operation $\text{op}_{k'} = \text{write}(x, o) \in OP_k$, at which point we had $l \in \text{label}(x)$. By induction hypothesis we have $\exists IE_{n'}(p_1, x)$,

where $p_1 \in \text{originated}(l)$ and it directly follows that $IE_{n'}(p_1, x) \circ ... \circ \text{op}_{k'} \circ ... \circ \text{op}_{k+1} = IE_{n''}(p_1, p_2)$.

(d) If op_{k+1} was of the form $\text{open}(o, p_2)$, it implies that $l \in \text{clabel}(o)$. This means that there must have been an operation $\text{op}_{k'} = \text{create}(x, o) \in OP_k$, at which point we had $l \in \text{label}(x)$. By induction hypothesis $\exists IE_{n'}(p_1, x)$, where $p_1 \in \text{originated}(l)$ and it directly follows that $IE_{n'}(p_1, x) \circ ... \circ \text{op}_{k'} \circ ... \circ \text{op}_{k+1} = IE_{n''}(p_1, p_2)$.

□

Note that if a space management mechanism is in place and it is of the *loose* type as discussed above, then labels are potentially dropped. This means that the first property of the model may no longer hold. The second property (if a label is present information must have been exchanged), however, always holds. For *strict* models, both properties hold.

Also note that a direct consequence of the model's properties it follows that

$l \notin \text{label}(p_2) \Leftrightarrow \nexists IE_n(p_1, p_2); \forall n, p_1 \in P_g$

i.e. if no label is present at a principal p_2, then no communication has taken place between principal p_2 and a principal from P_g.

4.5 Case Studies

In the following we present several case studies to demonstrate how to use the general model to create a sub-model to achieve desired accumulation of information. Most of these case studies are based on a single host computing environment because this was the main motivation for the development of the model.

4.5.1 User Influence Labels

To accomplish user influence tracking as described in Section 3.2.1 using label propagation, we implement our model as follows. The system processes on the sys-

tem are our principals as they are the active subjects that initiate communication exchanges. Passive objects are all system resources that are shared between processes. This includes files, shared memory, devices, global variables, mutexes, and locks. For each of those objects the system needs to manage the associated label sets. For those objects that may be dynamically created by processes (files, shared memory) the system also needs to keep track of their creator label (clabel) sets.

The creation of new processes is usually handled through system calls such as `fork` under Unix. At this point an exact copy of the process is made but with a new process id. The label set associated with the process may be copied for the new process at this point as well.

Channels between principals are the types of communication exchange channels between processes: sockets, pipes, and signals. For each of those channels the system needs to keep track of the label sets associated with the data. As most operating systems use data queues to store the elements in the channel that have not been read, an implementation should not cause much overhead. The operations on channels in a system closely resemble those of our model and the implementation should be straightforward. For channels that do not require an explicit open, we may simply drop the requirement that an open channel must exist for the read/write operations.

Channels between a principal and an object are abstract in this case and it is sufficient to compute the label sets of the subjects involved at each operation.

The label set for user influence consists of all the user identifiers on the system. On many systems this number is typically limited by the size of the user id field – 16 bits for Unix-like operating systems. Furthermore, there is often an upper limit set on the number of possible users that is directly compiled into the kernel. Thus we have a limited number of elements in the set, that, in many cases, is not large.

Because we want to track all possible influence any user might have exerted, the label updating function *update* needs to preserve all labels and combine the label sets of interacting subjects. We therefore choose update $= \cup$.

Finally, the binding of a user label to a process running on the system should occur after some sort of user authentication. On many systems there are system calls that change the user context of a process to that of a specified user (`setuid` on Linux, for example). These system calls are therefore a natural choice to perform our `addlabel` operation. Depending on the system, some modifications may have to be performed to ensure that only the process resulting from the login carries the user label but not the system process that manages the login processes for everyone. For example, under Linux, there are several processes running the `getty` program for users to log onto the terminal. It must be ensured that the adding of the label occurs after the forking of the process. Otherwise, the process running `getty` will accumulate all user labels of subsequent logins and pass those on to to its child processes.

As a result of the likely low number of users on the system space for labels should not be a problem. The upper limit for the label space per subject is the product of the size of the user id field and the maximum number of users allowed. Therefore, on many systems the space management discussed above will not be necessary.

4.5.2 Host Causality Labels

For host causality, we have an environment similar to that of user influence. We have the same set of principals, objects, and channels, and the propagation techniques are the same as well.

Also, we want to keep track of all network labels related to the processes in the same manner as we wanted to keep track of the user label from the previous scenario. Thus we also set update $= \cup$ in this case.

For the label set we need to choose labels that uniquely identify the network connection. For TCP/IP this could be the tuple consisting of local port as well as foreign port and IP address. Note that this label set is quite large (2^{64} possible labels for IPv4).

The adding of a label to a process should occur whenever the process receives data from a network connection. For TCP the reception of data is initiated by the three-way handshake and from a process' perspective whenever the listening process accepts a connection. Thus system calls such as `accept` under Linux can be used to perform the `addlabel` operation. Under UDP there is no concept of a connection and data is read by the listening process directly via system calls such as `recvfrom`. This binding of labels is identical to the setting of process origin in our previous work [11].

Because the label set is large, it is infeasible to allocate sufficient space for all labels per subject. Therefore, one of the space management techniques discussed earlier is recommended. It is desired by the nature of the information we seek that no labels be lost. Because of this, a strict model as defined above is required. It is part of future research to determine what a good balance between how much space to make available and disturbing a system's normal course of operation by disallowing operations should be.

With host causality labels it is now possible to correlate incoming and outgoing network traffic on the host. In particular, this may aid in the stepping stone detection network traceback research discussed in Section 2.3.2 and a host supporting host causality labels may be considered as an internal sensor as defined by Daniels [24].

4.5.3 Network Location Traceback Labels

We can extend the above host causality scenario to a complete network traceback environment. Principals now are all processes on all systems in the environment, and the object are all the system resources on all systems. Obviously, objects may only exchange information with principals on the same system. The label propagation mechanism within each system is exactly the same as described in the first case study. In addition to that labels need to be propagated when principals on different systems communicate with each other. For this purpose, a mechanism needs to be

devised that safely may transport labels across a network. This is part of future research and beyond the scope of this dissertation.

The label set should contain labels that uniquely identify the location of a subject initiating a cross-system communication. This could be a network address, such as the IP address of the system, or things such as GPS coordinates or GUIDs. We expect the label set to be quite large for the general case. Special cases may exist where there is a small maximum label set size, such as network traceback within intra-networks that contain only a small number of hosts.

The actual binding of the label via the `addlabel` operation is not as clear as in the above examples, though. The binding could occur automatically for data leaving a system. E.g. the network stack code of the system could add a label for the current system if it is not already present. Alternatively, the label could be bound when a user logs in locally to a system as described in the first case study. Yet another way to perform the binding would be an ISP who adds a label to incoming traffic on the border routers. If the ISP's infrastructure fully supports the label propagation, then traffic leaving the ISP's network through a border gateway may be correlated to traffic entering it.

It is obvious that the modifications and requirements of this implementation are restrictive and will find its use only in controlled environments where complete network traceback is desired and its benefit outweighs the restrictions. This case study is similar to the approach taken by Zhang and Dasgupta [119] (see Section 2.3.1) with the added benefit that hosts being used as stepping stones within an autonomous system are now also addressed.

4.5.4 Military Classification Labels

In many environments where confidentiality is a concern, the disclosure or compromise of some information may result in greater loss than disclosure of less sensitive information. Thus, to help provide graduated protection, it is helpful to partition in-

formation into categories based on potential loss. The military model of classification is one such partitioning [79].

In the US military model of *classification*, information is divided into unclassified (minimal or no loss if disclosed), FOUO (for official use only; unclassified, but restricted release), confidential (disclosure may cause some loss), secret (significant damage if disclosed), and top secret (grave damage if disclosed).

Persons (and equipment) that are to have access to information in this system are required to be *cleared* to the highest level of access needed to accomplish their tasks. It should be noted that simply because someone has a clearance at a particular level does not mean that he/she can access all information that is classified at that level. Instead, there needs to be a "need to know" the information as a further condition of disclosure. There may also be additional restrictions that are imposed by the stewards of particular data, such as it cannot be shared with allies (even if they have the right level of clearance), or the existence of the data itself must be denied. It may also be the case that information from two different sources should never be combined in one place because the combination might lead to inference of something sensitive not otherwise knowable from the individual parts. These classes of information are given labels such as Umbra or Majic. Often, these labels are themselves classified, and an unclassified abbreviation is used, such as MJ or GG. To access data in one of these categories or *compartments*, an individual must be "read in" to the special conditions of access to that category.

The combination of levels and categories forms a matrix to determine access control. To read a piece of information, a person must be cleared to at least the level of that information (secret or top secret), AND someone in charge of that information must agree that the person has a need to know the information, AND the person must be read into all of the categories that label the information. So, for example, if a datum is Secret and in compartments A, B and X, a person with Top Secret clearance would still not be granted access if she only had been granted access to B and X.

Information that is composed of other information or processes inherits all the categories of the component parts, and it takes on the highest level of classification of any of the component parts. For example, combining item 1 at Secret MJ, item 2 at Top Secret AA, and item 3 at Secret XQ would result in a new item at level Top Secret with labels MJ, AA, and XQ. Note that there might be special rules associated with MJ, AA or XQ that would restrict (or prohibit) this combination.

We can use our model to keep track of information flow violations (or, in combination with access control measures, prevent them), while automatically updating the classification information of documents as they are modified. In this case, our labels represent the most restrictive set of levels and categories that were actively accessed by a principal during one "session." A session is the duration in which a principal is active within the system (e.g. the period of time during which a user is logged in to a computing system). Note that the label set does not denote the principal's actual clearance credentials.

For reasons of simplicity, let's assume we have three classification levels: *unclassified* (u), *secret* (s), and *top secret* (t), and four categories: A, B, C, and D. The label set consists of the cross product between the levels and the possible subsets of the categories (i.e. $\{u, s, t\} \times 2^{\{A,B,C,D\}}$). Furthermore, there is an ordering $u < s < t$ of the levels. We define our label-updating function as max for the level-part and as $\cup$ for the category part.

At the start of a session a principal's label set is empty. Only through access to classified documents and communication with principals that already have acquired labels does a principal acquire labels itself. This ensures that a principal may still communicate with other principals of lower or different classification prior to the access of higher classified documents, but not thereafter. An access violation has occurred when a principal holds a label with a higher level or different categories than its actual clearance.

The actual implementation may be a computing system as described in the first case study. The label propagation is analogous. The documents in this case are the

objects on the system. When two principals communicate with each other, their label sets will be modified according to the update function. This guarantees that any documents that are created are always classified with the highest security levels necessary and the classification of objects that are modified is also properly updated.

5 IMPLEMENTATION

In the following we discuss the implementation of our model as a proof-of-concept study. The model was implemented by modifying the FreeBSD 4.12 operating system [40]. The implementation is kernel-based, which means that the propagation method lies in a protected space that cannot be tampered with from user mode programs. If the kernel can be trusted, then so can the label propagation and any information gained by it. Implementing the model for a real production operating system as opposed to a simulated one serves multiple purposes:

- We demonstrate that utilizing the model is feasible for modern operating systems.
- We can accurately measure the computational overhead needed for label propagation to work.
- We encounter and can address the difficulties and limitations that come with such an implementation.
- Results obtained from this proof-of-concept implementation may be applied to other operating systems with similar architectures.

Section 5.1 describes which subsystems would need to be addressed for a full implementation of the model we described. Given that our implementation is merely a proof-of-concept, we focus on the major subsystems needed for label propagation to function. It addresses all the important aspects that need to be considered, and most of the other subsystems can be implemented in a similar fashion. In the subsequent sections we describe the parts that were actually implemented, namely: the data structures and operations introduced to the kernel (Section 5.2), how to

modify network sockets as part of interprocess communication (Section 5.3), and how to handle label propagation for files (Section 5.4).

5.1 Subsystems Affected by Label Propagation

Applying our model for a computing system means that we need to translate the sets of principals, objects, and channels to entities of the system. Furthermore, the operations of the model need to be implemented accordingly. The acting principals on a computing system are its processes. Therefore, the set P maps to the set of processes on the system. This is consistent with the notion that on all computing systems, there is one initial process (such as `init` in the UNIX world), from which subsequently all other processes on the system are created (unless it is not a multi-process system, in which case there is only one initial process). The set of objects O of the model are all the resources that are shared on the system among processes. Those resources that belong solely to one given process need not be considered. Furthermore, the set of possible channels is comprised of the system's provisions for interprocess communication, as well as the means to access, modify, and create shared resources. To implement label propagation for an entire operating system, essentially three parts need to be completed:

1. The system needs to be aware of labels and be able to bind them to its principals.

2. All interprocess communication needs to be covered by the label propagation mechanism.

3. All objects shared among processes need to be associated with labels for the duration of the sharing and those labels updated according to the model.

5.1.1 Shared Resources

The resources that are shared among processes on a system are numerous. In FreeBSD and other UNIX-type operating systems they consist of at least the following [99]:

- Files. Files are data objects typically stored on secondary storage. Files are generally shared between numerous processes. While there may be a limitation to only those processes that possess the proper access credentials to perform operations on a file, those credentials are on a user/group basis and not on an individual process basis. Furthermore, the access credentials are subject to change arbitrarily, which means that labels need to be kept for files even if only one process has access to a file at a given time. Files are considered to be permanent objects on a computing system.

- Mutexes and Condition Variables. Mutexes are used for synchronization on a system. They are shared between entities within the same name space as the mutex. A free mutex may be locked by one entity at which point all other entities trying to obtain the lock have to wait until the mutex is unlocked. Condition Variables are used in conjunction with mutexes to signal any waiting entities that some event (usually a change in a variable) has occurred. Mutexes typically only occur within a given process, but it may be possible to share a dynamically allocated mutex among processes through shared memory.

- Read-write and Record Locks. Unlike mutexes, read-write locks do not block access to an entire "critical region" protected by the lock. Read-write locks differentiate between read and write mode: any number of processes may hold a read mode lock as long as no process holds a write mode lock, and a write mode lock can only be obtained when no other process holds any mode lock. Record locks further govern read and write access to specific regions of a file. They work like read-write locks, but one can specify if the entire file is locked

or merely a specific byte-region. As with mutexes, read-write locks are usually shared only by threads within a process. Record locks, however, are shared between processes.

- Semaphores. Semaphores are explicitly used for synchronization between processes. As with mutexes, semaphores can be locked (set to 1) and waited for (wait for value to be 0), but the value of a semaphore may also be read without blocking. Essentially, a semaphore is a 1-bit global variable shared among all processes. Some semaphores (such as System V semaphores [99]) also allow counting semaphores, whose values range between zero and some specified upper limit. Semaphores are temporary objects that exist in the system until they are destroyed.

- Shared Memory. Shared memory is heap memory that is mapped to the address space of multiple processes. Once the system has mapped the address space, only the processes that have access to the shared memory are involved in passing data to and from the memory space. A special case of shared memory is the memory-mapping of a file. In this case the file data is mapped into memory and the process(es) can directly modify the memory, affecting the file contents as well. When synchronizing the memory space with the file, the system is again involved.

- Sockets and Message Queues. Certain parameters can be set and retrieved for sockets and message queues as options. Thus, they qualify as shared resources through which data may be exchanged as well. We discuss sockets and message queues as part of interprocess communication in further detail below.

The implementation we describe in the following sections handles label propagation for a fixed number of globally shared labels. That means that there is an upper bound on the number of labels, which allows us to allocate sufficient space for the processes and objects or fail during their creation. Furthermore, the update function

will always succeed and we do not have to worry about the measures to take when there is insufficient space for labels as a result of the update (see Section 4.3.1).

As there is no system involvement when shared memory is processed, there is no elegant solution to keep track of label propagation. One might imagine monitoring all the system calls that involve copying of data between buffers and determine if any of the addresses involved fall into a shared memory region. However, because data may directly be assigned in chunks of up to a word length, this will not cover all of the information flow between processes. Barring the utilization of static analysis techniques as discussed in Section 2.2.2, a monitoring device such as Fenton's Memory Mark machine [37], or the utilization of virtual machines or special hardware, the only reliable method to properly implement the model is once again conservative: all processes that share memory between them need to be treated as a single principal during the duration of the sharing. For an implementation of this, for every operation that involves a principal, further lookups will have to be made to see if it belongs to such a group and then perform the operation for all processes involved. The utilization of shared memory is a feature not common to many programs. Thus, the inclusion of such a mechanism lies well outside of a proof-of-concept implementation, and we will not consider shared memory any further (and as a direct consequence we also will not have to consider mutexes, condition variables, and read-write locks).

5.1.2 Interprocess Communication

Interprocess communication mechanisms allow two processes on a system to exchange data. In UNIX-like operating systems there are the following mechanisms available [65, 99]:

- Pipes and FIFOs. A pipe is the original UNIX mechanism for interprocess communication. A uni-directional pipe consists of two file descriptors, one for writing and one for reading. Bi-directional pipes consist of two uni-directional pipes. Pipes have no name associated with them, and can only establish com-

munication between related processes (parent and child). This was changed with FIFOs (also called *named pipes*).

- Sockets. Sockets are the systems interface for sending and receiving data over a network. With facilities such as the *loopback device* and socket types for local data transfer (UNIX domain protocols), sockets can also be used for processes to communicate with each other.

- Message Queues. Message queues are used to deliver *message records*, which may contain arbitrary data, between processes. Unlike pipes and sockets, message queues need not be open between two processes. A process may write data to a message queue and then terminate, and a second process can retrieve the message record even after the termination of the writing process.

- Remote Procedure Calls. Remote procedure calls occur when a process invokes a procedure that is not located in its own environment but rather within a different process on the same host (doors) or on a different host (Sun RPC). Within a process the doors are identified by descriptors and externally by pathnames in the file system. Doors are not supported in FreeBSD and the Sun RPC calls utilize the socket mechanism to transport data.

- Signals. Signals are software interrupts that allow a process to react to asynchronous events. Apart from handling signals that originate from certain external events (such as a user aborting a program), processes can send signals to other processes and thus count as interprocess communication [97].

5.1.3 Operations and System Calls

In FreeBSD and many other operating systems the way a process is able to communicate with other processes or access system resources is governed through the use of system calls. System calls are the interface from user space processes and the operating system kernel. The shared objects are created and manipulated that

way and all the interprocess communication mechanisms utilize system calls. In the following we describe which FreeBSD system calls are relevant for the operations of our model.

Create

The only way to create a new process apart from `init` and the page daemon is through the `fork` system call [97]. The function is called once, but returns twice: once to the calling process (parent) returning the process id of the child process, and once to the newly created process (child) returning a value of 0. While there are the system calls `system` and `popen`, which also create new processes in the system, they do so by invoking `fork` and then one of the calls from the `exec`-family. Thus it is sufficient to address the inheritance of label sets at the `fork` system call.

There is a large number of system calls available to create shared objects. Files can be created with the `open`, `creat`, and `mknod` system calls. Record locks are created by invoking the `fcntl` system call on an open file descriptor with a command indicating the setting of the lock and a `struct flock` with types `F_RDLCK` or `F_WRLOCK` as parameters. Semaphores can either be named, in which case they are created with `sem_open` and `semget`, or they can be memory-based, in which case they are created with `sem_init`. However, the former case also requires shared memory between processes. In all of the above cases, the system must make sure that the newly created object's creator label set must be set to the label set of the process that initiated the system call.

Open

Pipes are created with the `pipe` system call. The call returns the two file descriptors to the process. After that the process usually calls `fork` and the parent and child process each will close one of the descriptors. Alternatively, the `popen` call combines these steps and further executes a command for the child process. The updating of

the label sets is already taken care of through the call to `fork`. A socket is created with the `socket` system call, but the communication channel is not open at this time. The `socketpair` system call, similar to `pipe` creates two connected sockets, one of which is usually passed to a child process through `fork` or through another communication channel supported by the system. This only applies to sockets of the Unix domain protocol suite. Because sockets are considered shared resources in the system, at the point of the creation the label set of the calling process has to be associated with the socket. With the creation of the socket, the communication channel is not opened, however. This is done through a series of system calls on both client and server side (`bind`, `listen`, `connect`, `accept`) for stream sockets. For datagram sockets no explicit opening of a channel is performed. Data is exchanged on a per packet basis through the read and write operations. Upon a successful opening of a stream socket the label sets of the processes involved need to be updated with each other (if both endpoints are local).

Message queues are not explicitly opened as they do not require two endpoints for communication. A message queue needs to be explicitly created, however. This is done with the `msgget` system call. In the case when the system call is used to create the queue (it is also used to read data from it), the label set of the creating process should be associated with the queue, as the queue also qualifies as a shared resource whose existence may be used to exchange information. Signals are passed on by the system instantaneously, without an explicit opening of a channel.

Opening a channel from a process to a shared object may not always occur. Obtaining the status of a lock or reading the value of a semaphore does not require an explicit setup of a channel. Thus the only explicit opening of a channel regarding the shared objects occurs for files. Here, the channels are opened via the `open` system call. Depending on the flags that are passed to the system call, the channel is either read-only, write-only, or bi-directional. In all the cases, the label set of the process that invoked the `open` call needs to be updated with that of file's creator label set.

Write

Writing data to a channel is different from storing data in an object. This is because the data may not immediately reach its destination, but rather is stored in the channel until it is read by the reading process. For this purpose, the channel endpoints (descriptors) usually have send and receive buffers, where the data of the writing process is put into the send buffer, and data is moved from the receive buffer to the reading process. In some cases send and receive buffer are the same, and in other cases there is some transport mechanism between the buffers. Labels need to be associated with each new data instance (packet) that enters the send buffer. Thus, if a packet of size n is written to the channel, the label set of the writing process needs to be associated with that packet, and this is equivalent to the n number of *enqueue* operations of our model (see Section 4.2). Figure 5.1 illustrates the way data is propagated through channels.

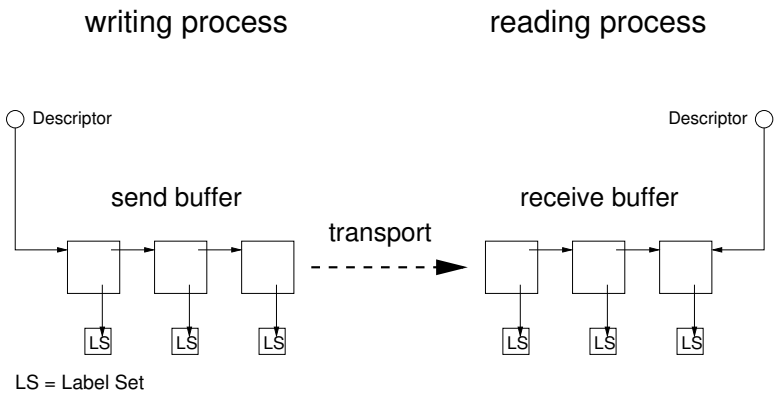

Figure 5.1. Data flowing through a channel

Pipes use the `write` and `writev` system calls to write data into the channel. Writing data to a socket is performed by one of the following system calls, depending on the socket type: `write`, `writev`, `send`[1], `sendto`, and `sendmsg`. For all the pipe

[1] `send` is not actually a system call but rather a library function that utilizes `sendto`. However it is listed as a system call in the FreeBSD manual pages.

and socket write calls the data that is sent to the send buffer needs to be tagged with the label set of the calling process. Furthermore, for sockets the `fcntl`, `ioctl`, and `setsockopt` system calls can be used to modify parameters of the socket that the corresponding reading process may interpret. Thus, for those calls, the label set associated with the socket needs to be updated with that of the calling process.

For message queues the send and receive buffers are identical and implemented as a priority queue. Furthermore, there are no specific descriptors that each process possesses but rather global endpoints open to all processes. Writing data to a message queue is performed via the `msgsnd` system call. At this point, the label set of the calling process needs to be associated with the message packet that is put into the queue. Certain parameters of the queue may also be set with the `msgctl` call. In this case the label set associated with the message queue itself, rather than those associated with the messages, needs to be updated with the calling process's label set. Signals are sent via the `kill` system call. They are delivered immediately and there is no explicit "reading" of the signal. A process is either able to handle the signal, or the signal is ignored. Thus for the `kill` system call the label set of the process receiving the signal needs to be updated with the label set of the calling process.

Writing data to a record lock is equivalent to creating it or performing the unlock procedure. The latter is also done via the `fcntl` system call, one of the `F_SETLK` commands and the `F_UNLCK` type. Semaphores, with their system wide accessibility, can be manipulated by other processes once they are created. The value of a non-counting semaphore is changed to 1 (locked) with the `sem_wait` and `sem_trywait` system calls, and set to 0 (unlocked) with the `sem_post` call. For counting semaphores, the value of the semaphore is manipulated with the `semget`, `semop`, and `semctl` system calls. Data is written to file objects when either their data is modified or when the metadata of the file changes. The system calls associated with altering files are `write`, `writev`, `pwrite`, `open` (with the `O_TRUNC` flag set), `fcntl`, `ioctl`, `truncate`, `ftruncate`, `chmod`, `fchmod`, `lchmod`, `chflags`, `fchflags`, `chown`, `fchown`, `lchown`,

`utimes`, `lutimes`, `futimes`, `rename`, and `link`, `symlink`, and `mkdir` for directories. In all these cases, the label set associated with the object needs to be updated with the label set of the calling process.

Read

As described in the previous section, reading data from a channel usually involves reading data from the receive buffer of the reading process's channel descriptor. Pipes use the `read` and `readv` system calls to read the data. Data is read from a socket through the `read`, `readv`, `recv`[2], `recvfrom`, and `recvmsg` system calls. For the read calls for pipes and sockets, the label sets of the packets that are read from the receive buffer must be used to update the label set of the process invoking the read. For sockets, the `fcntl`, `ioctl`, and `getsockopt` calls can be used to retrieve parameters from the socket. In this case the calling process's label set needs to be updated with the label set associated with the socket itself. Reading messages from a message queue is done with the `msgget` and `msgrcv` calls. Certain parameters of the queue may also be accessed through the `msgctl` system call. In the former case the label set associated with the message(s) is used to update the calling process's label set, in the latter case the label set associated with the queue itself must be used. For signals, there is no explicit read mechanism. Instead, the program the process executes must be coded to handle signals (see section above).

Reading data from a record lock is equivalent to trying to obtain a lock. This is done via the `fcntl` system call on an open file descriptor but this time with an `F_GETLK` command. The value of a non-counting semaphore can be obtained via the `sem_trywait` call (implicitly) or via the `sem_getvalue` system call. The value of counting semaphores is read with the `semget`, `semop`, and `semctl` system calls. Data that can be read from files is either the file data itself, or the metadata of the file. The system calls associated with reading data from a file are: `read`, `readv`,

[2] `recv` is not actually a system call but rather a library function that utilizes `recvfrom`. However it is listed as a system call in the FreeBSD manual pages.

`pread`, `stat`, `lstat`, `fstat`, `poll`, `access`, `chdir`, and `fchdir`, as well as `readlink` for directories.

Close and Destroy

When a channel is destroyed, all the unsent and/or unread data in the send and receive buffer is discarded and will not have any effect on label updating. Pipes are closed with the `pclose` system call. Sockets can be closed with the `shutdown` and `close` system calls. If `shutdown` is used, the socket merely shuts down communications in one or both directions, but the socket can be reconnected and parameters still be set and read. In this case the label set of the socket needs to be updated with the label set of the calling process. If `close` is used and the socket is connection-oriented (streaming), then the two label sets of the processes involved need to be updated with each other if both endpoints are local. Message queues are destroyed via the `msgctl` system call and the `IPC_RMID` command. When a message queue is destroyed, the label sets of any remaining processes that block for a read access need to be updated with the label set of the calling process and vice versa. Signals do not possess a close or destroy mechanism.

When closing an open channel to a file or removing a lock, according to our model, no label sets need to be modified. Therefore none of the respective system calls need to be modified. The same is true when destroying the object.

A process is destroyed in two ways: it either ends the program it is running and returns, or it receives a `SIGKILL` signal. The existing open communication channels the process possesses are closed by the system and thus no label sets need to be updated. As with closing a channel to a shared object, destroying a shared object has no effect on any label sets.

The number of system calls we have listed above that need to be modified to allow full label propagation according to the model we presented in Section 4.2 is extensive. Some of the implementation work may be reduced as some system

calls share lower level functions in the kernel, where the appropriate modifications may be performed (see the following sections). However, most of the complexity arises from the need to address communication through storage channels. In the general case processes will not abuse mechanisms such as the existence of files, locks, semaphores, or message queues, changes in socket parameters, or changes in file metadata to exchange information. With the introduction of observation techniques such as the ones we discuss in this document, this may change, however. If all legitimate channels on a system are effectively monitored then malicious users will find ways to circumvent those channels. However, addressing those storage channels in a proof-of-concept implementation is well beyond the scope of the work we present here. Therefore, for the proof-of-concept implementation we will pick the most relevant subsystems and focus only on implementing label propagation for the data channels on the system. That means that we will ignore the open and close operations of our model, and do not keep track of creator label sets. The techniques we describe in the following should apply for most of those instances, and many of the system calls we do not address can be modified in the same manner as we describe below.

In addition to the modification of the systems calls, we also need to be able to map certain information with respect to labels. Not all of the mappings are necessary, but they may improve performance of certain operations regarding labels. The mappings are:

- process id $\rightarrow$ label set
- object identifier $\rightarrow$ label set
- label $\times$ label set $\rightarrow$ boolean
- label $\rightarrow 2^{\{\text{processes}\times\text{objects}\}}$
- object identifier $\times$ label $\rightarrow$ boolean
- process id $\times$ label $\rightarrow$ boolean

The last three mappings are not required for label propagation to work. Actually, the last two are merely combinations of the first three. However, they may allow testing in a more efficient manner whether a given label is part of a process's or object's label set and also obtaining all the processes and objects whose label sets contain a specific label.

5.2 Data Structures and Operations

The main data structure that manages the labels in the kernel is a global table `label_table`, which is an array that contains the label data as well as its position within the table.

```
struct label {
  char data[LABEL_SIZE]; // actual label data
  long pos; // position in the global array
};
extern struct label *label_table[LABEL_SET_MAX];
```

This allows the referencing of a label to be its entry number within the label table. Thus, any given label set associated with a process or object can be as simple as a bit-vector, where each bit signifies whether the label at that position is contained in the set or not. That means that any given label set is a bit vector with as many bits as there are possible labels:

```
typedef struct _labelSet {char v[LABEL_VECTOR];} labelSet;
```

The constants have the following relationship (there a 8 bits in a byte):

```
#define LABEL_VECTOR (LABEL_SET_MAX / 8)
```

The relationship between label sets and the global label table are shown in Figure 5.2. Note that the actual content of the label is not important for the propagation at all. The operations described in the following will mostly involve only label sets.

It is conceivable to have a further mapping that, given a label, finds the position in the table. This may be useful to determine if a potentially new label to the system is already contained in the label table (and thus is not new). This may be achieved, for example, with a binary search tree using the label data as a key that maps back to the position in the global table. The global table will still be necessary, because once a position has been assigned to a label it must be permanent or the label sets will carry incorrect information. Plus, we achieve a $O(1)$ lookup for a given position in the table as opposed to the $O(\log n)$ such an operation would take if only a search tree were realized.

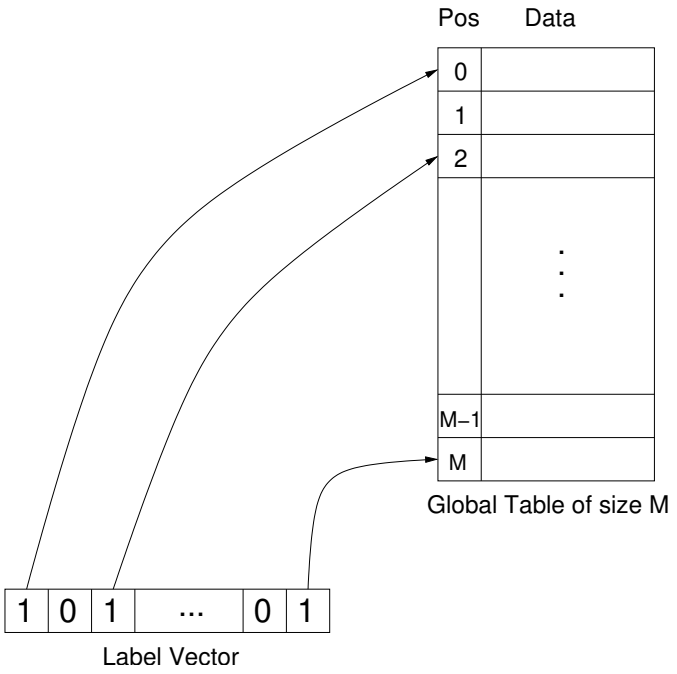

Figure 5.2. The main kernel data structure for labels

The `create_label` operation takes as arguments a buffer containing the label data and the length of the buffer. If the label is already contained in the label table, a pointer to that label is returned. Otherwise a new label is created in a free slot in

the table and a pointer to the label is returned. If no free slot is available, then a null pointer is returned.

Note that the system is not aware of any specific meaning of a label (i.e. user identity or location information). The system's purpose is merely to propagate the labels according to the operations of the model. Any interpretation of the labels is done by other extensions of the system (i.e., access control mechanisms, logging facilities, etc.), which are outside the scope of this work.

The `add_label` operation takes as arguments a label and a label set and sets the appropriate bit in the bit vector to `true`.

The `update_labels` operation takes as arguments two label sets: a source and a target. It then assigns the target label set the bit-wise "OR" of the source and target vectors.

To provide an interface to user space that allows user programs to query processes' labels, two new system calls were introduced to the system. The `getlabel` call retrieves the label set for a specified process id and stores it into the supplied buffer. The `getlabeldata` call retrieves the label data for the specified entry in the global label table and stores it in the supplied buffer. It is thus the responsibility of the user space program to compute the position in the label table from the bit vector and then retrieve the data.

For debugging purposes we also implemented another system call, `addlabel`, which binds a new label to a specified process. This call would not be part of a regular system.

Note that there are no operations removing labels from the system. That means that the proof-of-concept implementation will only be useful to a limited degree for label sets that are not fixed in size. This could be remedied by introducing a reference count for each label and, once the reference count reaches zero, removing the label from the table. However, this would imply that the update function no longer is a simple "OR" operation. The `update` function now would need to determine which labels were newly added to label sets as part of the update and increase the reference

count. Furthermore, whenever a process, shared resource, or label set residing in a channel gets destroyed the reference count needs to be decreased.

With the removal of labels from the label table we also introduce the problem of keeping track of the next free available slot in the table. However, this problem is similar to that of determining the next free process number of a system and can be solved in a similar manner.

With such a mechanism in place, however, the system could address unbounded label sets to a limited degree depending on the overall size of the label table and the retention of individual labels within the system. If a point is reached where no more labels can be assigned (or some threshold is reached), an alert could be issued, calling for resolution by a human being.

5.3 IPC: Sockets

From the different types of interprocess communication described in Section 5.1.2, the socket subsystem is the most complex one. The FreeBSD implementation of pipes utilizes supposedly the socket infrastructure to transmit data [65], but our performance overhead results in Section 5.5.4 indicate otherwise. Furthermore, the Sun RPC mechanism also uses sockets and the network subsystem to function. That and the fact that message queues are commonly not used too often by programs led us to implement label propagation for the socket subsystem.

As described in Section 5.1.3, sockets utilize send and receive buffers to store pending packets until they are sent over the network or read by the receiving process. The main data structure used to assemble those packets is the *mbuf*. Mbufs are used to build packets. They are small building blocks that contain space for a small amount of data and can be chained together to provide space for larger packets. Because of their small size, it is not feasible to store a label set within an mbuf. An mbuf only has 128 bytes of total space available. Furthermore, not every mbuf needs to be associated with a label set. The start of each data packet is indicated with a

special mbuf that has a packet header, and it is sufficient to associate labels with those mbufs. Thus we modify the `struct pkthdr` for the mbufs to contain a pointer to a label set. This pointer is initialized to zero every time a new packet header mbuf is allocated through the `M_GETHDR` macro. When mbuf are released back to memory, we also need to free the label set associated with it if it is of the type `M_PKTHDR` and actually contains a label set. This is done in the `m_free` function. When mbufs are copied with the `m_copym` function a new copy of the label set needs to be generated to avoid the same label set to be freed twice.

The system calls that we need to consider for an implementation are `read`, `readv`, `recv`, `recvfrom`, `recvmsg`, `write`, `writev`, `send`, `sendto`, and `sendmsg`. However, there are lower level functions that are invoked by those system calls, which, in turn, all call the `soreceive` function for the reading calls and `sosend` for the writing calls. Figure 5.3 illustrates the function hierarchy and the overall network stack structure of FreeBSD.

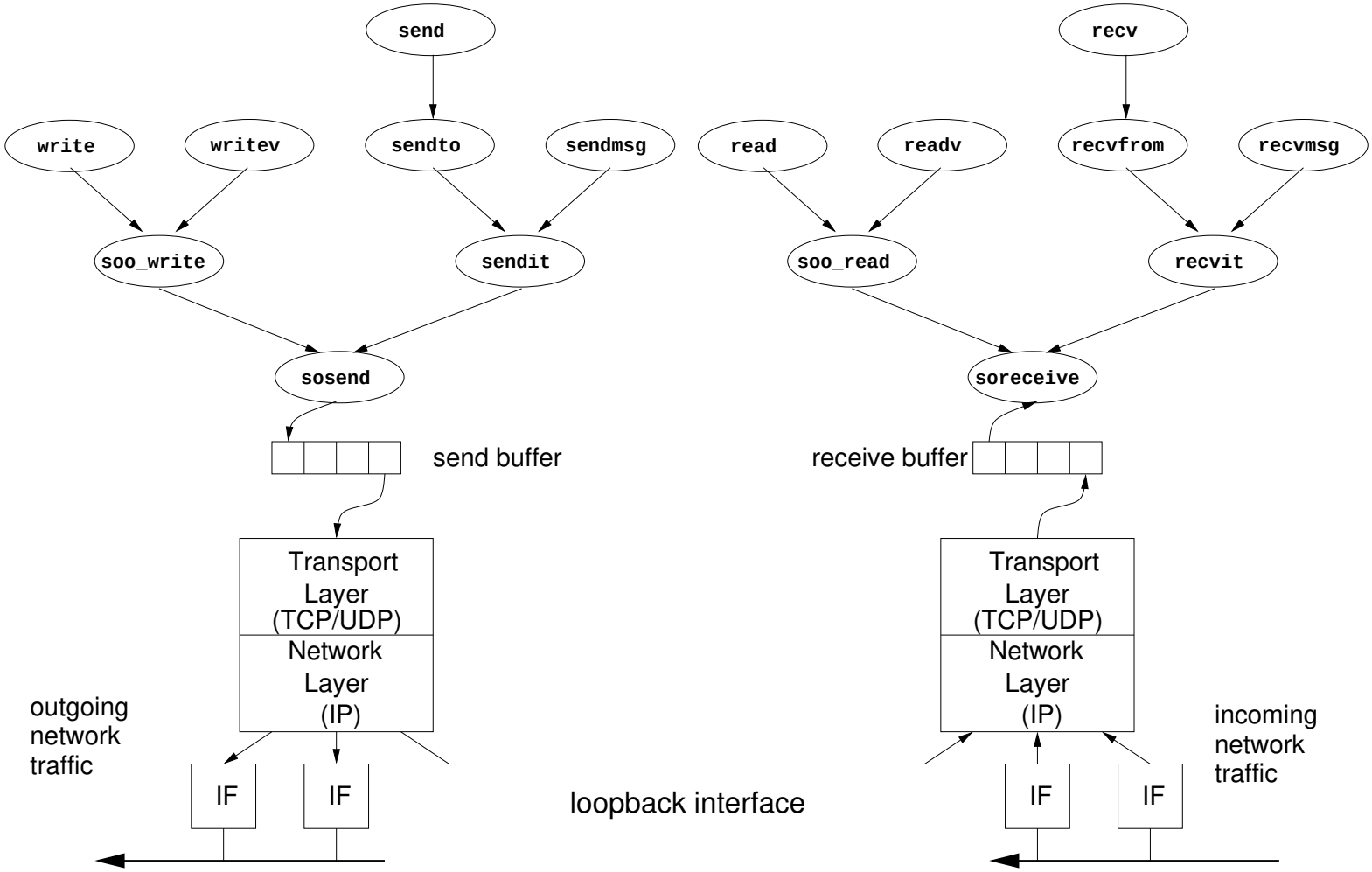

Figure 5.3. FreeBSD kernel functions for socket I/O

The sosend function is responsible for writing data into the send buffer and invoking the proper protocol's handling function for the packet through the pr_usrreq function that is associated with the socket [65, 99]. Sosend allocates a new mbuf of type M_PKTHDR unless one was already passed down to the function. At this point we check if the calling process has a label set associated with it. If so, we allocate a new bit vector for the label set, copy the process's label set to the new label set, and set the pointer in the mbuf's packet header (either the one allocated by sosend or the one that was passed down) to the new label set. Once passed to the send buffer where the appropriate protocol's handling function fetches the packets via the pr_usrreq mechanism, the mbufs are passed down the network stack, and if the receiving endpoint is local, they will be passed through the loopback device and back up the network stack (see Figure 5.3).

In some instances a protocol in the network stack has to append headers to the packet. If there is not sufficient space at the top of the packet header mbuf for the new header, then a new mbuf is prepended to the packet as its new header. This is done via the M_PREPEND macro and utilized for example in UDP. Thus the macro was modified to switch the pointer to the label set to the new mbuf header. For some unknown reason, TCP does not utilize the M_PREPEND macro. It always uses M_GETHDR and manually prepends the new mbuf. Therefore, the tcp_output function was also modified to switch the pointer. This means that if support for other network protocols is required, one needs to make sure that special cases like this are handled properly.

The soreceive function reads data from the socket's receive buffer, and blocks if no sufficient data is available to fulfill the request from the higher level functions. When the data is received from the socket receive buffer, soreceive copies the data to the user space buffers that were supplied by the system calls. Right before copying the data, both for out-of-band and regular data, the label set of the process that invoked soreceive indirectly needs to be updated. However, the process id of that process is not available to the function. Changing the function parameters

of `soreceive` would mean having to change the entire modular socket and vnode generic operations for the "receive" type, because the `soreceive` function may be invoked through a pointer that is supplied to map to generic function calls. We therefore decided to "piggyback" the process id through an unused parameter in the invocation of `soreceive` through `soo_read` and `recvit`. Both those function have access to the calling process's id and both of them pass a null value to the fourth argument of `soreceive` (an mbuf to which to transfer the receive buffer data directly). Thus a new message flag was created, `MSG_LABELPID`, which, when set, tells `soreceive` to interpret this pointer as a pointer to a process table entry. Now when data is about to be copied whose mbuf packet headers contain a pointer to a label set, the label set of the process that invoked the reading from the socket is updated with that label set, or a new label set is created for that process.

5.4 Shared Resources: Files

Looking at the amount of system calls that govern the use of files in Section 5.1.3, one can see that files are by far the most complex among the shared resources in FreeBSD. Furthermore, files are the only kind of shared object in the system that are intended for data exchange (we do exclude shared memory from our analysis; see Section 5.1.1). Thus our proof-of-concept implementation need to deal only with files for label propagation.

FreeBSD, like many other UNIX-like operating systems, supports many different file systems. To bring many different file systems with different layouts and operation into one single framework transparent to the user space processes, FreeBSD utilizes a virtual file system layer. This ensures that standard operations that the operating system supplies can be properly mapped to the file system specific functions. The generic operations that can be performed on a file descriptor by a process are mapped in a structure `struct fileops`, which contain pointers to the appropriate low-level

operation for the underlying file system. This is done for read, write, ioctl, poll, stat, and close operations.

There are two ways to implement label propagation regarding file systems: the labels can be stored with each file, or the kernel itself keeps track of which file is associated with what labels. The first approach has the advantage that there is little computational overhead as there needs to be no explicit mapping of files to labels. Simply by accessing the file, which is done anyway during the operations, the label set may be retrieved. Plus, the labels for a file are automatically stored permanently. However, this way each individual file system needs to be modified to accommodate label propagation. This might not be possible for certain file systems. While for some file systems, such as *ext2* [16], there are unused fields (e.g., the access control pointer), which can be used to point to blocks containing labels, there is no such extra space in the Reiser file system [9], and future file systems might be similarly frugal with the space they use. When the kernel keeps track of what the labels for a file are, then there needs to be some sort of lookup data structure for the mapping. However, by keeping the mapping in the kernel, the implementation is independent of the underlying file systems, supporting any of those supported by the operating system itself: Provided a unique identifier can be assigned to each file on the system. Furthermore, a data structure that contains all the files that have label sets associated with them allows us to efficiently answer questions such as: "Which are the files that possess label X?". If labels were stored directly with the file, then all the files on the system would have to be examined, not only those that have a label set.

Because of those advantages and the fact that an implementation affects fewer subsystems of the kernel, we have decided to keep track of file label sets directly in the kernel. For this, we have introduced a global `file_labels` mapping that utilizes an *AVL tree* [1], a version of a balanced binary search tree. This gives us $O(\log n)$ lookup and insertion time, where n is the number of files with labels. Given a file identifier, the `get_file_entry` function either returns the label vector associated

with the file, or null if the file has no labels. Plus, labels can be associated with a file via the `insert_file_label` function, which takes the file identifier and a label set as parameters and creates an entry in the search tree, copying the provided label set.

Files in FreeBSD (and according to the POSIX standard [47]) are uniquely identified system-wide by the pair of device and inode numbers. This is because, inode numbers are unique only within the disk partition where they reside. Both are currently 32-bit values, so there is a hard upper limit of 64 comparisons for up to 2^{64} total items when performing operations on the binary search tree. The actual number can be expected to be much lower given that the number of disk devices (partitions) can be expected to be at most a double-digit number, and the number of files with labels to be considerably less than 2^{32}. In terms of storage requirements, each entry of the AVL tree takes up 24 bytes plus the size of the label bit vector.

The system calls that are responsible for data transfer to an from files are `read`, `readv`, `write`, `writev`. The `read` and `readv` system calls, both invoke `fo_read`, which invokes the proper low-level function for read (this is true not only for regular files, but also for sockets and pipes). Therefore this is the place to handle the label propagation. After a successful call to the low-level read function we check whether the file descriptor is actually of type `DTYPE_VNODE`. If this is the case we retrieve the device and inode information from the `vnode` parameter that was passed to `fo_read`. If an entry for the device/inode pair exists in the `file_labels` tree, then we update the calling process's label set with the one retrieved by the lookup.

Similar to the reading calls, the `fo_write` function is called by the writing system calls. After a successful low-level write operation and if the calling process has a label set associated with it, we retrieve the device and inode numbers from the vnode. If a lookup in the `file_labels` tree yields a pointer to a label set, we update that label set with that of the process. If no label set previously existed for the file, an entry is created in the tree with a copy of the process's label set.

We furthermore implemented a new system call, `get_filelabel`, that, given the device and inode number of a file, returns the success value of the operation into a supplied variable and on success the label vector into a provided buffer. The device and inode number of a file can be obtained via the *stat* command or through any of the stat-family library calls. We did not implement a system call that returns the entire contents of the `file_labels` tree.

For testing and debugging purposes we also supplied a new system call named `set_filelabel`, that takes the device and inode numbers as well as a label and sets the label vector of the file to contain the label. If the file did not have a label vector associated with it, a new one is created and bound to the file.

5.5 Results

In the following we demonstrate the effectiveness of our approach by showing how to use the implementation to solve some of the problems discussed earlier. Furthermore, we will measure performance overhead to show that an implementation of label propagation is feasible.

We have already demonstrated the feasibility of using labels in the form of process origin information for the goal of gaining information about the true source of denial-of-service attacks as well as stepping stones [11,12]. Therefore, for demonstrating the further usefulness of label propagation, we will limit ourselves to addressing the two problems of user identity and location information as described in Sections 3.2.1 and 3.2.2. Furthermore, we will simulate a server compromise of a well-known service on the system and illustrate how the impact of the compromise can be determined by using host causality origin identifiers.

5.5.1 User Influence

We shall demonstrate the effectiveness of user influence labels by using the example from Section 3.2.1. As depicted in Figure 3.1, the process controlled by User A

reads data from File 1. It then communicates via IPC with a process controlled by User B, which subsequently creates a new file with the content of what was communicated during the exchange with User A.

To associate a user identifier with the processes that are under that user's control, we bind the user ID that is assigned to the process at login time to the process as a label as well. While this may seem redundant at first, note that the user ID of a process that is recorded in the process table is subject to change during subsequent login and logout operations (e.g., via the `su` command), while a label is persistent. The `setlogin` system call was modified to add the user identifier as a label to the process's label set that invoked the `setlogin` call.

To illustrate that any labels already attached to the original file will also be propagated, we first execute the `set_filelabel` system call to bind a label to the file. Then we execute a program that opens the file, reads data from it and then opens a TCP connection to a second process with a different user id and transmits the file. The program also prints out the label set to the console before and after each operation. The second process (listening on the TCP socket) reads the data from the first process and then creates a file with the data it received. Finally, we execute a program that calls `get_filelabel` to see what the label set of the newly created file is. The output from both user sessions are shown in Figure 5.4.

The `setfilelabel` program binds the label "File XXX" to the file XXX on the system. The `sender` program opens the file specified on the command line, reads a number of bytes from it, then opens a TCP connection to port 7000 on the local host and transmits the data that was read from the file. The `receiver` program listens for TCP connections on port 7000. Once a connection is established, it reads a number of bytes from the socket and then creates a new file with the data it received. The `sender` and `receiver` program also print out their label sets before and after the operations that receive data.

```
          SESSION A                                  SESSION B

% ./setfilelabel 265476 649649 "File XXX"
                                                   % ./receiver YYY
                                                   Receiver starting with pid 162
                                                   Label set at start:
                                                   Label 0: User 1002
                                                   Listening on port 7000
% ./sender XXX
Sender process starting with pid 163
Label set:
Label 1: User 1001
Label set after reading file:
Label 1: User 1001
Label 2: File XXX
Writing data to socket
                                                   Connection established
                                                   Label set after socket read:
                                                   Label 0: User 1002
                                                   Label 1: User 1001
                                                   Label 2: File XXX
                                                   Writing to file YYY

                                                   % ./filelabels YYY
                                                   File YYY has labels:
                                                   Label 0: User 1002
                                                   Label 1: User 1001
                                                   Label 2: File XXX
```

Figure 5.4. Output from the user influence test from both user sessions

Initially the two processes have only the user id label bound to them.[3] After reading the file, Process A now also contains the label "File XXX". After receiving data from the network socket from Process A, Process B has three labels associated with it: "User 1002", its original label, as well as labels "User 1001" and "File XXX". After Process B creates and writes data to file YYY, the `filelabels` program reveals that the label set of file YYY also contains those three labels. An investigator examining file YYY now can determine that both users 1001 and 1002 could have played a role in the current state of the file, plus that there is a possibility that the contents of file XXX might also have been an influence. We can further conclude that no other users could have been responsible in the creation or modification of the file YYY.

5.5.2 Location Information

To show that the label propagation also works well with non-custom programs, we also tested the location information case study as described in Section 3.2.2. Here, we assign origin information to a file containing C-source code, a library used in the code, the `gcc` compiler, and the current session (see Figure 3.2). We do this manually through programs calling `set_filelabels` and `addlabel` and then compile the program. Then we use the program from the previous example to print out the label set of the newly created file (see Figure 5.5).

As in the example, we bind a label called "OS-CDRom" to "/usr/bin/gcc", "Console" to the "sender.c" file, "Website" to the file "printutils.c", which we compile in directly as opposed to implementing a library from it first; the result is the same. Plus, we bind a label "192.168.0.1" to the shell process. We then compile the "sender.c" program. A `filelabels` lookup on the file "sender" now shows that the file is associated with all of the labels of the entities that played a role in its creation.

[3]For better readability we actually use a string 'User nnnn' as a label as opposed to a 2-byte label containing only the id itself.

```
% ./setfilelabel sender.c Console
% ./setfilelabel /usr/bin/gcc OS-CDRom
% ./setfilelabel printutils.c Website
% ./setproclabel -1 192.168.0.1

% gcc -o sender sender.c printutils.c

% ./filelabels sender
File sender has labels:
Label 0: Console
Label 1: OS-CDRom
Label 2: Website
Label 3: 192.168.0.1
```

Figure 5.5. Output from the location information case study

5.5.3 Remote System Compromise

A system is *compromised*, when an unauthorized user has gained control over the system. This is typically done by exploiting a vulnerability of the system to receive access to the system and permissions to perform certain tasks. In many cases, the compromise occurs through the exploit of a vulnerability of one of the system's network services. Our label propagation mechanism does not differentiate between authorized and unauthorized accesses and operations. If we are able to bind a location label to the process accepting the network traffic, then those labels are propagated for both legitimate as well as malevolent uses. If a real compromise occurs from a remote location, all processes and files affected by it will be labeled with that location label.

To simulate a system compromise we will run a program that accepts network connections and supplies a shell. This is the basic functionality of a backdoor program, but could also be the result of the compromise of a well-known server daemon process (be it `httpd` running as the http user, or even `sshd` running as the root user). The nature of the compromise (e.g. buffer overflow or script vulnerabilities) is not important, the end result is the same: a remote attacker has access to a shell with

the privileges of the daemon process that was compromised. Thus it is sufficient to test the label propagation with a "normal" remote login via `ssh`.

For our labeling approach to capture the entry point of the intrusion, we need to associate network-location labels to those processes receiving data from the network. For this, we have modified the `accept` system call to bind a label to the process invoking `accept` whenever a connection is successfully accepted. For demonstration purposes, we only use the foreign IP address as a label. For a more complete network identifier, the 4-tuple of foreign IP address and port as well as the protocol and local port information can be used. For UDP, a similar addition can be made to the `recv` and `recvfrom` system calls.

Figure 5.6 shows an `ssh` session with location labels enabled. The process running the `proclabels` program is clearly marked with the location label[4], which means that the process running the shell also carries the label. Furthermore, if we create new files on the system or modify existing ones, the label is propagated.

```
florian@schlaraffenland:~> ssh morpheus-8
Password:
Last login: Tue May 17 14:09:40 2005 from schlaraffenland
Welcome to FreeBSD!
%./proclabels
Process ID: 74778
Label 0: 128.10.243.68
%echo test > testfile
%./filelabel testfile
File testfile has device 0x40d04 and inode 651811
Syscall result: 0. Ret: 1
File testfile has the following labels:
Global pos: 0 data: 128.10.243.68
%
```

Figure 5.6. An `ssh` session with location labels

The approach we chose of binding the label at the time of `accept` has the disadvantage that the common network server architecture as described by Stevens [98]

[4]Again, we bind a string containing the IP address as a label for the purpose of better readability

has the server accept a connection and then fork a child process, which inherits all the connections from its parents and does the actual handling of that particular session, while the parent process goes back to the listening state. This means that over time the server process will accumulate all the labels of the past network connections and pass those labels on to its children. This can be avoided if we supply a new system call that accepts the connection, automatically forks a child process, and only then binds the label to the child. An alternative location to bind the network-location label is the interface on which the data is received. If the label is generated there and then associated with the mbufs that make up the network packet, then our socket implementation as described in Section 5.3 will automatically update the labels to exactly the processes that receive the data. The disadvantage of this is that every time data enters the system from the network we need to make a lookup whether the network-location label is already in the system or not, which may slow down performance.

5.5.4 Performance Overhead

In this section we will describe the results of performance overhead measurements we performed for the proof-of-concept implementation. We ran our experiments on a Sun SunFire V60x with an Intel 2.8 GHz Xeon processor, 512MB RAM, and a 36GB SCSI hard drive. As a baseline we use the generic FreeBSD 4.12 kernel that comes with the regular installation of the operating system. We then ran the performance tests on the same hardware booting the modified version of the kernel.

To measure the overall performance, we utilize the LMBench [67] benchmark suite. LMBench is a set of small micro-benchmarks, which measure system latency and bandwidth of data movement among the processor and memory, network, file system, and disk. LMBench is a widely used benchmark suite used to profile many hardware and software systems, providing more accurate results compared to other benchmarks in many cases [68]. We have broken down the tests into four categories:

1. Processor and process tests. These are tests that measure the time it takes a process to perform certain tasks. These include a basic system call (null call), the installing of a signal handler (sig inst.), the signal handler overhead (sig hand.), the time to fork a new process (fork), the time to execute a simple program (exec), and the time to execute the '/bin/sh' program (sh). Of particular interest to us are the times for the fork and the executions as these are directly affected by our modifications. The fork mechanism takes care of the label propagation by inheritance and the execute test further makes some read operations to access the specified programs.

2. File system tests. The file system tests consist of several simple tests to measure the execution time of the read system call for a file (read), the write system call (write), performing a stat operation on a file (stat), performing an fstat operation (fstat), the opening and closing of a null file (open/close), as well as the select operation on 500 file descriptors (select). These tests use the file system cache, not the actual time it takes for the disk operation as there are too many unknown factors to consider to generate reproducible results for those. Of particular interest are the measurements of the read and write operation as they are directly affected by our modifications.

3. Network latency tests. These tests determine the time it takes for network messages to propagate. Short control messages are sent back and forth between processes and the round trip time is measured. This is done for pipes as well as sockets of types AF_UNIX, TCP, and UDP, all within the local host. All of these tests are relevant as they all measure the performance of the socket subsystem that we modified.

4. I/O bandwidth tests. The bandwidth test measure the data throughput on the system. All of these tests write a certain amount of data to a file or a communication channel in transfers of constant size. The file write test writes 8MB of data in 64K buffers. For the IPC bandwidth tests two processes

are created that transfer the data between them. Pipes transfer 50M in 64K chunks, and the TCP and AF_UNIX sockets transfer 50M in 1M chunks. All of these tests are relevant as they directly measure the impact of our modifications to the I/O throughput of the system.

These tests were run on the regular FreeBSD 4.12 kernel (`FreeBSD`) as well as on three variants of our modified kernel. The first version (`Label-0`) is the kernel with all the modifications for label propagation in place but without any labels actually present in the system. This captures the overhead of a system with label propagation in place without labels present, but also gives a measure for how the socket subsystem is affected if the processes involved have no label sets associated with them. The second version (`Label-s`), in addition to the `Label-0` version also has a label associated with the shell process that invokes the LMBench test suite. It is sufficient to only bind one label to the process as the entire label vector is propagated once a process is marked to have labels. Furthermore, a small set of files is marked with labels as well. This number is initially one, but as files are created during the performance test, the number will increase slightly. This version will give a measure for the socket subsystem overhead as well as the file system performance for a small set of files with labels. The last version (`Label-l`), in addition to the `Label-s` version, has a large number of files in the system. This will give a measure for the file system overhead for a large number of files.

To label the files for the `Label-l` set, we used a program that utilizes the `set_filelabel` system call in a `for`-loop for different inode numbers to generate a set of 100,000 files – 50,000 for each of the two disk partitions on the system – that have labels associated with them. The maximum number of labels in the system was set to 1024, which means that the label vectors are 128 bytes long. The number was chosen to be significantly higher than the number of users on the system (around 20) so that we gain a measure for a large fixed-size label set implementation. For a fixed-label system this is a large number – for user identification labels we would expect the size to normally lie anywhere between 10 and 100 – so the overhead mea-

surements are pessimistic. Each test was performed 200 times for each kernel version, and the results are shown in Tables 5.1 through 5.4. Detailed measurement results including mean, standard deviation, minimum, and maximum for each version are included in Appendix 6.

Table 5.1
Processor and process tests – times in μs

Kernel	Null call	sig inst.	sig hand.	fork	exec	sh
FreeBSD	0.4504	0.6643	1.3467	129.0783	594.1023	1176.8880
Label-0	0.4499	0.6666	1.3404	128.3902	596.1760	1173.6708
Label-s	0.4498	0.6650	1.3364	129.1161	602.4783	1185.3990
Label-l	0.4501	0.6658	1.3393	129.1359	604.5881	1187.9500

Table 5.1 shows that our experiments show only a slight increase in execution time for the process tests. System calls and signal handling is not affected by our modifications. The `fork` system call has to copy an extra 132 bytes, 128 for the label vector and 4 for the label flag. The measured overhead for this is less than 1%. The times for the `exec` and `/bin/sh` tests are influenced by the file system performance. The observed overhead here lies between 0.3% when no labels are present to 1.7% for the large set of labeled files (both for the `exec` test).

Table 5.2
File system tests – times in μs

Kernel	read	write	stat	fstat	open/close	select
FreeBSD	1.0437	0.9836	2.2547	0.6598	3.3546	19.8101
Label-0	1.1235	0.9839	2.2744	0.6649	3.4365	20.0437
Label-s	1.2139	1.3903	2.2739	0.6662	3.4457	19.9333
Label-l	1.3297	1.4934	2.3362	0.6621	3.4701	19.9283

Table 5.2 shows the execution times for the file system calls. As expected, there is no noticeable difference between the `FreeBSD` and the `Label-0` kernels. We measured a an average of 0.0798 μs (7.6% overhead) for the `read` test, which can be attributed to the lookup that is performed for the file. The `stat`, `fstat`, `open/close`, and `select` test did not have any noticeable differences between the kernel versions, as expected. With labels present on the system the `read` test now showed a difference of 0.1702 μs (16.3%) for the small label set and a difference of 0.2860 μs (27.4%) for the large label set. When comparing the `Label-s` and `Label-l` versions, the overhead for the 100,000 labeled files (a binary search tree depth of about 16) is 0.1831 μs (15.1%). For the `write` test, there is no noticeable difference between the `FreeBSD` and the `Label-0` versions. This was to be expected as no lookups are performed and no labels need to be inserted into the binary tree. For the remaining two versions, the `write` test was measured with an overhead of 0.4067 μs (41.3%) for `Label-s` and 0.5098 μs (51.9%) for `Label-l`, respectively. This is because first a lookup is performed to see if a label set is already present in the search tree, and if not one needs to be allocated and inserted. When comparing the small and the large label set versions, we observed an overhead of 0.1031 μs (7.4%). Note that all the observed differences lie in the tenths of microsecond range and were performed on the file system cache for a small read and write buffer. This makes it difficult to predict what the effect on a "normal" system is. The results do, however, reflect the worst case scenario, where a program performing rapid and short read or write operations could be slowed down noticeably. For the general case, though, we do not expect this to occur.

The local networking latency times are shown in Table 5.3. The measured overhead in all instances is small. There is no noticeable difference between the `FreeBSD` and `Label-0` versions, as expected. For `Label-s` the we measured an average overhead of 8.2% for AF_UNIX sockets, 6.1% for UDP, and 6.4% for TCP. `Label-l` has an observed overhead of 8.4% for AF_UNIX, 5.9% for UDP, and 6.5% for TCP. The fact that the measurements for the pipe latency do not vary noticeably between the

Table 5.3
Network latency tests – times in μs

Kernel	pipe	AF_UNIX	UDP	TCP
FreeBSD	11.2702	12.5225	17.0421	17.8300
Label-0	11.4273	12.6802	17.1144	18.0908
Label-s	11.4340	13.5479	18.0801	18.9715
Label-l	11.4582	13.5722	18.0479	18.9950

different versions leads us to believe that pipes on FreeBSD do not utilize the socket subsystem despite such claims [65].

Table 5.4
I/O bandwidth tests – in MB/s

Kernel	file write [5]	TCP	AF_UNIX	pipe
FreeBSD	63576.9450	377.8209	617.4526	1845.4127
Label-0	63571.9400	385.6075	612.9818	1842.9119
Label-s	63630.4250	365.5163	544.0799	1835.8787
Label-l	63595.2050	366.4497	543.9735	1833.3753

Table 5.4 shows the I/O bandwidth measurement results. Surprisingly, despite the rather large overhead for the `write` system call test the file write performance does not differ noticeably from that of the original FreeBSD kernel, even outperforming it slightly during our measurements for `Label-s` and `Label-l`. Between the `FreeBSD` and `Label-0` versions we did not observe any noticeable difference. The fact that there is a 2.0% improvement in our measurements for `Label-0` in the TCP throughput, however, suggests that there is a large variance in the network bandwidth tests as can also be seen in the standard deviation values in Appendix 6.

[5] file write is in KB/s

`Label-s` shows a deprecation of 3.3% in the TCP throughput and a deprecation of 11.9% for the AF_UNIX bandwidth. For `Label-l` we measured a TCP bandwidth 3.0% worse than `FreeBSD` and an AF_UNIX deprecation of 11.9%. Again, pipes are not noticeably affected by process labeling, which we take as further evidence that they do not use the socket subsystem.

The overall performance overhead that we observed in our experiments for process labeling is promising. The overhead for the individual tests ranges from smaller than 1% to no more than 10% in many cases. The `read` and `write` tests carry a higher overhead of up to 51.9%, but the file write bandwidth was not affected at all. For programs that frequently write small amounts of data into the file system cache this will be a problem, but overall we do not expect the file system slowdown to be severe, especially when factoring in the time to write the cache to the disk.

The network measurements were performed for a label vector size of 128 bytes (1024 labels), and we expect the overhead to be smaller for smaller vector sizes. This may well be the case for certain fixed-label set size applications of a process labeling approach, such as user influence, where we expect the label vectors to be smaller than 10 bytes for most systems. If a fully dynamic approach with a potentially unbounded label set size is desired, however, performance is likely to suffer more.

6 CONCLUSIONS

In this dissertation we have determined that there is a lack of audit data on current computing systems. As a result of this certain relations between events cannot be established or only insufficiently so through cost-intensive and manual analysis. We further presented a label-propagation model, that lets the system propagate arbitrary labels of information among its principals and objects based on how information flows within the system. We have demonstrated how those labels can be used to gain some of the desired information regarding the causal effects of events. Our proof-of-concept implementation shows the feasibility of incorporating label propagation for a production-type operating system with little to no overhead.

Of the questions of who did what, where, when, how and why, only a few can be answered from the information collected by current computing systems. This is in part because of space constraints, but also, as we have discussed, because some of the desired information is impossible to obtain on systems that run arbitrary programs.

Forensics and security were not design objectives for the most commonly used file systems. Some of our desired information could be obtained by, for example, recording more information on one-time events such as the creation of a file. The "create" timestamp, the user who created a file, and the user agent path could be recorded in a fixed amount of space. Other information such as detailed file modification or access information are unbounded in their space requirements and therefore recording them might not be suitable in every situation. Moreover, information such as user influence or location of file operations are generally undecidable, which means that the information is not recorded and heuristics must be used, or only verified programs are allowed to be executed on the system.

Not every system is suited to collect all of the desired information we have discussed in Chapter 3. For a typical home computer none of the extra information may be worth the space requirements or restrictions that would result from recording it. However, when factors such as due diligence, protecting critical information, or being able to quickly determine what happened on a computing system are important, all of the extra information discussed may play an important role. Current systems do not offer the ability to record much of the desired information even if one wanted to record it.

Some of the information that we classified in Section 3.1 outside Category 1 may immediately be recorded by existing systems. A file creation time that cannot be modified anymore should be present on any file system. Recording the user id of the process performing file creation, access, or modification is also a simple inexpensive addition of more valuable data. In general, however, it will depend on the kind of system as to which of the information we have discussed in this paper should actually be recorded and how much of it. Recording everything we have mentioned on every system is not realistic. However, policies in some organizations may require recording a large portion of it. These may range from high-security computing systems, where even the access of certain files should be documented in its most complete form (who, where, when, how?), to home computers where maybe only the question of where certain files came from matters.

In the future of system and file system design, forensics and security will play a more important role. For some of the information we discuss in Chapter 3 we do not offer explicit solutions on how to implement obtaining and storing it. This is part of future research in the field of digital forensics. Nor do we mandate what kind and how much information should be recorded. This will depend on individual systems and the requirements they have in regard to forensics. We do, however, hold the opinion that if desired, it should be able to record such information.

In Chapter 4 we have presented a general model for label propagation based on information flow among principals. Labels may be used to propagate meta information about the principals as they communicate with each other. The case studies demonstrate in which manner labels can be utilized to generate audit data for digital forensics or intrusion detection, or data that can be used for access control. The model does not attempt to control information flow, it merely adds new information in the form of labels to those flows. Because we use a heuristic to determine causality there will be a number of false positives. This is acceptable because having to infer which information may have caused the output of a principal from a (small) set of prospective culprits is better than having no indication at all what caused the output (any principal could have).

However, labels are mostly meaningful when their presence at principals and objects is limited. For example, if labels are user identifiers, and a User A's actions were influenced by another malicious User B, then A's process is labeled with both user ids. If only those two labels are present and there is malicious behavior by A the list of potential culprits may be narrowed down to A and B. If, for whatever reason, A is labeled with all possible user identifiers on the system, the labels have become worthless because no information may be gained from them. For this reason, a production system that implements our model should make sure that labels are only propagated when necessary and that a principal cannot add new labels to its label set frivolously to obscure its label set. Our proof-of-concept implementation in Chapter 5 adheres to these principles. Also, it might be necessary to develop new, label-friendly programming paradigms. These could include the concept of an *execution context* for processes, where a new context is created for a specific task and then discarded without the process gaining information about what occurred while the context was active. This way, labels could be bound to the context and disappear with it instead of remaining with the process even though they are not relevant for any new tasks.

Currently, network services on a system are offered through special daemon processes that accept the network connections and then spawn off a child process that handles the rest of the communication. If location information were added as a label to those processes each time a connection is accepted, soon the daemon process as well as all its subsequent child processes will carry all those labels unnecessarily. To avoid this situation, the server paradigm needs to be changed. The above situation could be solved by some sort of combined `accept-fork` new system call, that accepts the connection, forks a child process, and then binds the label only to the child. Alternatively, labels could be generated directly at the network interface to label network packets. The advantage of the former is that labels are created less frequently, reducing the cost of lookup for existing labels. The advantage of the latter is that only processes that actually read the network packets are updated with the new labels. This way, the existing paradigm need not be modified at all.

In Section 4.3 we discuss space management models. Whenever a loose space management model is used, labels will be lost. Depending on which labels are deleted when and what type of audit recording of labels takes place, a malicious principal may attempt *label washing*: getting rid of (some of) his own labels by acquiring more labels or making other principals take on more labels. While this can not be prevented, intensive auditing and techniques from intrusion detection for abnormal system behavior could be utilized to identify those attempts.

If strict space management models are in place, a malicious principal may perform denial-of-service attacks on the system by exhausting the resources for storing the labels. If a global label pool exists all principals will be denied further operations. If label space is localized, then a principal may still perform a denial-of-service attack on others if he can "trick" other principals to acquire a large label set. This could be done by "infecting" an object that is accessed by many principals with a large label set.

For these reasons any framework that utilizes the model should make sure that labels only be propagated and created when absolutely necessary. Well-behaving

principals from our case studies should not carry many labels in their label sets. In a computing system, opportunities need to be created for programmers who develop label-friendly programs. If the framework is used to enforce policy, then access control mechanisms need to be implemented in conjunction with the label propagation. This can further reduce the probability that a well-behaving principal acquires unnecessary labels. For example, if a user by default does not have any access to other users' data, he will only pick up another user label if that user explicitly grants such access and the data is actually accessed. If the framework is used to monitor policy violations, some sort of alert mechanism, such as in intrusion detection, could be utilized to identify principals and objects whose label sets satisfy certain parameters. For this, some sort of human control mechanism that allows an investigation of the label sets and also removal of labels after the investigation, if necessary, may be useful to keep the overall label sets small. For example, after a remote compromise has been detected and contained, the files that were affected by it should be cleared of the labels obtained during the compromise and its consequences. Naturally, such a control mechanism should lie outside the normal system capabilities, such as a special run-level with an operator sitting at the console.

Given this, it becomes clear that a computing system for the average home user is not the target platform for our model. Limitations imposed on the system would be too restrictive to justify the benefits. For example, if peer-to-peer file sharing software were executed for a host causality system, a single file, once completely downloaded, may already have hundreds of labels bound to it. If the user then accesses the file via a shell command, the process running the shell inherits the labels and so will all the subsequent processes that are executed from that shell and the files that are affected by them.

In some cases it might be sufficient to address only true data exchange channels for our label propagation. This means that storage channels through shared objects are not considered. This can easily be done by not modifying the label sets for the `open` and `close` operations of our model. This is the case for our proof-of-concept

implementation. Such a framework will capture lesser forms of information flow but it is still useful as we expect label sets to stay smaller in general. Malicious principals could now utilize these storage channels to bypass the model, but those channels are low-bandwidth and for certain cases raising the bar in such a manner might be sufficient.

In Chapter 5 we have discussed how to implement the propagation model for a production-type operating system, namely FreeBSD. We have chosen an operating system kernel for our implementation as the kernel's system call interface to the user space closely resemble the operations described in our model. For label propagation to be secure in this scenario, we have to make the assumption that the kernel is trusted and untampered. Securing the kernel is certainly outside the scope of the work we presented in this dissertation but project such as LIDS [115] can be used for such measures. An operating system's kernel is not the only place where labeling of principals may be implemented. Integrating label propagation into a virtual machine will enhance the trustworthiness of the labels but comes at the cost of utilizing the virtual machine. This approach is already being realized [51]. But label propagation may also be utilized in user applications or a middle-ware layer. This may be desired when the label granularity of system objects is too coarse to be of value for certain applications. Consider a database management system where the databases are stored in large files. Associating labels about database transactions with the files themselves might result in too many labels being associated with too many entities of the database system. If instead the database management system implemented label propagation through the interface it provides to the clients, labels could be used to analyze or enforce information flow within a database.

The data structures we have used for our proof-of-concept implementation were designed for a fixed-size label set. Furthermore, we decided to keep the accounting for file object label within kernel memory as opposed to implement label support for a file system. Overall, the optimal data structures for label propagation will

depend closely on the types of labels that need to be supported. Our overhead measurements are conservative. For a fixed-size label set user influence approach we do not expect that 1024 labels are needed on the system. The disk I/O measurements were conducted on the cached portion of the file system, meaning that when factoring in time for actual disk access, the overhead we measured for the `read` and `write` calls is negligible, as well. Despite the conservative measurements, the performance results are encouraging. We have measured an overhead between 0% and roughly 10% for all of our tests except for the `read` and `write` system call times. Most of the overhead measured here we attribute to the managing of the binary search tree. Thus, if label support is added directly to a file system, we expect this overhead to be reduced, significantly, as well.

An interesting area of immediate future research in label propagation is therefore the development of suitable data structures for different kinds of labels. This includes how to manage all the labels known globally to the system, but also how each process and object is associated with a label set. This research needs also be concerned with how to permanently store labels on the system, as in our implementation all labels are lost when the system restarts. This is no trivial problem, as labels preserved on long term storage may need to be incorporated into a running system at a different time than the system's start up routine.

The examples we provide in Sections 5.5.1, 5.5.2, and 5.5.3 demonstrate the effectiveness of our approach. We had already demonstrated the value of labeling for network traceback in earlier work [11,12]. When examining processes and objects on the system we now can make statements as to whether or not they were influenced by external factors such as different users of remote locations. We are not able to assert for sure that if a label is found there was an actual influence, but we have proven in Chapter 4 that the lack of such labels means that no communication took place through those channels of the system that support label propagation. Keeping the label sets of the principals and objects of a system small is therefore an important

goal, which further needs to be addressed by future research. For this, a combination of label propagation and access control mechanisms could be devised.

Limitations

The work we present in this document addresses many important aspects of label propagation, the model's properties, space concerns, and correctness as well as usability. However, there are limitations that we do not specifically address but may be of concern. Some of the limitations we discuss in the following apply to the implementation of the model only, whereas others are also true for the theoretical models.

The label data is not encrypted and potentially can be seen by any user on the system. Depending on the nature of the label, this may lead to privacy concerns. One simple measure could be to limit access to the labels only to a limited set of users (e.g. via the `getlabel` system call), for example the system administrators. However, this may not be sufficient in all scenarios. One can imagine labels whose propagation is desired but where only the originator of the label should have the option to disclose the data. Using a random token instead of the actual label data may solve the problem in some cases, but once the mapping of token to data is revealed, all users once again have access to the label data. However, because the only requirement we impose on the label update function is that the label needs to be preserved (or at least one should be able to deduce the previous label from the current one), one can imagine the use of cryptographic functions that generate a unique identifier for the result of each update operation, where the original label data may be decrypted with a key. However, we would expect that this functionality would come with a performance cost.

When a principal or an object accumulates many labels, the usefulness of the labels' presence degrades. When a subject possesses all possible labels, an investigator has gained no extra information compared to a system that does not utilize label

propagation. This kind of *label obfuscation* may be attempted deliberately by a malicious principal to obfuscate his tracks. In many existing systems there exist globally writable objects that are read by many principals on the system. In FreeBSD there is the `syslog` facility to which all processes may report and is further accessed by many processes on the system. Services such as `cron` and `at` also may pick up labels from all the processes that utilize them. These globally shared objects are a problem when looking to avoid label obfuscation. One could try to eliminate the global sharing by providing mini-services that are valid only for one given process. While this may be acceptable for `cron` and `at`, a per-process `syslog` service would yield scattered log files, with much less usefulness than a single log file would have.

The above discussion about how to deal with label obfuscation and globally shared resources on an existing system shows that label propagation will not always work smoothly together with existing systems. Our proof-of-concept implementation shows that label propagation can work with a system such as FreeBSD, but some of the limitations will be difficult if not impossible to remove. The intent of our work lies primarily in the introduction of the label propagation paradigm. While legacy systems may be adapted for label propagation, we feel that systems designed with label propagation in mind will benefit the most from the concepts discussed here. But not only systems should be designed with label propagation in mind. *label-friendly* programming techniques could be used to keep the label set of a process as small as possible. If all "normal" programs adhere to this principle, misbehaving processes could be better identified and contained.

The implementation of label propagation we present in this dissertation does not support separate label sets for threads of a multi-threaded process. As threads share memory space among them, the system is not able to monitor information flow between threads. Given that some programs use a large number of threads to perform tasks, in some cases the granularity of labels for the entire process may be too coarse, meaning that the process will accumulate all of its threads' labels. However, as with shared memory, barring special hardware that can monitor those

information transfers, implementing label propagation on the operating system level prohibits label propagation on the thread-level.

Labels are not stored permanently in our proof-of-concept implementation. Once the system reboots, all labels are lost. Incorporating label support directly into a file system, as mentioned above, is one necessary step to achieve a permanent label retention on a system. In addition to that the labels need to be read from storage at start-up and be written to storage at shutdown. This introduces the danger that labels may not be stored when the system is not shut down properly (i.e. it crashes). Storing the label periodically can reduce the amount of labels that is potentially lost, but the threat of losing labels remains. Also, if labels sets are stored in label-vectors as in our proof-of-concept implementation, the global label table needs to keep labels always in the same order. This is because not all labels are necessarily introduced to the system at start-up. Some file systems with labels may be mounted at a later time at which point the label-vector bits need to point to the correct entries in the label table. Furthermore, if a file system from a different system is mounted and also has labels associated with it, the new labels need to be incorporated into the current system or be discarded. In the former case, the label data has to be stored on the file system device, as well.

Future Work

In this dissertation we have demonstrated that it is possible to add significantly useful audit information to a system with little computational overhead by binding labels that convey information such as user identity or location information to principals on the system and propagate those based on how information flows between principals and objects. The above discussion shows that there is still research to be done in the area of run-time label propagation and with this dissertation we only lay the foundation for it. As part of the immediate future work, we see the following:

- Research if and how existing programs and programming paradigms fit in with a label propagation framework in regard to the amount of labels they accumulate and the implications for access control that has in strict models.
- Determine how to integrate label propagation into access control and intrusion detection mechanisms.
- Devise a configurable space management mechanism for labels that allows to associate subjects with resource groups and enforce label label constraints as discussed in Section 4.3.
- Explore new uses for labels outside of those discussed in this dissertation.
- Identify scenarios where label propagation and its (possible) limitations award the most benefits to justify its use.

LIST OF REFERENCES

LIST OF REFERENCES

[1] G.M. Adelson-Velskii and E.M. Landis. An Algorithm for the Organization of Information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.

[2] M. Adler. Tradeoffs in Probabilistic Packet Marking for IP Traceback. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, 2002.

[3] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., April 1980.

[4] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. *MITRE Report MTR 2547 v2*, 1973.

[5] S. Bellovin, Marcus Leech, and Tom Taylor. ICMP Traceback Messages. Technical report, IETF Internet Draft, February 2003. Work in progress.

[6] S. M. Bellovin. Security Problems in the TCP-IP Protocol Suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[7] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, 1977.

[8] D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[9] Florian Buchholz. The Structure of the Reiser File System. `http://www.cerias.purdue.edu/homes/florian/reiser/reiserfs.php`.

[10] Florian Buchholz, Thomas E. Daniels, Benjamin Kuperman, and Clay Shields. Packet Tracker Final Report. Technical Report 2000-23, Center for Education and Research in Information Assurance and Security (CERIAS), West Lafayette, IN, 47901, 2000.

[11] Florian Buchholz and Clay Shields. Providing Process Origin Information to Aid in Network Traceback. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, July 2002. CERIAS TR 2002-22.

[12] Florian Buchholz and Clay Shields. Providing Process Origin Information to Aid in Computer Forensic Investigations. *Journal of Computer Security*, 12(5):753–776, September 2004.

[13] Florian Buchholz and Eugene H. Spafford. On the Role of File System Metadata in Digital Forensics. *Journal of Digital Investigation*, 1(4):298–309, December 2004.

[14] Florian Buchholz and Eugene H. Spafford. A Model for Label Propagation Based on Causality. *under submission*, 2005.

[15] Michael A. Caloyannides. *Computer Forensics and Privacy*. Artech House, Norwood, MA, 2001.

[16] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Frank B. Brokken et al., editor, *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[17] B. Carrier and C. Shields. A Recursive Session Token Protocol for Use in Computer Forensics and TCP Traceback. In *Proceedings of the IEEE Infocomm 2002*, June 2002.

[18] Brian D. Carrier and Eugene H. Spafford. Defining Event Reconstruction of Digital Crime Scenes. *Journal of Forensic Sciences*, 49(6), 11 2004. CERIAS TR 2004-37.

[19] Eoghan Casey, editor. *Handbook of Computer Crime Investigation*. Academic Press, San Diego, CA, 2002.

[20] Eoghan Casey. *Digital Evidence and Computer Crime*. Academic Press, San Diego, CA, second edition, 2004.

[21] Characterizing and Tracing Packet Floods Using Cisco Routers. `http://www.cisco.com/warp/public/707/22.html`.

[22] Franklin Clark and Ken Diliberto. *Investigating Computer Crime*. CRC Press, Boca Raton, FL, 1996.

[23] R.C. Daley and P.G. Neumann. A General-Purpose File System For Secondary Storage. In *Fall Joint Computer Conference*, 1965.

[24] Thomas E. Daniels. *Reference Models for the Concealment and Observation of Origin Identity in Store-and-Forward Networks*. PhD thesis, Purdue University, West Lafayette, IN, 12 2002. CERIAS TR 2002-31.

[25] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to ip traceback. *"ACM Transactions on Information and System Security (TISSEC)"*, 5(2):119–137, May 2002.

[26] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Technical Report RFC 2460, Internet Society, December 1998. `ftp://ftp.isi.edu/in-notes/rfc2460.txt`.

[27] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[28] Dorothy E. Denning. Secure Personal Computing in an Insecure Network. *Communications of the ACM*, 22(8):476–482, 1979.

[29] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[30] Dorothy E. Denning and Peter F. MacDoran. Location-based Authentication: Grounding Cyberspace for Better Security. In *Internet Besieged: Countering Cyberspace Scofflaws*, pages 167–174. ACM Press/Addison-Wesley Publishing Co., 1998.

[31] Trusted Computer System Evaluation Criteria. Technical report, Department of Defense, December 1985. 5200.28-STD.

[32] T. W. Doeppner, P. N. Klein, and A. Koyfman. Using Router Stamping to Identify the Source of IP Packets. In *7th ACM Conference on Computer and Communications Security*, pages 184–189, Athens, Greece, November 2000.

[33] D. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *Proceedings of the 2002 Recent Advances in Intrusion Detection (RAID)*, 2002.

[34] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[35] James P. Early. An Embedded Sensor for Monitoring File Integrity. Technical report, CERIAS, January 2002. CERIAS TR 2001-41.

[36] Dan Farmer and Wietse Venema. *Forensic Discovery.* Addison Wesley, Boston, MA, 2004.

[37] J.S. Fenton. Memoryless Subsystems. *The Computer Journal*, 17(2), 1974.

[38] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing. Technical Report RFC 2827, Internet Society, May 2000.

[39] J. Chapman Flack and Mikhail Atallah. A Toolkit for Modeling and Compressing Audit Data. Technical report, COAST, Purdue University, 1998. COAST TR 98-20.

[40] FreeBSD Operating System. `http://www.freebsd.org`.

[41] S. Garfinkel, G. Spafford, and A. Schwartz. *Practical Unix and Internet Security.* O'Reilly, third edition, 2003.

[42] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *2003 Symposium on Network and Distributed System Security*, February 2003.

[43] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[44] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162. ACM Press, 1987.

[45] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[46] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. *Lecture Notes in Computer Science*, 1782:180–199, 2000.

[47] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). `http://standards.ieee.org/catalog/olis/arch_posix.html`.

[48] Institute for Security Technology Studies. Law Enforcement Tools and Technologies for Investigating Cyber Attacks: A National Needs Assessment. Technical report, Dartmouth College, 2002.

[49] Institute for Security Technology Studies. Law Enforcement Tools and Technologies for Investigating Cyber Attacks: Gap Analysis Report. Technical report, Dartmouth College, February 2004.

[50] J. Ioannidis and S. M. Bellovin. Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of the 2002 Network and Distributed System Security Symposium*, San Diego, CA, February 2002.

[51] Xuxian Jiang, Dongyan Xu, and Florian Buchholz. Tracking Worm Contamination: a Process Coloring Approach. *under submission*, May 2005.

[52] R. Joshi, K. Rustan, and M. Leino. A Semantic Approach to Secure Information Flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[53] H.T. Jung, H.L. Kim, Y.M. Seo, G. Choe, S.L. Min, C.S. Kim, and K. Koh. Caller Identification System in the Internet Environment. In *UNIX Security Symposium IV Proceedings*, pages 69–78, 1993.

[54] Richard A. Kemmerer. Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[55] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Technical Report RFC 2401, Internet Society, November 1998. `ftp://ftp.isi.edu/in-notes/rfc2401.txt`.

[56] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[57] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236. ACM Press, 2003.

[58] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[59] Warren G. Kruse II and Jay G. Heiser. *Computer Forensics: Incident Response Essentials*. Addsion-Wesley, Boston, MA, 2002.

[60] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. PhD thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.

[61] B. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[62] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[63] Paul J. Leach and Rich Salz. UUIDs and GUIDs. `http://www.webdav.org/specs/draft-leach-uuids-guids-01.txt`.

[64] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. Technical report, NSA Information Assurance Research Group, February 2001.

[65] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, Boston, MA, 1996.

[66] John McLean. Proving Noninterference and Functional Correctness Using Traces. *Journal of Computer Security*, 1(1), 1992.

[67] L. McVoy and C. Staelin. LMBench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, January 1996.

[68] L.W. McVoy and S.R. Kleiman. Extent-like Performance from a Unix File System. In *USENIX Winter Conference*, pages 33–43, January 1991.

[69] Microsoft Corporation. FAT: General Overview of On-disk Format. `http://www.microsoft.com/hwdev/download/hardware/fatgen103.pdf`, 1999.

[70] G. Mohay, A. Anderson, B. Collie, O. De Vel, and R. McKemmish. *Computer and Intrusion Forensics*. Artech House, Norwood, MA, 2003.

[71] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of the 2001 USENIX Security Symposium*, Washington D.C., August 2001.

[72] R.T. Morris. A Weakness in the 4.2BSD Unix TCP-IP Software. Technical Report 17, AT&T Bell Laboratories, 1985. Computing Science Technical Report.

[73] Ira S. Moskowitz and Myong H. Kang. Covert Channels - Here to Stay? In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 235–244, Gaithersburg, MD, 1994. National Institute of Standards and Technology.

[74] The National Insistute of Justice. Electronic Crime Needs Assessment for State and Local Law Enforcement. `http://www.ojp.usdoj.gov/nij/pubs-sum/186276.htm`, April 2001.

[75] J. Palsberg and P. Ørbæk. Trust in the Lambda-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[76] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[77] K. Park and H. Lee. On the Effectiveness of Probabilistic Packet Marking for IP Traceback under Denial-of-service Attack. In *Proceedings IEEE INFOCOM 2001*, pages 338–347, April 2001.

[78] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001.

[79] C.P. Pfleeger and S.L. Pfleeger. *Security in Computing*. Prentice Hall PTR, Upper Saddle River, NJ, third edition, 2003.

[80] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[81] J. Postel. Internet Protocol. Technical Report RFC 791, Internet Society, September 1981. `ftp://ftp.isi.edu/in-notes/rfc791.txt`.

[82] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[83] Hans Reiser. Reiser File System Whitepaper. `http://www.namesys.com`.

[84] H.G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[85] D.M. Ritchie and K. Thompson. The UNIX Time-Sharing System. In *Fourth ACM Symposium on Operating System Principles*, Yorktown Heights, New York, October 1973.

[86] M.K. Rogers and K. Seigfried. The Future of Computer Forensics: A Needs Analysis Survey. *Computers & Security*, 26, 2004.

[87] G.-C. Rota. The Number of Partitions of a Set. *American Mathematical Monthly*, 71:498–504, 1964.

[88] J. Rowe. Intrusion Detection and Isolation Protocol: Automated Response to Attacks. Presentation at Recent Advances in Intrusion Detection (RAID), 1999.

[89] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[90] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding When to Forget in the Elephant File System. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[91] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, August 2000.

[92] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.

[93] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[94] A.C. Snoeren, C. Partridge, L.A. Sanchez, C.E. Jones, F. Tchakountio, and W.T. Strayer S.T. Kent. Hash-Based IP Traceback. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001.

[95] D. X. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the IEEE Infocomm 2001*, April 2001.

[96] S. Staniford-Chen and L.T. Heberlein. Holding Intruders Accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, CA, May 1995.

[97] W.R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, 1993.

[98] W.R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, Upper Saddle River, NJ, 1998.

[99] W.R. Stevens. *UNIX Network Programming: Interprocess Communications*, volume 2. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1999.

[100] R. Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[101] Sun Microsystems. SunSHIELD Basic Security Module Guide. `http://docs.sun.com/db/doc/802-5757`.

[102] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. Reprinted in *The Undecidable* (Ed. M. David). Hewlett, NY: Raven Press, 1965.

[103] A.M. Turing. Correction to: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 43:544–546, 1938.

[104] Unix System Manual Pages. Finding files: find(1).

[105] Unix System Manual Pages. make(1): GNU make utility to maintain groups of programs.

[106] Wietse Venema. TCP WRAPPER, A Tool for Network Monitoring, Access Control, and for Setting Up Booby Traps. In *Prococeedings of the 1992 USENIX Security Symposium*, September 1992.

[107] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[108] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[109] Eric W. Weisstein. Bell Number. `http://mathworld.wolfram.com/BellNumber.html`.

[110] Christian Wettergren. Runtime Information Flow Analysis and Security: Licentiate Thesis Proposal. `http://www.it.kth.se/~cwe/phd/licprop.ps`, 1996.

[111] A. Whitaker, R.S. Cox, and S.D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of USENIX OSDI 2004*, December 2004.

[112] A. Whitaker, R.S. Cox, and S.D. Gribble. Using Time Travel to Diagnose Computer Problems, September 2004.

[113] Edward Wilding. *Computer Evidence: A Forensics Investigations Handbook*. Sweet & Maxwell, London, 1997.

[114] S.F. Wu, L. Zhang, D. Massey, and A. Mankin. Intention-Driven ICMP Trace-Back. IETF Internet draft, February 2001. Work in progress.

[115] Huagang Xie and Philippe Biondi. Linux Intrusion Detection System. `http://www.lids.org`.

[116] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2003.

[117] K. Yoda and H. Etoh. Finding a Connection Chain for Tracing Intruders. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, October 2000.

[118] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, 2001. CERIAS TR 2001-42.

[119] Shu Zhang and Partha Dasgupta. Denying Denial of Service Attacks: A Router Based Solution. In *The 2003 International Conference on Internet Computing*, pages 301–307, June 2003.

[120] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

APPENDIX

Appendix A: Detailed Performance Results

Table 1
`FreeBSD` kernel results

Test	Mean	Std. dev.	Min	Max
Simple syscall	0.4504	0.0030	0.4453	0.4625
Sig inst.	0.6643	0.0068	0.6590	0.7080
Sig hand.	1.3467	0.0574	1.2860	1.6540
Process fork+exit	129.0783	3.2668	122.4000	137.7073
Process fork+execve	594.1023	10.1764	563.3000	622.6667
Process fork+sh	1176.8880	19.9638	1113.0000	1215.0000
Simple read	1.0437	0.0149	1.0207	1.1109
Simple write	0.9836	0.0397	0.9479	1.2274
Simple stat	2.2547	0.0418	2.1703	2.4678
Simple fstat	0.6598	0.0173	0.6435	0.7232
Simple open/close	3.3546	0.0515	3.2460	3.5216
Select on 500 fd's	19.8101	0.3660	19.3444	21.4862
Pipe latency	11.2702	0.0838	11.1587	12.0729
AF_UNIX latency	12.5225	0.0717	12.3987	12.7805
UDP latency	17.0421	0.1113	15.9089	17.3074
TCP latency	17.8300	0.2080	17.6456	19.5522
File write bandwidth	63576.9450	370.2997	62130.0000	64002.0000
Socket bandwidth	377.8209	21.7406	342.7500	402.5900
AF_UNIX bandwidth	617.4526	2.8237	602.8400	621.9300
Pipe bandwidth	1845.4127	13.6448	1756.6700	1874.1400

Table 2
`Label-0` kernel results

Test	Mean	Std. dev.	Min	Max
Simple syscall	0.4499	0.0019	0.4423	0.4666
Sig inst.	0.6666	0.0094	0.6580	0.7540
Sig hand.	1.3404	0.0360	1.2980	1.5310
Process fork+exit	128.3902	2.5452	122.7222	138.3250
Process fork+execve	596.1760	8.2989	570.9474	614.5882
Process fork+sh	1173.6708	17.4250	1119.5000	1198.3000
Simple read	1.1235	0.0336	1.0807	1.2916
Simple write	0.9839	0.0290	0.9469	1.1514
Simple stat	2.2744	0.0547	2.1665	2.5250
Simple fstat	0.6649	0.0201	0.6450	0.7531
Simple open/close	3.4365	0.0960	3.2972	4.0867
Select on 500 fd's	20.0437	0.3377	19.3631	23.8216
Pipe latency	11.4273	0.2647	11.2593	14.2089
AF_UNIX latency	12.6802	0.4095	12.5021	16.1295
UDP latency	17.1144	0.1865	15.1329	17.6448
TCP latency	18.0908	0.4819	15.7554	21.9137
File write bandwidth	63571.9400	373.0444	62303.0000	64002.0000
Socket bandwidth	385.6075	28.4093	324.6300	415.7700
AF_UNIX bandwidth	612.9818	4.7987	563.8100	618.4900
Pipe bandwidth	1842.9119	10.5459	1811.4800	1872.4000

Table 3
`Label-s` kernel results

Test	Mean	Std. dev.	Min	Max
Simple syscall	0.4498	0.0019	0.4494	0.4682
Sig inst.	0.6650	0.0067	0.6560	0.6880
Sig hand.	1.3364	0.0261	1.3060	1.5540
Process fork+exit	129.1161	2.1423	122.6444	136.9268
Process fork+execve	602.4783	7.6896	576.6000	617.2222
Process fork+sh	1185.3990	17.1613	1118.6000	1212.2000
Simple read	1.2139	0.0394	1.1284	1.3454
Simple write	1.3903	0.0509	1.3116	1.5767
Simple stat	2.2739	0.0783	2.1796	2.5328
Simple fstat	0.6662	0.0209	0.6444	0.7495
Simple open/close	3.4457	0.0744	3.3386	3.7203
Select on 500 fd's	19.9333	1.6248	19.3259	40.5520
Pipe latency	11.4340	0.0521	11.3264	11.6137
AF_UNIX latency	13.5479	0.0720	13.3962	13.8179
UDP latency	18.0801	0.1492	17.7706	18.9459
TCP latency	18.9715	0.2525	16.5678	19.2630
File write bandwidth	63630.4250	326.9002	62478.0000	64002.0000
Socket bandwidth	365.5163	29.6615	320.9600	397.3400
AF_UNIX bandwidth	544.0799	13.8608	450.0500	551.4300
Pipe bandwidth	1835.8787	18.6896	1790.7000	1880.0400

Table 4
`Label-l` kernel results

Test	Mean	Std. dev.	Min	Max
Simple syscall	0.4501	0.0025	0.4487	0.4683
Sig inst.	0.6658	0.0085	0.6560	0.7300
Sig hand.	1.3393	0.0259	1.2970	1.5390
Process fork+exit	129.1359	2.1143	122.9778	136.3902
Process fork+execve	604.5881	7.8515	577.0000	620.0000
Process fork+sh	1187.9500	17.7554	1122.0000	1223.4000
Simple read	1.3297	0.0492	1.2599	1.4668
Simple write	1.4934	0.0387	1.4354	1.5906
Simple stat	2.3362	0.1001	2.1991	2.5577
Simple fstat	0.6621	0.0145	0.6435	0.7580
Simple open/close	3.4701	0.0978	3.3426	3.7235
Select on 500 fd's	19.9283	1.4843	19.3408	40.2463
Pipe latency	11.4582	0.2381	11.3235	14.7124
AF_UNIX latency	13.5722	0.1859	11.7654	15.0804
UDP latency	18.0479	0.2078	15.6827	18.4821
TCP latency	18.9950	0.2842	16.4914	20.4588
File write bandwidth	63595.2050	339.2577	62478.0000	64002.0000
Socket bandwidth	366.4497	27.7231	324.0000	397.0400
AF_UNIX bandwidth	543.9735	9.5682	431.7300	551.9100
Pipe bandwidth	1833.3753	17.3504	1767.1600	1866.7300

VITA

VITA

Florian Buchholz was born in Braunschweig, Germany. In 1998 he received a Diplom in *Informatik* from the Technische Universität Braunschweig, Germany and in 2000 a Master's degree in Computer Science from Purdue University in West Lafayette, Indiana. He earned his Ph.D. in Computer Sciences from Purdue University in 2005. In the fall of 2005 he will be joining the faculty of the Computer Science Department at James Madison University in Harrisonburg, Virginia as an Assistant Professor.