AUTHORIZATION AND TRUST IN SOFTWARE SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kevin A. Walsh

January 2012

AUTHORIZATION AND TRUST IN SOFTWARE SYSTEMS

Kevin A. Walsh, Ph.D.

Cornell University 2012

Nexus Authorization Logic (NAL) provides a principled basis for specifying and reasoning about credentials and authorization policies. It extends prior access control logics that are based on “says” and “speaks for” operators. NAL enables authorization of access requests to depend on (i) the source or pedigree of the requester, (ii) the outcome of any mechanized analysis of the requester, or (iii) the use of trusted software to encapsulate or modify the requester. To illustrate the convenience and expressive power of this approach to authorization, a document-viewer application suite was implemented for the α-Nexus operating system. One of the viewers enforces policies that concern the integrity of excerpts a document contains; another viewer enforces confidentiality policies specified by labels tagging blocks of text; and a third viewer enforces policies that impose chain-of-custody restrictions on stages of an image-editing pipeline. To study how compatible this approach to authorization is with existing principles for building trustworthy systems, a filesystem that pervasively instantiates a number of well-known security principles was implemented for α-Nexus. The design and overall performance of this filesystem was compared to a Linux filesystem that largely ignores the security principles.

BIOGRAPHICAL SKETCH

Kevin Walsh was born in Rochester, New York. He received the B.S., *magna cum laude*, in Computer Science from Cornell University in 1998. Kevin then moved to Fria, Republic of Guinea, where he spent two years working as a high school math teacher with the Peace Corps. In the summer of 2000, he returned to New York to marry Jessica Seem. He received the M.S. in Computer Science from Cornell University in 2007. As part of his doctoral studies, he completed a graduate minor in Math Education and became a certified high school math teacher. Kevin and Jessica have two sons, Elliot and Casey.

To Jess, Elliot, and Casey.

ACKNOWLEDGEMENTS

I thank the institutions that provided direct support for my research through fellowships: a National Science Foundation Graduate Research Fellowship (2002–2005) and an Intel Foundation Fellowship (2006–2008). The research presented in this dissertation was supported in part by NICECAP cooperative agreement FA8750-07-2-0037 administered by AFRL, AFOSR grant F9550-06-0019, National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft and Intel. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

I owe this dissertation—and much else—to Fred Schneider. He provided advice, support, and encouragement when it was needed. His patience and endless help made all the difference. He is a true mentor.

Andrew Myers provided useful feedback on the research that led to this dissertation. David Henderson has been invaluable as a source of inspiration and advice as I pursued my interest in mathematics education. I thank them both for reading drafts of this dissertation, suggesting needed improvements, and patiently serving as members of my committee during the many years of my graduate studies.

I have lived and studied at Cornell for much of the last eighteen years. Three faculty in particular helped make Cornell my home. Thorsten von Eicken started me on this path by showing me what it means to teach and by kindling my interest in research. During my undergraduate studies, and many times since, I benefited from his enthusiasm and kindness. He remains an inspiration to me. Throughout my time at Cornell I have benefited from Ken Birman's consistent encouragement and, when needed, occasional prodding. For the last several years—ever since he joined the Cornell faculty—Hakim Weatherspoon has provided friendly encouragement and advice.

I am indebted to many other faculty and friends from Cornell and elsewhere. Amin Vahdat took a chance on me when, as a recently returned Peace Corps math teacher, I arrived unannounced at his office door at Duke University. He gave freely of his time, knowledge, and energy over the two years that I stayed at Duke. I thank him especially for catapulting me back into systems research after my time abroad. Rajinder Khosla of the National Science Foundation—and long-time family friend—provided encouragement and advice when I sought a fellowship and at every step of my studies. I thank Martín Abadi for his technical advice, particularly for proposing GROUP MONOTONICITY. In addition, he, Lujo Bauer, Willem de Bruijn, Michael Clarkson, Robert Constable, Joe Halpern, and Ted Wobber gave useful feedback on the research that led to this dissertation. Eva Tardos helped me navigate the many difficulties of graduate school, and I am grateful to her. I thank Walker White for his reliable career advice while I prepared to finish my graduate studies.

Many thanks to all those graduate students with whom I studied, researched, and wrote, and whose friendship I am honored to have shared, especially Filip Radlinski, Tom Roeder, Dan Williams, Alan Shieh, and Patrick Reynolds (first at Duke, then again at Cornell).

I can't thank my parents, Edward and Janet Walsh, enough. They have encouraged and supported me through everything. I try to live by their example, and teach my children the same. Jessica's parents, Bob and Carolyn Seem, have always been kind and supportive, and they have given consistently helpful advice on matters large and small.

Just a few days before starting my graduate studies, my first son, Elliot, was born. The second, Casey, arrived before I completed my Master's degree. But while I have struggled to make this small contribution to understanding what makes a computer system trustworthy, Jessica has taught our sons to communicate, to reason, and to act. I am humbled by their growth, and by Jessica's determination to do right in all things. I love you all.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Figures . . . ix

1 Introduction **1**
1.1 A Model . . . 2
1.2 Authorization Policies . . . 3
1.2.1 Discretionary Access Control (DAC) . . . 3
1.2.2 Mandatory Access Control (MAC) . . . 8
1.2.3 Role-Based Access Control (RBAC) . . . 12
1.3 Credentials-Based Authorization . . . 13
1.4 Principles for Trustworthy Systems . . . 16
1.5 Contributions of This Dissertation . . . 20
1.5.1 Nexus Authorization Logic . . . 22
1.5.2 A Document Viewer Suite . . . 23
1.5.3 A Mutual-Suspicion Filesystem . . . 24
1.6 Organization of This Dissertation . . . 25

2 Nexus Authorization Logic **26**
2.1 NAL Syntax and Semantics . . . 26
2.1.1 Beliefs and says . . . 28
2.1.2 Deduction and Local Reasoning . . . 29
2.1.3 Delegation . . . 30
2.2 Predicates and Terms in NAL . . . 32
2.3 Principals in NAL . . . 33
2.3.1 Sub-principals . . . 35
2.3.2 Groups . . . 37
2.3.3 Principals in Practice . . . 38
2.4 Guards: Theory and Practice . . . 43
2.4.1 Credential Distribution and Revocation . . . 44
2.4.2 Credential Distribution in α-Nexus . . . 48
2.4.3 Sources of Derivations . . . 57
2.4.4 Authenticated Channels in α-Nexus . . . 58
2.5 Discussion . . . 61
2.5.1 Genesis of NAL . . . 61
2.5.2 Other Related Work . . . 65

3 A Document-Viewer Suite Using NAL **68**
3.1 *TruDocs*: Analytic and Axiomatic Bases for Authorization . . . 68
3.2 *ConfDocs*: A Synthetic Basis for Authorization . . . 78
3.3 *CertiPics*: Axiomatic and Synthetic Bases for Authorization . . . 83

3.3.1 Axiomatic Bases for Image Integrity 84
3.3.2 Synthetic Bases for Image Integrity 85
3.3.3 Implementation Details 90

4 A Mutual-Suspicion Filesystem 97
4.1 Use of the α-Nexus Operating System 100
4.1.1 Credentials-Based Authorization for MSFS 101
4.1.2 α-Nexus Shared Memory and IPC Channels 106
4.1.3 α-Nexus Device I/O Privileges 108
4.2 System Decomposition 109
4.3 Structure and Organization of MSFS 111
4.3.1 File Access Requests 113
4.3.2 Caching 114
4.4 MSFS Implementation: Components and Privileges 115
4.4.1 Cache Management Component 117
4.4.2 Policy Management Component 119
4.4.3 TCB for MSFS DAC Policy Enforcement 123
4.4.4 Disk Driver Components 124
4.4.5 Filesystem Driver Components 126
4.4.6 VFS Components 128
4.4.7 Data Replication in MSFS 129
4.4.8 MSFS Implementation Optimizations 130
4.5 Filesystem Evaluation 133
4.5.1 TCB Contributions 133
4.5.2 The Cost of Isolation 137
4.5.3 Filesystem Performance 141
4.6 Discussion 152
4.6.1 Secure Filesystems 152
4.6.2 Security Principles 156

5 Concluding Remarks 161

A NAL Inference Rules 167
A.1 NAL Rules from Constructive Predicate Logic 168
A.2 NAL Definitions and Rules Derived from CDD 169
A.3 NAL Extensions 170
A.4 NAL Derived Inference Rules 170

B NAL Guard Library and Proof Checker 171
B.1 Formula Representations 172
B.2 Stack-Based Proof Checking 173
B.3 Implementation Details 176
B.4 External Theories 178

Bibliography 179

LIST OF FIGURES

2.1 NAL syntax. 27
2.2 NAL abbreviations. 27

3.1 A subset of *TruDocs* credential and use policy terms. 75
3.2 *NYT* policy—an example image integrity policy. 86
3.3 *JCB* policy—an example image integrity policy. 86
3.4 A subset of *CertiPics* image transformation programs. 92
3.5 Outline of *CertiPics* proof generator strategy for *NYT* and *JCB*. 93

4.1 A proof of the authorization policy for MSFS's `format` method. 102
4.2 Structure of the MSFS filesystem. 112
4.3 Final implementation of the MSFS filesystem. 116
4.4 Lines of code for MSFS and two alternative designs. 134
4.5 Lines of code contributed by various filesystem designs to TCBs. 135
4.6 Syscall latency and IPC latency micro-benchmark results. 138
4.7 Syscall throughput and IPC throughput micro-benchmark results. 138
4.8 Open/close latency benchmark results for MSFS and Linux FAT32. 142
4.9 Read latency benchmark results for MSFS and Linux FAT32. 144
4.10 Read throughput benchmark results for MSFS and Linux FAT32. 145
4.11 Read latency and throughput benchmark results for various clients. 147
4.12 Enumerate latency benchmark results for MSFS, MSFS-*, and Linux FAT32. 149

CHAPTER 1

INTRODUCTION

Authorization is the process of deciding whether a given *principal* (e.g., a user) should be allowed to perform some specified action (e.g., read a file). Authorization is implemented in a software system by a *guard*, which checks requests from a principal to perform actions. The guard decides whether to authorize or deny a request, usually based on which principal made the request, the nature of the request, and perhaps the system state.

Authorization thus often requires a means for attributing each request to the principal that made the request. If a valid user name and password is entered on the keyboard, for example, then we might attribute each subsequent request from the keyboard to the user with that name. And if a digitally signed request arrives over some channel, and the signature can be verified using some public key, then we might attribute the request to a principal that knows the corresponding private key. The process of attributing requests to principals is known as *authentication*.

Support for *audit*—the ability to record, review, and evaluate past system actions—is important for establishing accountability. Audit can be facilitated by recording in an *audit log* all actions that are performed. The audit log is used by administrators to diagnose problems or verify compliance with some security policy. An audit log is most useful if it records not only *which* actions were performed but which principal requested that they be performed and *why* they were allowed to be performed. If an administrator, for example, discovers while reviewing an audit log that a file was accessed inappropriately, then it is essential to know which principal accessed the file and, ideally, why the access request was authorized. Authorization helps to create a comprehensive audit framework by recording decisions made by guards, along with the inputs used to make those decisions.

Authorization, authentication, and audit—collectively dubbed the gold standard [80] because *Au* is the atomic symbol for gold—are widely considered essential for computer security. This dissertation explores new approaches to authorization and its implementation in computer systems.

1.1 A Model

In the model we adopt in this dissertation for authorization, a *subject* is a principal that makes requests. Some *authorization policy*, which is a set of rules, prescribes which requests are authorized. And an *object* is an entity on which *operations* are performed. For authorization intended to safeguard the integrity or confidentiality of data in a filesystem, for example, the subjects might be users, and the objects might be files and directories, with operations that include read, write, enumerate, delete, etc.

There is considerable flexibility in the choice of subjects. A user is a subject, as discussed previously; users make requests by employing a keyboard or other input devices or by running programs that make requests on that user's behalf. A process executing above an operating system kernel also could be a subject; it makes requests by invoking system calls. A thread within a process likewise can be a subject. A machine might send messages conveying requests over a network, and various hardware components within a single machine make requests to each other, so machines and hardware components can be seen as subjects, too. Note, some of these subjects are fine-grained while others are coarse grained. A process is a finer-grained subject than a user, for example, because many processes make requests on behalf of the same user.

Just as there is flexibility in the choice of subjects, there is flexibility in the choice of objects and operations. In this chapter, we limit our discussion to objects implemented by an operating system kernel and operations that correspond to system calls. In subsequent chapters, we discuss authorization in other contexts.

1.2 Authorization Policies

For authorization, as elsewhere, separation of mechanism and policy has advantages [82]. It allows policy changes to be deployed quickly and without updating code, so long as the existing guards support the new policies. It also allows guard implementations and policies to be separately developed, debugged, or analyzed for correctness.

We view authorization policies in terms of *privileges*, each of which represents a right to perform certain operations on certain objects. When a subject is allowed to perform some operation p on an object, we say the subject *holds* privilege p for that object. The set of privileges held by a subject need not be static. We should expect privileges will be granted or revoked in response to various events, and the set of privileges that a subject holds might depend on the system state.

One way to evaluate an approach to implementing guards is based on how effectively such guards support a desired class of policies. We briefly consider a few classes of policies and guards that have played an important role in the development of authorization for software systems. This discussion is primarily intended to illustrate the diversity of policies that an authorization framework might strive to support. However, it also provides examples to which we will return.

1.2.1 Discretionary Access Control (DAC)

A *discretionary access control* (DAC) policy is one in which the *owner* of an object controls the assignment privileges for that object to other subjects. DAC is one of two classes of policy described in the "Orange Book" [46], ITSEC [39], and later the Common Criteria [40]—influential documents that define requirements for secure software systems intended for the U.S. Department of Defense and others. The policies commonly enforced by filesystems systems are DAC policies: the user that creates a file or directory is the owner and specifies which other subjects are allowed to read, write, or execute a file, enumerate a directory, etc.

A formalization of DAC policies can be obtained by representing privileges in an *access control matrix* [79] m containing a row for each subject and a column for each object, where each entry $m[S, o]$ specifies the set of privileges held by subject S for object o. A guard authorizes a request from subject S to perform operation p on object o if and only if $p \in m[S, o]$ holds. Traditionally, a special privilege `own` is defined, and `own` $\in m[S, o]$ is interpreted to mean that S is an owner of o, hence that S can control the assignment of privileges for o. Ownership is at the heart of DAC, so there is a distinction between `own` and other privileges. To reinforce that distinction, this dissertation departs from the traditional formalization and instead introduces a separate *owner relation* to define which subject owns each object. If every object has exactly one owner—a common restriction, and one we adopt here—then we write $owner(o)$ to denote the subject that owns object o. Usually $owner(o)$ is the subject that requested that o be created.

Changes to the access control matrix are necessary whenever objects or subjects are created or destroyed and whenever privileges are granted or revoked. These changes must themselves be performed only in response to an authorized request; otherwise, a subject could circumvent DAC by simply adding any desired privileges to the corresponding entry of the access control matrix. So a DAC policy will also specify special operations, called *commands*, that modify the access control matrix. For example, `addPriv`(S', o, p) might add privilege p to $m[S', o]$ and `delPriv`(S', o, p) might remove privilege p from $m[S', o]$. In a DAC policy, $owner(o)$ is authorized to invoke these commands, but most other principals are not.

An owner might want to confer on some other subject S' the ability to invoke `addPriv`$(\cdot, \cdot, \cdot)$. This is an example of *delegation of authority*. One way to formalize delegation of authority is by defining a special privilege $p*$ for each ordinary privilege p and allowing any subject S' that holds $p*$ for object o to invoke `addPriv`(S'', o, p). Thus an owner, by invoking $\texttt{addPriv}(S', o, p*)$, can delegate authority for controlling privilege p for object o to subject S'. Subsequently, S' can invoke $\texttt{addPriv}(S'', o, p)$ to grant privilege p to some subject S''.

A system might also allow the owner of an object to be changed by providing commands that modify the owner relation. Usually, only system administrators are allowed to invoke these commands. One reason for allowing system administrators, but not owners, to change $owner(o)$ is that ownership brings certain responsibilities. A user might be responsible for the contents of files they own, for example, or a user might be charged for storing those files. But one consequence of granting special powers to system administrators is that now a system administrator may modify entries in the access control matrix for any object simply by becoming the owner of that object first.

The access control matrix formalism is naturally supported in two ways: access control lists and capabilities. We discuss these briefly before considering other types of authorization policies.

Access Control Lists (ACLs)

An *access control list* (ACL) for an object summarizes one column of the access control matrix. Thus the ACL for object o is a list of entries, each of which contains the name of a subject S and a description of the privileges $m[S, o]$ that S holds for o. Accompanying each ACL is an *owner record* specifying the name of the object's owner. The owner record represents one element of the owner relation. ACLs are one of the earliest mechanisms for enforcing DAC policies [41].

The integrity of an ACL and owner record can be maintained by storing them along with the object they are associated with—for example, as part of the on-disk representation of a file or directory—and by managing ACLs and owner records within the operating system kernel.

When ACLs are used, the privileges held by a particular subject are listed with the different objects, hence privileges typically are stored in a variety of different memory and disk locations. This hinders the ability to review the set of privileges a subject holds. However, review for a single object is cheap. An owner can examine the ACL for an object to ensure certain subjects do not appear, for example.

In many settings, privileges are granted to subjects by virtue of membership in a group. Each user that is a member of the group $Students$, for example, might be granted read privileges for every file in a directory $Homeworks$. A change in group membership can be expensive if it requires updating ACLs for many different objects. One way to avoid this expense is to allow group names to appear on ACLs. Groups provide a level of indirection that eliminates the need to update ACLs for multiple objects when group membership changes. Allowing groups to appear on ACLs can also reduce the size of ACLs, potentially leading to reduced storage and run-time costs.

When ACLs are being used, the guard for object o simply checks, for each requested operation, if the operation and requester's name is listed on the ACL for o. Most modern operating systems implement guards with broad support for ACLs. The traditional user/group/other file-permissions scheme from Unix [135] can be seen as a (particularly limited) kind of ACL containing exactly three entries: a set of privileges for the user, a set of privileges for other members of one of the user's groups, and a set of privileges for all other subjects.

Capabilities

A *capability* [45] is a pair comprising an object name and a set of privileges for that object. Capabilities can be used to implement DAC; a capability held by a subject represents a single entry of an access control matrix. And a *capability list*, which enumerates all of the capabilities held by a single subject, represents a single row of an access control matrix. A survey of early capability-based systems is provided by Levy [83].

Integrity of capabilities and capability-lists can be ensured through a variety of implementation approaches.

- Capabilities can be stored within or managed by an operating system kernel.
- Special-purpose hardware can be used to manage capabilities.
- Each capability can be represented by a message containing an object name and associated privileges, signed with the private key of the object's owner, the kernel, or some other principal.
- Capabilities can be implemented as typed objects within a type-safe programming language, where strong typing ensures the integrity of the capabilities.

When capabilities are managed by the kernel or hardware, commands like `addPriv`$(\cdot, \cdot, \cdot)$ are implemented as system calls or special hardware instructions. In the cryptographic and type-safe programming language approaches, a capability is held by each subject that possesses the appropriate signed message or a variable of an appropriate type, so a subject can pass any capability it holds to any subject with which it communicates.

Capabilities and capability-lists are used in commercial operating systems. The file descriptor table for a process is an example of a capability list. Each file descriptor table is maintained by the operating system kernel, with each entry specifying an object (such as a file, socket, or pipe) and a set of privileges that the process holds for that object. A second example comes from a widely implemented POSIX draft standard [70], which defines a set of capabilities that represent privileges to perform operations useful for system administration.[1]

While it is straightforward to enumerate privileges held by a particular subject in a capability-based implementation, review for a particular object can be expensive or impractical. Revocation in a capability-system also can be challenging, since all of the ca-

[1]POSIX capabilities are not true capabilities, because they represent privileges to invoke certain operating system kernel interfaces, rather than representing privileges to invoke operations on an object. There is only one kernel and its interfaces are usually fixed at compile-time, so POSIX need not define mechanisms corresponding to DAC commands for adding or removing objects.

pabilities for an object might not be stored in a central location. And in some capability implementations, particularly those based on cryptography, capabilities are stored and managed entirely by subjects, making revocation and review significantly more difficult.

1.2.2 Mandatory Access Control (MAC)

A mandatory access control (MAC) policy is one in which an administrator or other central authority controls the assignment of privileges for objects to subjects. MAC policies are suitable for organizations that impose uniform policies across all subjects and objects rather than relying on the discretion of users. Some systems implement elements of both DAC and MAC, generally by authorizing a request if and only if both the DAC and MAC policies are satisfied. MAC is the second of two classes of policy described in the Orange Book [46] and subsequent standards [39, 40].

A variety of MAC policies have been proposed. Below, we discuss two common types. Domain type enforcement [8] and Chinese wall [29] policies are also well known MAC policies, but we omit them for brevity.

Multilevel Security (MLS) Policies

Intelligence and military communities require control over the dissemination and integrity of information. Multilevel security (MLS) policies [111, 143][2] seek to control dissemination by ensuring that a subject can learn information only if: (i) the subject is deemed sufficiently trustworthy; and (ii) the subject has a need to know the information.

In an MLS policy, each object o is assigned a *classification*, denoted $\lambda_{obj}(o)$, and each subject S is assigned a *clearance*, denoted $\lambda_{subj}(S)$. Object classifications and subject clearances are selected from a set of *security labels*. In one widely studied MLS scheme [44,120],

[2]MacKenzie and Pottinger [89] detail the history of the development of MLS, and Bell [19] provides a retrospective examination of its formalization.

a security label is a pair. The first element of the pair designates a sensitivity level—`U` (unclassified), `C` (confidential), `S` (secret), or `TS` (top-secret)—and the second element of the pair designates a set of compartments constructed from descriptors, such as `crypto`, `nuclear`, etc. There is a strict total order $\sqsubset$ on the sensitivity levels ($\mathtt{U} \sqsubset \mathtt{C} \sqsubset \mathtt{S} \sqsubset \mathtt{TS}$) and the usual partial order $\subseteq$ on sets of compartments. From these we define a partial order relation $\preceq$ for security labels, where $\langle sl, cmpt \rangle \preceq \langle sl', cmpt' \rangle$ holds if and only if $sl \sqsubseteq sl'$ and $cmpt \subseteq cmpt'$ hold. Thus, the security labels are a lattice.

MLS limits the dissemination of information within a system by systematically restricting each subject's ability to read or write data, as follows.

- A subject S is authorized to read data from object o if and only if

$$\lambda_{obj}(o) \preceq \lambda_{subj}(S).$$

- A subject S is authorized to write data to object o if and only if

$$\lambda_{subj}(S) \preceq \lambda_{obj}(o).$$

These policies were formalized by Bell and La Padula [18] who then proved that, if a system enforces such a policy, then information never flows from entities (either subjects or objects) with high security labels to entities with low security labels.

In practice, MLS can be quite complex. Policies might include citizenship requirements for users, object secrecy levels may change over time on a fixed schedule, and user clearances may be contingent on successful polygraph and background checks, either of which may expire after a fixed interval. Garg et al. [57] provide a case study of MLS policies as implemented by U.S. intelligence agencies.

MLS policies are MAC policies because authorization decisions are not at the discretion of users. This does not imply that decision making for MAC is centralized. For instance, when the objects in the system are text documents, authority to assign classifications is typically delegated to one or more *classification authorities* trusted by the administrator. And when subjects are human users, authority to assign clearances is delegated to a set of *clearance authorities* trusted by the administrator. Thus, a guard that implements MLS might be forced to use input from a variety of sources when making authorization decisions for read and write requests.

Biba Integrity Policies

The Bell and La Padula formalization of MLS concerns the confidentiality of information. Biba [22] proposed policies for protecting data integrity. Similar to MLS, Biba integrity policies assign each subject and object an *integrity label* chosen from some lattice. We use $\Lambda_{subj}(S)$ to denote the integrity label assigned to subject S, and $\Lambda_{obj}(o)$ to denote the integrity label assigned to object o. We might define two integrity levels, `HI` (high integrity) and `LI` (low integrity), with a strict total order $\sqsupset$ on them (`HI` $\sqsupset$ `LI`), and a set of categories, such as `production`, `development`, etc. A security label is then a pair $\langle il, cat\rangle$ comprising an integrity level il and a set cat of categories. Given the usual partial order $\supseteq$ on sets of categories, we form a lattice by defining a partial order relation $\succeq$ for integrity labels, where $\langle il, cat\rangle \succeq \langle il', cat'\rangle$ holds if and only if $il \sqsupseteq il'$ and $cat \supseteq cat'$ hold.

A Biba policy restricts a subject's ability to access objects.

- A subject S is authorized to read data from object o if and only if
$$\Lambda_{obj}(o) \succeq \Lambda_{subj}(S).$$
- A subject S is authorized to write data to object o if and only if
$$\Lambda_{subj}(S) \succeq \Lambda_{obj}(o).$$

It is clear that Biba and MLS policies share a similar structure. The lattice of security labels used by an MLS policy is, in fact, the dual of the lattice of integrity labels in a Biba policy.

And whereas MLS only allows information to flow upwards in the security label lattice, Biba only allows information to flow downwards in the integrity label lattice.

Clark-Wilson Policies

The policies discussed thus far concern read and write operations. But in many commercial settings, allowing any single subject to have unrestricted read or write privileges for an object would be considered an unnecessary risk. In a bank, for example, an employee might be deemed sufficiently trustworthy to transfer funds from one account to another, but not to withdraw or to add funds from/to either account, even though all involve the same read and write actions on the same underlying data. For this reason, the MLS and Biba policies are ill suited for use in commercial settings.

There are several additional problems with using MLS and Biba in a commercial setting. First, there might not be an overall ordering on the trustworthiness of employees or on the classification of business data. Instead, the integrity and confidentiality of important data might be protected through the use of *separation of duties*, which helps prevent errors and fraud by requiring that multiple subjects cooperate in carrying out an action. In addition, the integrity of an object in a commercial setting is not defined in terms of which subjects have performed operations on the object but by constraints on the object's data. A bank's ledgers, for example, must balance at all times.

Clark and Wilson [36] proposed a set policies to address these issues. Clark-Wilson policies prohibit subjects from modifying data directly, but instead require subjects to invoke *trusted procedures*, programs that implement transactions on system data. Trusted procedures perform *well-formed transactions*, which, by definition, transform system state only in integrity-preserving ways. So, if the system state satisfies integrity constraints before a trusted procedure is executed, then the system state after execution will satisfy the integrity constraints.

In a Clark-Wilson policy, subjects hold privileges for invoking only certain trusted procedures, rather than for invoking individual read and write operations on particular objects. Assignment of privileges to subjects is determined by some central authority, making Clark-Wilson a MAC policy. However, Clark-Wilson policies do not require subjects or objects to have security labels. Instead, access to system data depends on which subject is making a request and which trusted procedure the subject invokes to carry out the request.

1.2.3 Role-Based Access Control (RBAC)

In both DAC and MAC, a subject may have a variety of responsibilities, each of which requires the subject to hold some set of privileges. If a subject's responsibilities change, then the set of privileges the subject holds might also have to change. Role-based access control (RBAC) [52, 121] introduces *roles* as an aid to managing the assignment of privileges to subjects in such scenarios. A role is assigned a set of privileges necessary for some task or responsibility. For example, the $\mathit{Student}$ role might be assigned privileges sufficient to submit homeworks to be graded, while the Grader role is assigned privileges to read submitted homeworks and modify a database of grades. Roles can form a hierarchy, with higher roles defined in terms of lower roles. The $\mathit{Instructor}$ role, for example, would be assigned, at minimum, all the privileges assigned to the Grader and $\mathit{Student}$ roles.

With RBAC, each subject is assigned some set of possible roles, and a subject is at any time allowed to *inhabit* one or more of these assigned roles. When a subject inhabits a role, the subject holds all of the privileges associated with that role. So only subjects assigned the $\mathit{Instructor}$ role can inhabit that role. And a subject might chose to inhabit the Grader role, rather than the $\mathit{Instructor}$ role, when grading submitted homeworks. This helps avoid accidental misuse of privileges.

The assignment of roles to subjects can be governed by DAC or MAC policies, or a combination of both. For example, an administrator might assign the $\mathit{Instructor}$ role to some set of subjects, and any subject that inhabits the $\mathit{Instructor}$ role might be allowed to control the assignment of $\mathit{Student}$ and Grader roles. More complex rules are also possible. We might stipulate, for example, that no subject be assigned both $\mathit{Student}$ and Grader roles, or at least that no subject inhabit both these roles simultaneously, which has the character of a MAC policy.

1.3 Credentials-Based Authorization

In *credentials-based authorization*, requests to perform actions are accompanied by *credentials*. Each request is examined by a guard that uses the credentials accompanying the request, perhaps augmented with other credentials conveying information about system state, to make authorization decisions. Authorization decisions thus can be decentralized, with authority shared by the guard and the principals who issue credentials.

Accountability for authorization decisions is made explicit through the credentials. So for each request that is authorized by the guard, an audit log can record that outcome and the credentials used to justify the guard's decision. The result is a particularly descriptive form of audit that identifies the basis on which subjects' requests are authorized.

As an example, consider how credentials-based authorization can be used to enforce an MLS policy for requests from some user Alice to read some document d. Two credentials might be passed to the guard: a credential issued by a clearance authority describing Alice's clearance, and a credential issued by a classification authority describing the classification of document d. The guard would authorize the request if and only if the evidence contained in these credentials is sufficient to discharge the MLS policy that Alice's clearance dominates d's classification. Notice that attribution is an important part of a credential, because the issuer of each credential can be a factor in determining whether the information conveyed by that credential should be trusted. Suppose, for example, the administrator for an MLS policy trusts clearance authority CA but not CA'. Then a credential from CA describing $Alice$'s clearance might result in $Alice$'s request being authorized, whereas a similar credential from CA' would not result in a request being authorized.

Authorization Logics

In credentials-based authorization, the choice of language for expressing credentials and policies determines the kinds of policies that can be enforced. A natural choice is to use a formal logic, which defines a set of formulas and a set of inference rules. In this approach, credentials are modeled by formulas that convey facts about the system or the world, policies are modeled by formulas that describe the conditions under which subjects are allowed to perform actions, and a guard authorizes a request if and only if the credentials in evidence discharge the appropriate policy according to the inference rules.

A number of special-purpose logics have been designed for use in authorization—the ones we discuss here derive from a logic due to Lampson et. al. [3,78]. Common features of these logics include schemes for naming or describing principals in addition to operators that model attribution of statements to principals and delegation between principals. Principals in the logic typically include subjects (because subjects make requests to perform actions), entities that issue credentials, and a variety of other entities. Continuing the example of an MLS policy implemented using credentials-based authorization, we might model $Alice$'s credential from some clearance authority CA by a formula:

$$CA \;\textsf{says}\; \texttt{cleared}(Alice, \langle \texttt{C}, \{\texttt{crypto}\}\rangle).$$

The $\textsf{says}$ operator models attribution, so here, the statement $\texttt{cleared}(Alice, \langle \texttt{C}, \{\texttt{crypto}\}\rangle)$ is being attributed to the principal CA. The speaks-for operator ($\rightarrow$) models delegation, so we might model the fact that some CA is trusted by the administrator using a formula:

$$CA \rightarrow Admin.$$

A formal logic for authorization offers the advantage that the meaning of a policy and the consequences of issuing a credential are defined by known inference rules, so they are unambiguous. A formal logic can also provide the basis for implementing guards. Since credentials and policies are interpreted as formulas, a guard can use automated proof search to check whether a policy is satisfied by a request. Or, the work of constructing a proof could be performed elsewhere, with the guard simply checking the proof, using an automated proof checker. In proof-carrying authorization [6], requesters are responsible for constructing proofs, and these proofs accompany requests to perform actions. A guard that checks a proof is likely to be simpler and more efficient than one that creates a proof, because inventing proofs is harder than checking proofs.

Most authorization logics assume that each principal has a unique name and that all requests and credentials can be correctly attributed to those names. One challenge when designing a logic, then, is to support all of the types of principals encountered in practice. Consider, for example, a logic that names each principal using a unique public key. Principals might then sign the requests they make and the credentials they issue by using a private key. Guards would verify such signatures to ensure proper attribution. Unfortunately, cryptographic operations can be costly in practice, both for protecting confidentiality of keys and for creating and verifying digital signatures. Moreover, most principals in practice are ill-suited for storing private keys. A user might rely on a process to manage a private key, but that means the user is also depending on an operating system and perhaps a filesystem, which in turn rely on some hardware. Ideally, an authorization logic would allow each of these entities—kernels, processes, machines, etc.—to be treated as distinct principals in their own right, because each makes requests and can issue credentials. But a logic that equates principals with keys oversimplifies, as does a logic that treats these principals as completely independent entities or fails to make the dependencies between these principals explicit.

1.4 Principles for Trustworthy Systems

Thus far, we have used notions of *trustworthiness* and *trust* somewhat informally. These notions are not identical. A principal is *trustworthy* to the extent that

(i) it performs all tasks expected of it, and no others,

(ii) it does not violate any *security goals*, and

(iii) there is evidence to this effect.

Broadly, these three elements concern functionality, security, and assurance, respectively. To be considered trustworthy, a principal must exhibit these elements despite failures, errors, or attacks. By contrast, to the extent that one principal depends on another for its own functionality or security, then we say that the former principal places *trust* in the latter principal. Ideally, a principal would only place trust in principals that are trustworthy, but this need not be the case—trust can be misplaced. For example, a process can only be as trustworthy as a guard it trusts. And in the context of authorization, trust usually entails relocating authority from one part of a system to another.

We view a system as a set of *components*, each with state and behavior that can be analyzed independently. Thus, some components may be deemed trustworthy, while others are not. A component whose behavior deviates from its specification is defined to be *compromised*; we make no assumptions about possible deviations. The component is thus the smallest unit of compromise, and we do not admit the notion of a partially compromised component.

Because a component that is compromised may violate security goals, building a trustworthy system requires attention to the behavior of each component. Authorization, because it restricts behaviors, can play an important role in the design of trustworthy systems. Designing a powerful authorization framework is not sufficient for achieving trustworthiness, of course; the framework must be correctly deployed and configured.

Various principles have been proposed to guide the design and implementation of systems, and these principles inform how authorization should be used in the service of trustworthiness. In this section, we describe a few of the principles that are frequently advocated in the literature though perhaps not often instantiated in practice.

Several principles concern components that cooperate in some common task. Components cooperate by interacting. For example, they might interact by sharing state, transferring control to each other, or synchronizing their behaviors. Without loss of generality, we assume that all interaction between components occurs by sending and receiving requests, responses, or other types of messages over prespecified channels.[3] If, by sending messages, one component can cause another to perform arbitrary actions, then the recipient is placing *full trust* in the sender. Conversely, a recipient that performs only certain actions at the request of a sender can be considered *suspicious* of the sender. Full trust may lead to violations of security goals, because it enables compromise to spread: a component that places full trust in some compromised component C may itself become compromised by virtue of receiving and acting on messages from C. Thus, suspicion is best encouraged and full trust avoided, leading to a classic security principle [124]:

> **Principle:** ***Mutual Suspicion.*** Prefer designs that reduce the likelihood that any compromised component can cause the compromise of another [124].

This principle can be instantiated by employing a variety of authorization mechanisms, alone or in combination. Examples include the following.

- Restrict the language of requests and responses. This is done, in effect, when an application programming interface (API) specifies a set of interfaces and a protocol or message format understood by those interfaces. When a recipient implements such an API, senders are able to instigate the actions defined by the API but are (presumably) prevented from instigating other actions at the recipient.

[3] Familiar shared memory and method invocation semantics can be formulated in terms of sending and receiving messages over channels, as can all other types of interaction.

- Check incoming requests and ignore those that violate some policy. Following the most typical way to instantiate authorization, a guard is interposed to intercept requests at run-time, perform various checks, then forward to the intended recipient only those requests found to comply with the policy.
- Sanitize incoming messages before acting on them. Web servers, for example, commonly protect against SQL injection and XSS attacks [131] by implementing a transformation that, for every request the Web server receives, replaces suspect character sequences with harmless ones before any further processing.

We previously discussed privileges in the context of authorization guards, but notice that reliance on APIs or sanitization induces a similar notion of privileges. So just as an entry on an ACL or a capability represents a privilege, the ability to link against and invoke an API represents a privilege, albeit one that is enforced at compile-time.

Implicit in the above examples is an assumption that all requests to recipients are subjected to some prescribed measures. A familiar security principle summarizes this obligation [5]:

> **Principle: *Complete Mediation.*** Authorize, using an appropriate enforcement mechanism, every request for a component to perform some action [5].[4]

What obligations are entailed by Complete Mediation depend on the particular mechanisms used to authorize requests. For guards or sanitization, Complete Mediation requires that all requests to a component be checked by an appropriate guard or be appropriately sanitized. And when Mutual Suspicion is instantiated by selecting a restricted set of APIs, Complete Mediation requires that all requests be handled by the appropriate API.

Complete Mediation presumes that components interact only over some set of well-defined channels. Some form of component isolation is thus implicit. An operating sys-

[4]Anderson [5] described did not provide a name for this principle; the term *Complete Mediation* is due to Saltzer and Schroeder [119].

tem typically implements isolation between processes by using a virtual memory architecture that prevents one process from accessing state associated with another process. And within a single process, software techniques (e.g., [21,142]) can provide isolation between objects or code modules. In all cases, boundaries created by isolation mechanisms define the system's components.

Assurance of system trustworthiness requires analyzing all components that hold privileges for any action that might violate that system's security goals. This suggests a conservative approach to granting privileges, which Saltzer and Schroeder [119] call the Principle of Least Privilege.

> **Principle:** ***Least Privilege.*** Grant every component of the system the fewest privileges necessary to complete its task [119].

Least Privilege facilitates implementing Mutual Suspicion. One way that a component can cause the compromise of another is by abusing privileges that instigate actions at that other component. Eliminating unnecessary privileges thus helps reduce the likelihood that a compromised component will have sufficient privileges to compromise other components.

Of course, instantiating Mutual Suspicion, Complete Mediation, and Least Privilege in a system does not by itself eliminate the possibility that a system's security goals might be violated. These security principles merely help by limiting the impact of compromised components. Some components, by necessity, perform functions that could violate a security goal. This set of components is called the *trusted computing base* (TCB) [103] for that security goal. A TCB likely includes not only software components, but also hardware and firmware.

Components in the TCB for some security goal must all be trusted not to violate that security goal. One way to reduce the risk that such trust is misplaced is prescribed by:

> **Principle:** ***Minimization of Trusted Computing Bases.*** Make each TCB as

small as possible, consistent with the functions it has to perform [103].[5]

A smaller TCB should be easier to analyze; a larger TCB is more likely to have bugs, hence more easily attacked. In an ideal system design, TCBs would be chosen to minimize the probability that security goals are violated. Minimization of Trusted Computing Bases only approximates what is sought, equating greater TCB size with increased probability of compromise.

1.5 Contributions of This Dissertation

A central issue for any authorization mechanism is the underlying rationale for authorization decisions. An untrustworthy principal might attempt accesses that violate a security policy, whereas (by definition) a trustworthy one wouldn't. So a guard would never err in authorizing requests made by trustworthy principals. However, determining whether a principal is trustworthy is rarely feasible, so guards typically substitute something that is easier to check. This dissertation proposes a taxonomy involving three bases for predicting trustworthiness: axiomatic, analytic, and synthetic.

- *Axiomatic bases*. Access control lists embody an *axiomatic basis* for making authorization decisions. Axioms are statements that we accept without proof. With guards that use an ACL, we accept without proof that all principals on the ACL are trustworthy, and the guard only authorizes requests made by these principals. The same rationale is applied when a system uses a principal's reputation as the basis for deciding whether that principal's requests should be authorized. An axiomatic basis is also implicit when a guard authorizes requests to run some executable only if the value of a hash indicates the executable is unaltered from a standard software

[5]The term *trusted computing base* was coined by Nibaldi [103]. Nibaldi's formulation of the security principle, which was subsequently incorporated into the Orange Book [46] and which we follow here, builds on Saltzer and Schroeder's [119] Principle of Economy of Mechanism, which in turn formalizes Schroeder's [124] earlier notions of simplicity in protection mechanism design and implementation.

package or if a digital signature establishes the executable was endorsed by some approved software provider.

- *Analytic bases*. Analysis provides a way to predict whether certain behaviors by a program P are possible, so some guards employ an *analytic basis* for authorizing requests made by principals executing P. Specifically, an analysis establishes that P is incapable of certain abuses and, therefore, granting the request will not enable the security policy to be violated. Proof-carrying code [97] is perhaps the limit case. In this approach, a program P is accompanied by a proof that its execution satisfies certain properties; a request to execute P is authorized if and only if a proof checker trusted by the guard establishes that the proof is correct and that the properties proved are sufficiently restrictive. As another example, some operating systems [21] will authorize a request to load and execute code only if that code was type checked by an analyzer trusted by the operating system; type checking is a form of analysis, and programs that type check cannot exhibit certain malicious or erroneous behaviors.

- *Synthetic bases*. Finally, a *synthetic basis* for authorization is involved whenever a program is transformed prior to execution, if that transformed program is trustworthy in ways the original was not. Examples of this approach include sandboxing [60], software-based fault isolation [142], in-lined reference monitors [50], and other program-rewriting methods [66, 126].

The discussion above suggests that authorization is a proxy for a trustworthiness test, rather than the more traditional view of authorization as a set of mechanisms for filtering requests. Therefore, a guard may be designed to trust a program analyzer, a program rewriter, an administrator, or others in the course of predicting whether a requesting principal is trustworthy for some request. A significant aspect of engineering a trustworthy system is thus deciding when one principal (e.g., a guard) should trust some other principal (e.g., an analyzer).

We conjecture that a single basis for establishing trustworthiness is unlikely to suffice throughout an entire system. Different schemes are going to be useful in different settings, and schemes that combine bases will also be useful—for example, type safety can be enforced by using a hybrid of program analysis (an analytic basis) and code generation that adds run-time checks (a synthetic basis).

This dissertation presents an approach, based on credentials-based authorization, that incorporates and unifies axiomatic, analytic, and synthetic bases for predicting trustworthiness. We seem to be the first to classify authorization schemes according to this taxonomy and the first to entertain creating such a unifying framework. We evaluate the merits of our approach by building novel applications and operating system services, and we examine how instantiating our approach to authorization impacts the instantiation of well-known security principles.

1.5.1 Nexus Authorization Logic

We developed a logic NAL (Nexus Authorization Logic) for specifying and reasoning about credentials and authorization policies.[6] NAL extends Abadi's access control logic CDD [1,2], adding support for axiomatic, analytic, and synthetic bases[7] and adding compound principals (groups and sub-principals). These extensions were designed to help bridge the gap from the simplifications and abstractions found in CDD to the pragmatics of actual implementations.

NAL was designed for use in operating systems like α-Nexus,[8] which employs a Trusted Platform Module (TPM) [137] secure co-processor as a hardware-protected root of trust. NAL's scheme for naming principals and NAL's operators for attribution and delegation were informed by the needs of α-Nexus and the capabilities a TPM offers.

[6]Joint work with Fred Schneider and Emin Gün Sirer.

[7]In fact, any authorization logic that supports a sufficiently expressive language of beliefs should be able to enforce authorization policies according to axiomatic, analytic, and synthetic bases.

[8]α-Nexus is a predecessor version of the Nexus [125] operating system.

1.5.2 A Document Viewer Suite

To illustrate the convenience and expressive power of our approach to authorization, a suite of document-viewer applications was implemented.[9] The suite comprises three applications that run partly or entirely on the α-Nexus operating system.

- *TruDocs* (<u>Tru</u>stworthy <u>Doc</u>ument<u>s</u>) controls the display of documents that contain excerpts and whose integrity is based on policies that restrict the use of those excerpts; it employs an analytic basis for authorization.
- *ConfDocs* (<u>Conf</u>idential <u>Doc</u>ument<u>s</u>) implements multilevel security policies to protect the confidentiality of documents built from text elements that have security labels; it employs both analytic and synthetic bases for authorization.
- *CertiPics* (<u>Certi</u>fied <u>Pic</u>ture<u>s</u>) enforces the integrity of displayed digital images by imposing chain-of-custody restrictions on the image-editing pipeline; it employs synthetic and axiomatic bases for authorization.

Each of these applications relies on α-Nexus system services, and each employs NAL-based authorization guards and NAL credentials. *TruDocs* and *CertiPics* rely on attestation facilities of α-Nexus to create credentials, which are then conveyed to guards on remote machines. In *ConfDocs*, documents are stored within a sealed storage facility implemented by α-Nexus. α-Nexus sealed storage and *ConfDocs* guards work together to ensure Complete Mediation with respect to multilevel security policy enforcement. The policies enforced by guards in all three applications are decentralized, and the guards depend on information gathered from multiple sources.

[9]Joint work with Fred Schneider and Emin Gün Sirer.

1.5.3 A Mutual-Suspicion Filesystem

For an authorization framework to be useful, it should at minimum not preclude the instantiation of security principles that engender trustworthiness. Ideally, an authorization framework would provide leverage so that these security principles are more easily instantiated. We explored the connection between authorization and the security principles of Section 1.4 by implementing a filesystem called MSFS for α-Nexus.[10] MSFS uses NAL extensively for authorization but also pervasively instantiates these security principles. We measured the performance of MSFS; the results suggest that run-time overhead need not be significant for a design that relies on credentials-based authorization using NAL and that embraces a small trusted computing base and the principles of Mutual Suspicion, Complete Mediation, and Least Privilege.

The design of MSFS differs from traditional filesystems, but MSFS still exports a typical interface to clients. As with most other filesystems, MSFS enforces a DAC policy for information stored on disks. For user files, MSFS enforces an authorization policy similar to what is found in conventional filesystems: The owner of a file controls an ACL that specifies access by other users and user groups. MSFS, however, also enforces DAC for meta-data, including file, directory, and filesystem meta-data, disk configuration data, and (nominally) unused portions of disks. In fact, every byte stored by MSFS on a disk is associated with some owner record and ACL. For meta-data, the owner record and ACL entries name components of the MSFS implementation itself. Thus components that comprise MSFS are subjects, too, and their requests to access data are governed by a DAC policy.

MSFS relies on credentials-based authorization internally, and MSFS also implements credentials-based authorization for clients. Owner records and ACLs in MSFS are encoded as NAL expressions, so they can be written in terms of any principal that can be expressed by NAL. This enables rich sets of principals to serve as owners of data and/or

[10]Joint work with Fred Schneider.

to appear on ACLs, and it enables MSFS to authorize requests to access data using credentials from many sources.

1.6 Organization of This Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 gives the syntax and semantics of NAL. It also discusses how NAL can be used to instantiate authorization guards. We draw examples from our experience building guards for the α-Nexus kernel, its system services, and some prototype applications. This is followed by a discussion of related work on authorization logics and other approaches to authorization. Chapter 3 discusses $\mathit{TruDocs}$, $\mathit{ConfDocs}$, and $\mathit{CertiPics}$, focusing on how NAL is used by these applications. Chapter 4 describes the design and implementation of the MSFS filesystem for α-Nexus. Concluding remarks are in Chapter 5. A complete list of NAL inference rules can be found in Appendix A, and Appendix B gives details of the NAL proof checker implementation used by α-Nexus, MSFS, and the applications described in this dissertation.

CHAPTER 2

NEXUS AUTHORIZATION LOGIC

Nexus Authorization Logic (NAL) provides a rigorous basis for specifying and reasoning about credentials and authorization policies. NAL is designed to allow authorization of access requests to depend on any or all of: (i) the source or pedigree of the requester (i.e., an axiomatic basis), (ii) the outcome of a mechanized analysis performed on the requester (i.e., an analytic basis), or (iii) the use of trusted software to encapsulate or modify the requester (i.e., a synthetic basis). In this chapter, we provide an introduction to NAL, focusing on how NAL can be used to specify credentials and authorization policies.

2.1 NAL Syntax and Semantics

NAL extends prior access control logics that are based on "says" and "speaks for" operators. Thus, principals and the says and speaks-for ($\rightarrow$) modalities are what makes NAL different from the higher-order predicate logics often used in programming; the rest of this section discusses those differences in some detail.

The syntax of NAL formulas is given in Figure 2.1, and some useful NAL abbreviations appear in Figure 2.2. There and throughout this dissertation, identifiers typeset in lower-case sans-serif font (e.g., x) denote propositional variables; identifiers typeset in lower-case italic font (e.g., v) denote first-order variables; $\overline{v}$ abbreviates a list $v_1, v_2, \ldots, v_n$; literals (including state-functions, predicates, and constants) are typeset in typewriter font (e.g., `read`); identifiers typeset in upper-case italic font (e.g., A, B, ...) denote principals; and identifiers typeset in calligraphic font (e.g., $\mathcal{F}$, $\mathcal{G}$, ...) denote NAL formulas. The language for terms τ is left unspecified.

We write

$$\textsc{RuleName:}\frac{\mathcal{F}_1,\ \mathcal{F}_2,\ \ldots,\ \mathcal{F}_n}{\mathcal{F}}$$

$$
\begin{array}{rll}
A ::= & \{\!|\mathsf{v} : \mathcal{F}|\!\} \;\mid\; A.\tau & \text{compound principals} \\
\mathcal{F} ::= & f(\tau, \dots) \;\mid\; A \;\mathsf{says}\; \mathcal{F} \;\mid\; \mathsf{x} \;\mid\; (\forall \mathsf{x} : \mathcal{F}) & \\
& \mid\; \mathcal{F} \wedge \mathcal{F} \;\mid\; \mathcal{F} \vee \mathcal{F} \;\mid\; \mathcal{F} \Rightarrow \mathcal{F} \;\mid\; \mathsf{true} & \text{formulas} \\
& \mid\; (\forall \mathsf{v} : \mathcal{F}) \;\mid\; (\exists \mathsf{v} : \mathcal{F}) &
\end{array}
$$

Figure 2.1: NAL syntax.

$$
\begin{array}{rl}
\mathsf{false}: & (\forall \mathsf{x} : \mathsf{x}) \\
\neg\mathcal{F}: & (\mathcal{F} \Rightarrow \mathsf{false}) \\
A \rightarrow B: & (\forall \mathsf{x} : (A \;\mathsf{says}\; \mathsf{x}) \Rightarrow (B \;\mathsf{says}\; \mathsf{x})) \\
A \xrightarrow{\overline{\mathsf{v}}:\mathcal{F}} B: & (\forall \overline{\mathsf{v}} : (A \;\mathsf{says}\; \mathcal{F}) \Rightarrow (B \;\mathsf{says}\; \mathcal{F})) \quad \text{for } \overline{\mathsf{v}} \text{ not free in } A \text{ or } B
\end{array}
$$

Figure 2.2: NAL abbreviations.

to define an inference rule RULENAME that allows *conclusion* $\mathcal{F}$ to be inferred assuming that *premises* $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$ have been. Appendix A gives the complete list of NAL axioms and inference rules. We focus below on aspects of the logic concerned with principals and the says and speaks-for modalities.

NAL adopts many of its inference rules from CDD [1, 2]. Like CDD, NAL is a constructive logic. Constructive logics are well suited for reasoning about authorization [55], because constructive proofs include all of the evidence used for reaching a conclusion and, therefore, information about accountability is not lost. Classical logics allow proofs that omit evidence. For example, we can prove $\mathcal{G}$ using a classical logic by proving $\mathcal{F} \Rightarrow \mathcal{G}$ and $\neg\mathcal{F} \Rightarrow \mathcal{G}$, since from these theorems we can conclude $(\mathcal{F} \vee \neg\mathcal{F}) \Rightarrow \mathcal{G}$, hence $\mathsf{true} \Rightarrow \mathcal{G}$ due to Law of the Excluded Middle. This classical proof, however, does not say whether it is $\mathcal{F}$ or it is $\neg\mathcal{F}$ that serves as the evidence for $\mathcal{G}$, and thus the classical proof is arguably unsatisfactory for an audit of why $\mathcal{G}$ holds.

2.1.1 Beliefs and says

NAL, like its predecessors [1–3, 13, 17, 47, 73, 81, 86], is a logic of beliefs. Each principal A has a *worldview* $\omega(A)$, which is a set of beliefs that A holds or, equivalently, formulas that A believes to be true. NAL formula A says $\mathcal{F}$ is interpreted to mean: $\mathcal{F} \in \omega(A)$ holds.

NAL extends CDD by allowing formulas to include system- and application-defined predicates in place of propositions. Since NAL terms can include the names of principals, NAL formulas can convey information about, hence potential reasons to trust, a principal. For example, the NAL formula

$$\mathit{Analyzer} \text{ says } \texttt{numChan}(P, \texttt{"TCP"}) = 3 \tag{2.1}$$

holds if and only if worldview $\omega(\mathit{Analyzer})$ contains a belief that $\texttt{numChan}(P, \texttt{"TCP"}) = 3$ holds. System-defined function $\texttt{numChan}(P, \texttt{"TCP"})$ yields the number of TCP connections open at P.

A NAL formula like (2.1) could specify a credential or specify (part of) an authorization policy. As a credential, formula (2.1) asserts that $\mathit{Analyzer}$ believes and is accountable for the truth of $\texttt{numChan}(P, \texttt{"TCP"}) = 3$; as a specification for an authorization policy, formula (2.1) compels a guard to establish that $\texttt{numChan}(P, \texttt{"TCP"}) = 3$ is in $\omega(\mathit{Analyzer})$ in order to grant a request.

The worldview of each principal is presumed to contain all NAL theorems. NAL therefore includes a *necessitation* inference rule:

$$\text{SAYS-I:}\frac{\mathcal{F}}{A \text{ says } \mathcal{F}} \tag{2.2}$$

We assume a constructive logical theory for reasoning about system- and application-defined predicates; SAYS-I (2.2) asserts that those theorems are part of each principal's worldview.

Principals may hold beliefs that are not actually true statements and/or that are in conflict with beliefs that they or other principals hold. Just because A says $\mathcal{F}$ holds does not necessarily mean that $\mathcal{F}$ holds or that B says $\mathcal{F}$ holds for a different principal B. However, beliefs that a principal holds are presumed to be consistent with beliefs that same principal holds about its own beliefs:

$$\text{SAYS-E:}\ \frac{A \textsf{ says } (A \textsf{ says } \mathcal{F})}{A \textsf{ says } \mathcal{F}} \tag{2.3}$$

2.1.2 Deduction and Local Reasoning

A principal's worldview is assumed to be *deductively closed*: for all principals A, any formula $\mathcal{G}$ that can be derived using NAL from the formulas in $\omega(A)$ is itself in $\omega(A)$. This supports having the usual implication-elimination rule

$$\text{IMP-E:}\ \frac{\mathcal{F}\ ,\ \mathcal{F} \Rightarrow \mathcal{G}}{\mathcal{G}} \tag{2.4}$$

along with a rule for distributing says over implication:

$$\text{DEDUCE:}\ \frac{A \textsf{ says } (\mathcal{F} \Rightarrow \mathcal{G})}{(A \textsf{ says } \mathcal{F}) \Rightarrow (A \textsf{ says } \mathcal{G})} \tag{2.5}$$

Notice that all formulas in DEDUCE (2.5) refer to the same principal. This *local-reasoning restriction* limits the impact that a principal with inconsistent beliefs can have. In particular, from A says false, DEDUCE (2.5) enables us to derive[1] A says $\mathcal{G}$ for any $\mathcal{G}$, but DEDUCE (2.5) cannot be used to derive B says $\mathcal{G}$ for an arbitrary principal B. So the local-reasoning restriction causes inconsistency within $\omega(A)$ to be contained. The local-reasoning restriction also prevents mutually inconsistent beliefs held by an unrelated set of principals from being combined to derive A says false for any principal A [1,2].

[1]Here is that proof: false $\Rightarrow \mathcal{G}$ is a theorem for all $\mathcal{G}$. Therefore, by SAYS-I (2.2) we conclude A says (false $\Rightarrow \mathcal{G}$) is a theorem for all $\mathcal{G}$. We then use DEDUCE (2.5) and IMP-E (2.4) to derive A says $\mathcal{G}$.

2.1.3 Delegation

The notation $A \rightarrow B$ (read "A speaks for B") abbreviates the NAL formula

$$(\forall \mathsf{x} : (A \ \mathsf{says}\ \mathsf{x}) \Rightarrow (B \ \mathsf{says}\ \mathsf{x})). \tag{2.6}$$

Since propositional variable x in sub-formulas "A says x" and "B says x" of (2.6) can be instantiated by any NAL formula, we conclude that if $A \rightarrow B$ holds then all beliefs in the worldview of principal A also appear in the worldview of principal B; therefore $\omega(A) \subseteq \omega(B)$ holds. In terms of credentials, $A \rightarrow B$ characterizes the consequences of B delegating to A the task of issuing credentials. Not only would A be accountable for such credentials but so would B.

The transitivity of $\rightarrow$ follows directly from definition (2.6), and therefore we have the following as a derived inference rule of NAL:

$$\rightarrow \text{ TRANS:}\frac{A \rightarrow B\ ,\ B \rightarrow C}{A \rightarrow C}$$

One theorem of NAL is

$$(B \ \mathsf{says}\ (A \rightarrow B)) \Rightarrow (A \rightarrow B), \tag{2.7}$$

which implies that the following is a derived inference rule of NAL:

$$\text{HAND-OFF:}\frac{B \ \mathsf{says}\ (A \rightarrow B)}{A \rightarrow B} \tag{2.8}$$

An interpretation of HAND-OFF (2.8) (or equivalently theorem (2.7)) is that each principal B is an authority on its own delegations.

NAL also supports an abbreviation to assert the *restricted delegation* that only certain beliefs held by A are attributed to B. We write $A \xrightarrow{\overline{\mathsf{v}}:\mathcal{F}} B$ (read "A speaks for B about $\mathcal{F}$"), where no identifier in $\overline{\mathsf{v}}$ appears free in B or A, to assert:

$$(\forall \overline{\mathsf{v}} : (A \textsf{ says } \mathcal{F}) \Rightarrow (B \textsf{ says } \mathcal{F}))$$

Such a restricted delegation allows us to specify in NAL that a principal B delegates authority or trusts another principal A for only certain kinds of credentials.

For example, we should not be surprised to find a university registrar $\mathit{UnivReg}$ trusted by academic department CSdept about whether somebody is a student at that university. We specify this trust by writing the NAL restricted delegation formula

$$\mathit{UnivReg} \xrightarrow{\mathsf{v}:\mathsf{v}\in\mathtt{Students}} \mathit{CSdept}, \tag{2.9}$$

meaning

$$(\forall \mathsf{v} : \mathit{UnivReg} \textsf{ says } \mathsf{v}\in\mathtt{Students} \quad\Rightarrow\quad \mathit{CSdept} \textsf{ says } \mathsf{v}\in\mathtt{Students}).$$

Restricted delegation (2.9) limits which credentials issued by $\mathit{UnivReg}$ can be attributed to CSdept. So, credential $\mathit{UnivReg}$ says $\mathcal{F}$ can be used with (2.9) to derive CSDept says $\mathcal{F}$ when $\mathcal{F}$ is "$\mathit{Bob}\in\mathtt{Students}$" but not when $\mathcal{F}$ is "$\mathtt{offer}(\mathtt{CS101}, \mathtt{Spr})$".

Restricted delegation (2.9) limits the abuse of privilege and the spread of bogus beliefs by asserting that CSDept will adopt only certain kinds of beliefs held by $\mathit{UnivReg}$. Some care is required, though, when using this defense. A second unrestricted delegation

$$\mathit{UnivReg} \xrightarrow{\mathsf{v}:\mathsf{v}\notin\mathtt{Students}} \mathit{CSdept}$$

along with (2.9) could allow CSdept says false to be derived if $\mathit{UnivReg}$ is compromised and thus willing to issue bogus credentials

$$\mathit{UnivReg} \textsf{ says } S\in\mathtt{Students}$$

$$\mathit{UnivReg} \textsf{ says } S\notin\mathtt{Students}$$

for some student S.

As with $\rightarrow$, we have a corresponding derived inference rule for transitivity of $\xrightarrow{\overline{\mathtt{v}} : \mathcal{F}}$:

$$\xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} \text{ TRANS}: \frac{A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} B, B \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} C}{A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} C}$$

and we have the NAL theorem

$$(B \textsf{ says } (A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} B)) \Rightarrow (A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} B),$$

which leads to a corresponding derived inference rule:

$$\text{REST-HAND-OFF}: \frac{B \textsf{ says } (A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} B)}{A \xrightarrow{\overline{\mathtt{v}} : \mathcal{F}} B}$$

2.2 Predicates and Terms in NAL

NAL is largely agnostic about how predicates and terms are implemented.[2] But an authorization mechanism that evaluates NAL formulas would be impractical unless efficient implementations are available for NAL predicate and term evaluation. The α-Nexus kernel, for example, provides efficient system routines for processes to read certain operating system state (abstracted in NAL formulas as terms) and to evaluate certain predefined predicates on that state. Also, any deterministic boolean-valued routine running in α-Nexus can serve as a NAL predicate. So if an authorization policy can be programmed in α-Nexus then it can be specified using a NAL formula.

The designer of a guard in α-Nexus must decide what sources to trust for information about the current and past states. Presumably, a guard would trust predicate evaluations that it performs itself or that the α-Nexus kernel performs on its behalf. Other components might have to be trusted, too, because it is unlikely that every principal would be able to evaluate every predicate due to constraints imposed by locality and/or confiden-

[2]Because it is a constructive logic, NAL does require that all terms and predicates be computable.

tiality. Arguably, a large part of designing a secure system is concerned with aligning what must be trusted with what can be trusted. NAL helps focus on these design choices by having each credential explicitly bind the name of a principal to the belief that credential conveys, thereby surfacing what is being trusted.

NAL is agnostic about predicate and function naming, assuming only that each name is associated with a unique interpretation across all principals. One approach it to define an authoritative interpretation (including an evaluation scheme) for each name; all principals are then required to use that. Implicit in such a solution would have to be some way to determine what is the authoritative interpretation for a given name. α-Nexus addresses this by implementing hierarchical naming, where a name encodes the identity of the principal that is the authority for performing evaluations.

2.3 Principals in NAL

Principals are the entities to which beliefs can be attributed. Examples include active entities like processors, processes, and channels, as well as passive objects like data structures and files. Principals in NAL thus encompass both the subjects and the objects of an authorization model, and as will become clear in the discussion below, NAL principals include entities like groups and roles. Some NAL principals make requests and issue credentials, which are modeled as beliefs attributed to those principals. Other NAL principals do not make requests or issue credentials in the traditional sense, but are simply used as a convenience for describing and reasoning about related collections of beliefs.

We require that distinct NAL principals have distinct names and that credentials attributed to a principal cannot be forged. Schemes that satisfy these requirements include:

- Use a public key as the name of a principal, where that principal is the only entity that can digitally sign content using the corresponding private key. A principal named by a public key K_A signifies that a belief $\mathcal{F}$ is in worldview $\omega(K_A)$ by digitally signing an encoding of $\mathcal{F}$. So, a digitally signed representation of the NAL statement $\mathcal{F}$, where public key K_A verifies the signature, conveys NAL formula K_A says $\mathcal{F}$.

- Use the hash of a digital representation of an object as the name of a principal associated with that object. A principal named by hash $H(obj)$ includes a belief $\mathcal{F}$ in its worldview $\omega(H(obj))$ by having an encoding of $\mathcal{F}$ stored in obj.[3] So by having $\mathcal{F}$ be part of obj, $H(obj)$ conveys NAL formula $H(obj)$ says $\mathcal{F}$.

The benefit of using a public key K_A to name a principal is that this name then suffices for validating that a credential K_A says $\mathcal{F}$ has not been forged or corrupted. Also, credentials conveying individual beliefs or subsets of beliefs in $\omega(K_A)$ can be issued at any time. But public-private key pairs are expensive to create. Moreover, private keys can be kept secret only by certain types of principals. With a Trusted Platform Module (TPM) [137], you can associate a private key with a processor and keep the key secret from all software that runs on the processor; without a TPM, you can associate a private key with a processor but keep the key secret only from non-privileged software. And there is no way to associate a private key with a non-privileged program executing on a processor yet have that key be secret from the processor or from privileged software being run.

Hashes are attractive for naming principals, because hashes are relatively inexpensive to calculate and do not require secrets. However, a principal must have read-access to obj in order to generate or validate a credential $H(obj)$ says $\mathcal{F}$ for conveying beliefs that, because they are stored in obj, are part of the worldview of $H(obj)$. Also, the use of hashes for naming principals is useful only for conveying static sets of beliefs held by objects whose state is fixed. Change the beliefs or the state of obj and name $H(obj)$ changes too, which means credentials previously issued for that object could no longer validate.

NAL is agnostic about what schemes are used to name principals. Our experience with α-Nexus applications has been that public keys and hashes both have uses.

[3]We might adopt the convention that every object obj involves two parts. The first part is a possibly empty list of the NAL formulas $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$ in $\omega(H(obj))$; the second part is any other digital content.

2.3.1 Sub-principals

System components often depend on other system components. In hierarchically structured systems, for example, higher levels depend on lower levels. Also, dependencies are created when one component loads and starts executing (or interpreting) another. The dependency of a principal Sub on another principal Dom can be so strong that Sub is *materialized* by Dom, hence Sub says $\mathcal{F}$ holds only if Dom says $(Sub$ says $\mathcal{F})$ holds. For example, execution of a program $Prog$ is ultimately materialized by computer hardware (say) CPU, and therefore $Prog$ says $\mathcal{F}$ holds only if CPU says $(Prog$ says $\mathcal{F})$ holds.

NAL offers *sub-principals* as a convenience for naming a principal that is materialized by another. Given a principal A and any term τ, sub-principal $A.\tau$ is a NAL principal[4] materialized by A. This is captured in a NAL rule:

$$\text{SUBPRIN:}\frac{}{A \to A.\tau} \tag{2.10}$$

Equivalent terms define sub-principals having the same worldviews:

$$\text{EQUIV SUBPRIN:}\frac{\tau_1 = \tau_2}{A.\tau_1 \to A.\tau_2}$$

Here, we assume some theory is available for proving premise $\tau_1 = \tau_2$.

Sub-principals are particularly useful for describing structures where a principal is multiplexed among various roles. For example, processes are often materialized by an operating system that multiplexes a processor. Thus the principal that corresponds to an executing program is a sub-principal of the operating system that runs that program; and the operating system itself is a sub-principal of the processor.

In Section 2.3.3 we discuss in detail how α-Nexus uses sub-principals. Here, as a

[4]Sub-principals can themselves have sub-principals, with left-associativity assumed, so that $A.\tau_1.\tau_2$ abbreviates $(A.\tau_1).\tau_2$.

somewhat simplified illustration, consider a system comprising a certification authority CA being executed by an operating system OS that is multiplexing a processor among a number of applications. And suppose the hash of the in-memory image for CA is h_{CA}, the hash of the in-memory image for OS is h_{OS}, and the processor's TPM stores a private key whose signatures can be verified using public key K_{CPU}. In NAL, these dependencies could be characterized using a sub-principal $K_{CPU}.h_{OS}$ for the OS and a sub-principal $K_{CPU}.h_{OS}.h_{CA}$ for the CA. According to SUBPRIN (2.10), we have:

$$K_{CPU} \rightarrow K_{CPU}.h_{OS} \tag{2.11}$$

$$K_{CPU}.h_{OS} \rightarrow K_{CPU}.h_{OS}.h_{CA} \tag{2.12}$$

A credential attributed to execution of CA would, in fact, be issued by K_{CPU}, materializing operating system OS, materializing CA. So the credential for a belief $\mathcal{F}$ held by CA would be specified by the NAL formula

$$K_{CPU} \text{ says } (K_{CPU}.h_{OS} \text{ says } (K_{CPU}.h_{OS}.h_{CA} \text{ says } \mathcal{F})),$$

from which we can derive

$$\begin{aligned}&K_{CPU}.h_{OS}.h_{CA} \text{ says}\\ &\qquad (K_{CPU}.h_{OS}.h_{CA} \text{ says}\\ &\qquad\qquad (K_{CPU}.h_{OS}.h_{CA} \text{ says } \mathcal{F})),\end{aligned}$$

by (2.11) and (2.12) and definition (2.6) of $A \rightarrow B$; using SAYS-E (2.3) twice then obtains

$$K_{CPU}.h_{OS}.h_{CA} \text{ says } \mathcal{F}.$$

Sub-principals are also useful for creating different instances of a given principal, where each instance is accountable for the credentials issued during disjoint epochs or under the auspices of a different nonce or different circumstances. This allows the subset of credentials issued by some principal A at a time when you trust A to be distinguished from credentials issued by A at other times. So instead of using a single principal $FileSys$, we might employ a sequence $\mathit{FileSys}.1$, $\mathit{FileSys}.2$, ..., $\mathit{FileSys}.i$, ... of sub-principals, each accountable for issuing credentials during successive epochs. Then by specifying security policies that are satisfied only by credentials attributed to a "current instance" $\mathit{FileSys}.\mathit{now}$ (for now an integer variable), a guard can reject requests accompanied by outdated credentials.

SUBPRIN (2.10) allows any statement by a principal A to be attributed to any sub-principal of A. That is, from A $\textsf{says}$ $\mathcal{F}$ we could derive $A.\tau$ $\textsf{says}$ $\mathcal{F}$ for any sub-principal $A.\tau$. Unintended attributions can be avoided by adopting a sub-principal naming convention. We might, for example, agree to attribute to sub-principal $A.\epsilon$ any belief by A that should not be attributed to any other sub-principal $A.\tau$ of A.

2.3.2 Groups

A NAL *group* is a principal constructed from a set of other principals, called *constituents*, and is specified *intensionally* by giving a characteristic predicate. We write $\{\!\!\{\mathsf{v} : \mathcal{P}\}\!\!\}$ to denote the group defined by characteristic predicate $\mathcal{P}$; the group's constituents are those principals A for which $\mathcal{P}[\mathsf{v} := A]$ holds.[5] As an example,

$$\{\!\!\{\mathsf{v} : \mathit{Analyzer}\ \textsf{says}\ \texttt{numChan}(\mathsf{v}, \texttt{"TCP"}) < 3\}\!\!\}$$

is the group of all principals that $\mathit{Analyzer}$ has certified as having fewer than 3 open TCP connections.

The worldview of a NAL group is defined to be the union, deductively closed, of the worldviews for its constituents. Thus, if the worldview for one constituent of the group contains $\mathcal{F} \Rightarrow \mathcal{G}$ and another contains $\mathcal{F}$, then the group's worldview contains beliefs $\mathcal{F}$, $\mathcal{F} \Rightarrow \mathcal{G}$, and $\mathcal{G}$—even if the worldview for no constituent of the group contains $\mathcal{G}$.

Because the worldview of each constituent is a subset of the group's worldview, we

[5] $\mathcal{P}[\mathsf{v} := \mathit{exp}]$ denotes textual substitution of all free occurrences of v in $\mathcal{P}'$ by exp, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by renaming bound variables to avoid capture.

conclude for each constituent A of group G, that $A \rightarrow G$ holds. Thus, if $\mathcal{P}[\mathsf{v} := A]$ holds then $A \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}$ holds:

$$\text{MEMBER: }\frac{\mathcal{P}[\mathsf{v} := A]}{A \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}} \quad \text{free variables of } A \text{ are free for } \mathsf{v} \text{ in } \mathcal{P} \tag{2.13}$$

Note that $A \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}$ does not necessarily imply that $\mathcal{P}[\mathsf{v} := A]$ holds. In the absence of a NAL derivation for $\mathcal{P}[\mathsf{v} := A]$, we could still derive $A \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}$ from derivations for $A \rightarrow B$ and $B \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}$.

When $v \rightarrow A$ holds for all constituents v of a group with characteristic predicate $\mathcal{P}$ (i.e., all v satisfying $\mathcal{P}$), then all beliefs in the group's worldview necessarily appear in $\omega(A)$, so the group speaks for A:

$$\rightarrow \text{ GROUP: }\frac{(\forall \mathsf{v} : \mathcal{P} \Rightarrow (\mathsf{v} \rightarrow A))}{\{\!|\mathsf{v} : \mathcal{P}|\!\} \rightarrow A} \tag{2.14}$$

This inference rule, in combination with MEMBER (2.13), allows us to justify the following derived inference rule, which asserts groups and $\rightarrow$ are monotonic relative to implication:

$$\text{GROUP MONOTONICITY: }\frac{(\forall v : \mathcal{P} \Rightarrow \mathcal{P}')}{\{\!|\mathsf{v} : \mathcal{P}|\!\} \rightarrow \{\!|\mathsf{v} : \mathcal{P}'|\!\}}$$

Finally, note that NAL does not preclude *extensionally* defined groups, wherein constituents are simply enumerated. For example, $\{\!|\mathsf{v} : \mathsf{v} \in \{A, B, C\}|\!\}$ is the extensionally defined group whose constituents are principals A, B, and C.

2.3.3 Principals in Practice

α-Nexus implements specialized naming schemes for abstractions that serve as principals, including hardware, the kernel, processes, users, and various groups.

Some aspects of naming in α-Nexus are constrained by limitations of the TPM.[6] As described in Section 2.3.1, the NAL principal named by public key K_{CPU} denotes the hardware where the TPM holds a corresponding private key. Principal K_{CPU} represents all fixed hardware resources, including the TPM and a small amount of immutable code known as the *core root of trust*. The core root of trust is responsible for initiating the boot sequence and is largely isolated from all other software. The hardware's owner, manufacturer, or some other principal can issue credentials attesting to the properties of the hardware and core root of trust or to its trustworthiness. For instance, we might have a credential that conveys the formula:

$$\mathit{Manufacturer}\ \textsf{says}\ \texttt{tpm\_version}(K_{CPU}, 1.1).$$

This credential can be used to prove membership in the group

$$\{\!|\mathsf{v} : (\exists \mathsf{m}, \mathsf{i} : \mathsf{m} \in \{M_1, ..., M_n\} \ \wedge\ \mathsf{i} \geq 1.1 \ \wedge\ \mathsf{m}\ \textsf{says}\ \texttt{tpm\_version}(\mathsf{v}, \mathsf{i}))|\!\},$$

whose constituents are machines made by known manufacturers $M_1, ...M_n$ and having a TPM version 1.1 or greater.

Contrary to the simplified example of Section 2.3.1, α-Nexus does not compute a simple hash h_{OS} over the in-memory image of the operating system. That approach would not capture critical boot-time code and configuration data. α-Nexus instead relies on *authenticated boot* (often *trusted boot* or *secure boot*) [7] to obtain a distinct sub-principal name for each distinct kernel that executes on the hardware . The TPM implements authenticated boot by recording *measurements* (typically hashes) of boot-time code and configuration data. These measurements are stored in a small number of per-boot append-only logs called *platform configuration registers* (PCRs).[7] PCR values $\overline{h} = h_1, \ldots, h_n$, where there

[6] α-Nexus uses TPM version 1.1, a predecessor to the now commonly available TPM version 1.2. Both TPM versions implement substantially similar functionality, but the more recent TPMs have fewer resource limits and higher performance. Here, and throughout this dissertation, we omit or simplify many details of the TPM's operation, particularly the handling of cryptographic keys and signed messages. See [91] for an informal but thorough description of the TPM's operation.

[7] TPM measurement logs have a fixed size—each PCR is large enough to hold only a single hash value.

are n PCRs, uniquely identify the kernel's code and configuration, so we use $\overline{h}$ as part of the kernel's NAL sub-principal name: $K_{CPU}.\mathtt{pcrs}(\overline{h})$.

The TPM can issue a credential by signing a message m with the TPM's private key, and the resulting credential conveys K_{CPU} says $\mathcal{F}$, where $\mathcal{F}$ is the NAL interpretation of message m. The kernel can issue a credential by invoking the TPM's $\mathtt{quote}(m')$ operation, where m' is a kernel-supplied message encoding some NAL formula $\mathcal{F}'$. In response, the TPM uses its private key to sign a message $m = \langle \overline{h}, m' \rangle$, where $\overline{h}$ encodes the current PCR values. The resulting credential conveys K_{CPU} says $(K_{CPU}.\mathtt{pcrs}(\overline{h})$ says $\mathcal{F}')$, which implies $K_{CPU}.\mathtt{pcrs}(\overline{h})$ says $\mathcal{F}'$. Thus, as intended, $\mathcal{F}'$ is attributed to $K_{CPU}.\mathtt{pcrs}(\overline{h})$.

A naming scheme for the kernel should distinguish between different executions of the same kernel, to protect against different executions issuing contradictory credentials. This differentiation could be realized by a hardware-maintained counter that increases monotonically on each reboot. The TPM does not currently support this functionality.[8] α-Nexus instead employs a sequence of NAL sub-principals to denote different executions of the kernel. Specifically, the α-Nexus kernel maintains a variable p denoting the current *epoch*. This variable is stored on disk and incremented on each reboot. α-Nexus relies on TPM facilities to ensure that no other software can modify p at any other time. The kernel issues credentials under the auspices of NAL sub-principal $K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(p)$ by including p in the message passed to the TPM's $\mathtt{quote}(\cdot)$ operation. This naming scheme allows NAL formulas to refer to a specific execution of the kernel or to various groups of kernel executions. For example, the NAL group

$$\{\!\{\mathsf{v} : (\exists \mathsf{p} : 10 < \mathsf{p} < 20 \ \wedge \ \mathsf{v} \rightarrow K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}))\}\!\}$$

has as constituents the 10th through 20th boots of the specified kernel.

The TPM *extend* operation combines a new hash value with any existing hash value in a PCR. Distinct sequences of extend operations result in distinct PCR values.

[8]TPM Version 1.1 does not implement hardware counters, and TPM Version 1.2 has only limited support—counters are not mixed into the PCRs, and are not incremented automatically by the hardware on boot.

The α-Nexus kernel chooses a sub-principal name for each process. The only requirement is that each sub-principal name be unique, so we adopted a naming scheme that simply appends a unique process ID pid onto the kernel's name:

$$K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p).\texttt{process}(pid) \tag{2.15}$$

To ensure uniqueness, process IDs are never reused within a single execution of the kernel. The kernel implements a $\texttt{quote}(m'')$ system call that causes the kernel to issue a credential on behalf of a process,[9] where m'' encodes some NAL formula $\mathcal{F}''$. When a process with ID pid invokes this system call, the kernel constructs a message $m' = \langle p, pid, m'' \rangle$ and then invokes $\texttt{quote}(m')$ at the TPM, which in turn signs message $m = \langle \overline{h}, m' \rangle$ with the TPM's private key. The resulting signed message is a credential that conveys

$$\begin{array}{l} K_{CPU} \textsf{ says} \\ \quad (K_{CPU}.\texttt{pcrs}(\overline{h}) \textsf{ says} \\ \quad\quad (K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p).\texttt{process}(pid) \textsf{ says } \mathcal{F}'')), \end{array}$$

which implies

$$K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p).\texttt{process}(pid) \textsf{ says } \mathcal{F}''.$$

Processes, of course, are free to extend this naming scheme to define their own sub-principals and to implement a $\texttt{quote}(\cdot)$ operation to allow those sub-principals to issue credentials.

The NAL name of a process conveys little information about the process. This was a deliberate design choice: there are a variety of bases—axiomatic, analytic, and synthetic—on which trust in an α-Nexus process might be established. Any small set of information included in the sub-principal name, such as the hash of the program being executed by the process or the name of the program's publisher, would have been insufficient for some

[9]In fact, a process P can cause the kernel to issue a credential on behalf of any principal A for which $P \rightarrow A$. We discuss how in Section 2.4.4.

bases. So instead, the α-Nexus kernel issues credentials attesting to various attributes of processes.

At the request of an executing process, the α-Nexus kernel will issue a credential specifying the hash of a *program manifest* for that process. A program manifest is a description of the program being executed, the initial arguments for that program, and configuration data for the program. For a process P, if h_{pgm} is the hash of P's program manifest, then the credential issued by the kernel will convey the NAL formula

$$K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p) \textsf{ says } \texttt{pgm\_hash}(P, h_{pgm}).$$

Note that P in this credential would, in practice, be replaced by an expression like (2.15) encoding P's full NAL principal name. The kernel can similarly issue credentials attesting to the start time of a process and other properties about which the kernel is authoritative.

The α-Nexus kernel is not the only source for credentials that attest to attributes of processes. The α-Nexus networking stack, for example, runs as a process and issues credentials attesting to network usage by processes. And the $Login$ process, which manages α-Nexus user accounts and interactive user authentication, issues to each process P a credential that conveys:

$$Login \textsf{ says } P \rightarrow Login.\texttt{user}(uid). \tag{2.16}$$

According to SUBPRIN (2.10), from (2.16) we derive $P \rightarrow Login.\texttt{user}(uid)$. We define $Login.\texttt{user}(uid)$ to be the NAL representation of the user with ID uid, so credential (2.16) means that P speaks for that user.[10]

The credentials issued by the kernel and α-Nexus processes can be used to construct

[10]User IDs (as opposed to user names) are never reused in α-Nexus, even across reboots. This ensures the sub-principal names uniquely identify a single user.

narrowly-tailored NAL groups. Consider the group

$$\{\!\!|\mathsf{v} : (\exists \mathsf{p} : \mathsf{v} \rightarrow \mathit{Login}.\texttt{user}(\mathit{uid}_{\mathit{Alice}}) \quad \wedge$$
$$K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(\mathsf{p})\ \textsf{says}\ \texttt{pgm\_hash}(\mathsf{v}, h_{pgm}))|\!\!\},$$

where $\mathit{uid}_{\mathit{Alice}}$ is the user ID for user Alice. The constituents of this group are processes that satisfy four constraints: (i) the TPM-enabled hardware executing the process is associated with key K_{CPU}; (ii) the underlying kernel has code and configuration matching the PCR values $\overline{h}$; (iii) the process's program manifest has hash h_{pgm}; and (iv) the process speaks for Alice. The epoch number is unconstrained, so the constituents of this group span multiple executions of the kernel.

We often define groups in order to have stable names for services or sets of processes that span multiple kernel executions. For example, the Login principal does not refer to a specific process. Instead, it denotes a NAL group with constraints similar to (i)–(iii) above but specifying the hash h_{login} of a program manifest describing the α-Nexus login program.

2.4 Guards: Theory and Practice

The decision to authorize a request can be posed as a question about NAL formula derivation. We represent requests, credentials, and authorization policies as NAL formulas. A guard G that enforces an authorization policy $\mathcal{G}$ allows a request $\mathcal{R}$ to proceed if and only if

(i) G has a set of unforged credentials $c_1, c_2, \ldots, c_n$, where credential c_i denotes a bit-string that conveys NAL formula $\mathcal{C}_i$, and

(ii) G establishes that NAL can be used to derive $\mathcal{G}$ from

$$\mathcal{R} \wedge \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \cdots \wedge \mathcal{C}_n.$$

Credentials could be stored at the requesting principal, at the guard, or elsewhere in the system; and they could accompany a request, be fetched when needed by the guard, or be sent periodically to the guard. Notice that if request $\mathcal{R}$ is granted, then a NAL derivation of $\mathcal{G}$ to discharge (ii) documents the rationale for this authorization decision, indicating the role each credential plays. The NAL derivation of $\mathcal{G}$ is thus a form of audit log—and a particularly descriptive one, at that.

A filesystem guard $\mathit{FileSys}$ might, for example, enforce discretionary access control[11] for accessing a file (say) `foo` by employing authorization policy

$$\mathit{FileSys} \textsf{ says } \texttt{read}(\texttt{foo}) \tag{2.17}$$

and issuing a restricted delegation

$$A \xrightarrow{\texttt{read}(\texttt{foo})} \mathit{FileSys} \tag{2.18}$$

for any principal A whose requests A $\textsf{says}$ $\texttt{read}(\texttt{foo})$ should be allowed to proceed, since (2.18) allows (2.17) to be derived from A $\textsf{says}$ $\texttt{read}(\texttt{foo})$. Store the restricted delegation credential for a principal at that principal, and the result is reminiscent of capabilities; aggregate and store all of the restricted delegation credentials at the guard, and access control lists result.

2.4.1 Credential Distribution and Revocation

Whenever a guard G has some credential c that G determines is not forged, the NAL formula $\mathcal{C}$ conveyed by c is part of G's beliefs. G having credential c thus can be formalized in NAL as G $\textsf{says}$ $\mathcal{C}$.

One might hope that

$$(G \textsf{ says } \mathcal{C}) \Rightarrow \mathcal{C} \tag{2.19}$$

[11]This is not the only way to specify discretionary access control using NAL and guards.

would hold and, therefore, beliefs acquired through credentials are sound. This property, however, is not guaranteed. A principal P might issue c but then change beliefs (perhaps because the state or environment has changed) in a way that invalidates $\mathcal{C}$.

If P invalidates $\mathcal{C}$ after c has been distributed to other principals, then an access could be granted on *false pretense* because a guard has c and therefore believes that $\mathcal{C}$ holds even though $\mathcal{C}$ does not. Even if guards check the truth of each credential just before use, access might still be granted on false pretense—concurrent actions elsewhere in the system could falsify the formula conveyed by a credential after being checked.

For example, a request A $\textsf{says}$ $\texttt{access}(obj)$ accompanied by a credential that conveys $\mathit{TimeServ}$ $\textsf{says}$ $\texttt{clock} = 900$ suffices to derive the authorization policy:

$$A \textsf{ says } \texttt{access}(obj) \quad \wedge \quad \mathit{TimeServ} \textsf{ says } \texttt{clock} < 1000 \tag{2.20}$$

But

$$(G \textsf{ says } (\mathit{TimeServ} \textsf{ says } \texttt{clock} = 900)) \quad \Rightarrow \quad \mathit{TimeServ} \textsf{ says } \texttt{clock} = 900$$

does not hold if $\mathit{TimeServ}$ revises its beliefs whenever the passage of time causes its internal clock to increase. A guard that checks whether credentials are not forged and whether a NAL derivation exists for authorization policy (2.20) thus could grant access for requests made after $\texttt{clock} < 1000$ ceases to hold. So the guard could grant requests on false pretense.

There are two general strategies for ensuring that (2.19) will hold and, thus, prevent accesses from being granted on false pretense:

(i) Require that antecedent G $\textsf{says}$ $\mathcal{C}$ of (2.19) is false prior to changes in beliefs that invalidate its consequent $\mathcal{C}$.

(ii) Choose for consequent $\mathcal{C}$ of (2.19) a formula that cannot be invalidated by principals changing their beliefs.

With strategy (i), all principals G that have a credential c must delete their copies (thereby falsifying G says $\mathcal{C}$) before any principal is allowed to change its beliefs in a way that invalidates $\mathcal{C}$. This credential revocation is not a new problem[12] and is generally considered infeasible when credentials propagate in unrestricted ways. But a feasible realization of strategy (i), used extensively in α-Nexus, exists when the propagation of credentials is restricted in a rather natural way.

In α-Nexus, a principal is called an *authority* if it is both the sole source of certain credentials and it is the only principal that can invalidate a NAL formula conveyed by those credentials. We ensure that the only way that a principal can obtain a credential issued by an authority is directly from that authority. The authority thus knows which principals have its credentials. Prior to invalidating the belief conveyed in such a credential, the authority requests that these principals delete their copies and awaits confirmations (either from the principal or from α-Nexus asserting that the principal has been terminated).

To ensure that principals obtain credentials A says $\mathcal{F}$ directly from the issuing authority A, such credentials are represented in a way that conveys $\mathcal{F}$ and allows the recipient to attribute $\mathcal{F}$ to A, but does not allow any other principal to validate that attribution. For example, the α-Nexus inter-process communication (IPC) mechanism implements authenticated, integrity-protected channels (see the discussions below, in Sections 2.4.4 and 4.1.2, for details). Such a channel can speak for an authority A and transmit a formula $\mathcal{F}$ that only the single recipient at the end point of that channel can attribute to A. Authorities in α-Nexus distribute credentials using such channels. However, the same effect also could be achieved on an ordinary channel by using shared-key cryptography and message authentication codes.

Strategy (ii) for ensuring that (2.19) will hold requires restricting the execution of principals and/or choosing for $\mathcal{C}$ a sufficiently weak formula. System developers thus are made responsible for supporting revocation. Fortunately, system execution gener-

[12] It arises, for example, in connection with capabilities and with public-key certificates that describe name-key bindings.

ally does satisfy certain restrictions—time never decreases and the past is immutable, for example—not to mention restrictions coupled to the semantics of system and application functionality. So some truths do not change as execution proceeds, and this can be leveraged for defining NAL formulas $\mathcal{C}$ that cannot be falsified by future execution. For instance, once $\texttt{clock} > 1000$ holds, it cannot be later falsified by time advancing. And a credential attesting that some predicate $\mathcal{P}$ once held (even if it doesn't now hold) cannot subsequently be falsified if $\mathcal{P}$ holds when that credential is issued.

Imposing additional execution restrictions on principals is the other way to instantiate strategy (ii) for ensuring that (2.19) continues to hold. Suppose that, in order to authorize some request, a guard G requires A says $\mathcal{P}$ for $\mathcal{P}$ a state predicate, but that A says $\mathcal{P}$ could be invalidated from time to time.

- One solution is to prevent principals from invalidating $\mathcal{P}$ until some time in the future—in effect using a credential that conveys a form of lease [62]. For example,

  $$A \textsf{ says } (\texttt{clock} < 1000 \;\Rightarrow\; \mathcal{P}) \tag{2.21}$$

  is not falsified when time advances, so (2.19) will now hold. And if credentials

  $$\mathit{TimeServ} \textsf{ says } \texttt{clock} < 1000 \tag{2.22}$$

  $$\mathit{TimeServ} \xrightarrow{v\,:\,\texttt{clock}<v} G \tag{2.23}$$

  are available, then G can still conclude that A says $\mathcal{P}$ is satisfied. Moreover, if $\mathit{TimeServ}$ is implemented as an authority then we can ensure that credential (2.22) satisfies (2.19); and if delegation (2.23) is never disseminated outside of G, then it too will satisfy (2.19).

- An alternative to using leases is to have principals follow a synchronization protocol before invalidating $\mathcal{P}$. For example, we might postulate a lock $\ell_{\mathcal{P}}$ with two modes of access. Any number of principals can concurrently hold `shared` access, and a principal can hold `exclusive` access only if no other principal holds `shared` or `exclusive` access, with the following restrictions on execution:

(i) a guard acquires `shared` access to $\ell_{\mathcal{P}}$ before authorizing a decision using a credential involving $\mathcal{P}$, and the guard releases the lock afterward,

(ii) a principal acquires `exclusive` access to $\ell_{\mathcal{P}}$ before falsifying $\mathcal{P}$ and must reestablish $\mathcal{P}$ prior to releasing $\ell_{\mathcal{P}}$.

Then a credential conveying

$$A \textsf{ says } (\texttt{locked}(\texttt{shared}, \ell_{\mathcal{P}}) \Rightarrow \mathcal{P})$$

is never falsified even though $\mathcal{P}$ might be, so (2.19) holds. Moreover, if a guard G acquires $\ell_{\mathcal{P}}$ with `shared` access before making an authorization decision, so G has credentials

$$\mathit{LockMngr} \textsf{ says } \texttt{locked}(\texttt{shared}, \ell_{\mathcal{P}}) \tag{2.24}$$

$$\mathit{LockMngr} \xrightarrow{\texttt{locked}(\texttt{shared}, \ell_{\mathcal{P}})} G \tag{2.25}$$

attesting to $\texttt{locked}(\texttt{shared}, \ell_P)$, then G guard can conclude that A says $\mathcal{P}$ is satisfied at that time. Credentials (2.24) and (2.25) can be made to satisfy (2.19) if $\mathit{LockMngr}$ is implemented as an authority and (2.25) is never disseminated outside of G.

2.4.2 Credential Distribution in α-Nexus

We implemented facilities in α-Nexus to avoid relying on credentials that might convey invalidated beliefs.[13] These facilities can be used to implement several variations of the strategies described above.

[13]Nexus implements somewhat different interfaces and abstractions than we describe here for α-Nexus. These differences reflects the evolution of Nexus as experience was gained implementing and using the system.

α-Nexus Time Service

The α-Nexus kernel is an authority on the system clock, and therefore it can take the role of $\mathit{TimeServ}$ for credentials that convey time-based leases, like credential (2.21). The kernel is a natural place to implement this authority because the kernel speaks for all processes, hence for all guards. We can also implement other time-keeping services as processes. So while a credential from A conveying

$$A \textsf{ says } (\texttt{clock} < 1000 \Rightarrow \mathcal{P})$$

would prompt the guard to read `clock` from the kernel, a credential conveying

$$A \textsf{ says } ((\mathit{NetTimeServ} \textsf{ says } \texttt{network\_clock} < 1000) \Rightarrow \mathcal{P})$$

unambiguously states that the guard must contact a different principal, $\mathit{NetTimeServ}$, instead.

α-Nexus Label Store

The α-Nexus kernel *label store* is an authenticated, integrity-protected channel that distributes credentials. It provides an alternative to using cryptography or having guards contact authorities over IPC channels. And having such an alternative is useful. Cryptography is expensive. With IPC channels, authorities are processes, so they consume system resources. Moreover, when authorities are implemented by processes, the use of credentials could involve multiple IPC operations for distribution, validation, and revocation.

The label store is a list of entries. Each entry is identified by a unique ID lid, and each entry encodes a pair $\langle A, \mathcal{F} \rangle$, where A is a principal and $\mathcal{F}$ is a formula. The pair derives the credential A says $\mathcal{F}$ issued by some α-Nexus process or by the kernel itself. Process fetches the entry to obtain that credential. Label store entries can be deleted, and they do not persist across reboots of the kernel.

The label store distinguishes two types of credentials by keeping, along with each label store entry, a boolean-valued attribute r. If $r = \mathsf{true}$ holds then the entry represents a *revocable credential*, for which the label store functions as an authority. When a label store entry of this type is deleted from the label store, the credential it represents is also revoked. If $r = \mathsf{false}$ holds then the entry represents an *irrevocable credential*; the label store functions merely as a secure channel for distributing the credential with no provisions for revoking the credential.

Irrevocable label store credentials. For an irrevocable credential, the credential obtained by fetching the entry is never revoked. A process that fetches the entry is allowed to receive either a signed or unsigned copy of the credential. The necessary signatures are generated on demand, as described below. The recipient can safely store the returned copy of the credential indefinitely. And with signed credentials, the process can also forward the credential to other principals. Thus an irrevocable credential obtained from the label store can outlive the label store entry itself, and irrevocable credentials can be held by a process across reboots of the kernel.

The α-Nexus kernel implements the following system calls to support irrevocable credentials in the label store.[14]

- `label_say_irrevocable`$(A, \mathcal{F}, pf)$ creates a label store entry for an irrevocable credential and returns lid, the unique ID for the new entry. To ensure the label store can't be used to forge credentials, α-Nexus allows a process P to invoke system call `label_say_irrevocable`$(A, \mathcal{F}, pf)$ only if P has been authenticated as the principal A. In the simplest case, A is simply the NAL name for process P and the pf parameter is ignored. Other cases, which make use of the pf parameter, are discussed below, in Section 2.4.4. In all cases, $P \rightarrow A$ holds, allowing $\mathcal{F}$ to be attributed to principal A rather than just process P, hence the pair stored in the newly created

[14] α-Nexus enforces discretionary access control for all operations on label store entries. We omit details of label store authorization guards and policies.

entry is $\langle A, \mathcal{F} \rangle$. The invoking process P can specify any formula for parameter $\mathcal{F}$. In particular, $\mathcal{F}$ may convey a form of lease by including references to the α-Nexus system clock.

- `label_fetch_signed`(lid) returns a digitally signed credential for the label store entry identified by the unique ID lid. The kernel creates the credential by invoking the TPM's `quote`$(\cdot)$ operation. Thus the credential for an entry containing the pair $\langle A, \mathcal{F} \rangle$ actually conveys the formula:

$$K_{CPU}.\texttt{pcrs}(\overline{h}) \textsf{ says } (A \textsf{ says } \mathcal{F}). \tag{2.26}$$

  Because the process P that created the label store entry was authenticated as A (thus $P \to A$ holds) and P is a sub-principal of $K_{CPU}.\texttt{pcrs}(\overline{h})$, from (2.26) we can derive[15]

$$A \textsf{ says } \mathcal{F},$$

  which is the desired meaning of the label store entry. A process that invokes `label_say_irrevocable`$(\cdot,\cdot,\cdot)$ followed by `label_fetch_signed`$(\cdot)$ could also obtain an identical credential by invoking the α-Nexus `quote`$(\cdot)$ system call (discussed previously, in Section 2.3.3).

- `label_fetch_unsigned`(lid) returns an unsigned representation of formula A says $\mathcal{F}$, where $\langle A, \mathcal{F} \rangle$ is the pair stored in the label store entry identified by unique ID lid.

- `label_delete`(lid) deletes the label store entry identified by unique ID lid. This does not revoke the credentials obtained by processes that previously fetched the entry, but instead only reclaims kernel resources associated with the label store entry. Specifically, a process that obtained a credential using `label_fetch_unsigned`(lid)

[15]Here is that proof: Process P is a sub-principal of $K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p)$, which in turn is a sub-principal of $K_{CPU}.\texttt{pcrs}(\overline{h})$. So from SUBPRIN (2.10), applied twice, we derive $K_{CPU}.\texttt{pcrs}(\overline{h}) \to P$. Since $P \to A$ and $\to$ is transitive, we obtain $K_{CPU}.\texttt{pcrs}(\overline{h}) \to A$. Using definition (2.6) of speaks-for and SAYS-E (2.3) with (2.26), we derive A says $\mathcal{F}$.

or $\texttt{label\_fetch\_signed}(lid)$ may retain copies of the credential for the now-deleted entry.

In addition to $\texttt{label\_say\_irrevocable}(\cdot,\cdot,\cdot)$, several other α-Nexus system calls result in the creation of label store entries. These are used to represent various irrevocable credentials that the kernel itself issues, such as those described in Section 2.3.3. So when a process invokes the $\texttt{process\_attest\_hash}()$ system call, for example, the kernel issues a credential by creating an appropriate entry in the label store. In this case, the label store entry would contain the pair

$$\langle K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p)\ ,\ \ \texttt{pgm\_hash}(P, h_{pgm})\rangle,$$

where P is replaced by the full NAL name of the requesting process, h_{pgm} is replaced by the hash of that process's program manifest, both p and $\overline{h}$ are replaced by appropriate values for the current boot of the kernel, and K_{CPU} is the public key for the hardware TPM.

Revocable label store credentials. For an entry in the label store that represents a revocable credential, a processes that fetches the entry is provided only with an unsigned (hence, unforwardable) representation of that entry's credential. Here, the α-Nexus kernel supports credential revocation on behalf of the process that created the entry. Specifically, the kernel allows a label store entry to be deleted only after each process that could be holding a copy of the credential for that entry confirms that all such copies have been deleted. To implement this functionality, for each entry having $r = \textsf{true}$, the label store keeps an attribute $holders$ listing processes that have fetched the label store entry but have not yet confirmed deletion of the resulting credential.

The label store can also (optionally) serve as an authority for a lock that is used in a revocable credential represented by an entry in the label store. The kernel allocates and manages a lock for each such label store entry, and processes invoke system calls to acquire and release `shared` or `exclusive` exclusive access to these locks. Because the lock is stored with the label store entry, it is trivial for the kernel to delete the lock along with the entry when the entry's credential is revoked.[16]

The kernel implements the following system calls to support revocable credentials in the label store.

- `label_say_revocable`$(A, \mathcal{F}, l, \mathit{pf})$ creates a label store entry for a revocable credential and returns lid, the unique ID for the new entry. A process P is authorized to invoke `label_say_revocable`$(A, \cdot, \cdot, \cdot)$ only if $P \rightarrow A$ holds. As before, the process can specify any formula for $\mathcal{F}$, including formulas that convey leases. If the boolean-valued l parameter is true, then the label store entry will use locking and the kernel allocates a lock ℓ_{lid} for the new label store entry.
- `label_fetch_revocable`$(\mathit{lid}, \mathit{callback})$ returns an unsigned representation of the credential for the label store entry identified by the unique ID lid. If no lock ℓ_{lid} has been allocated (i.e., the l parameter was false when the label store entry was created), then the credential conveys

  $$A \textsf{ says } \mathcal{F},$$

  where $\langle A, \mathcal{F} \rangle$ is the pair encoded in the label store entry. Otherwise, there is a lock ℓ_{lid}, and the credential conveys the formula

  $$A \textsf{ says } (\texttt{locked}(\texttt{shared}, \ell_{\mathit{lid}}) \Rightarrow \mathcal{F}).$$

  Either way, the pair $\langle P', \mathit{callback} \rangle$ is inserted into the $\mathit{holders}$ list for that label store entry, where P' is identity of the process fetching the label store entry and $\mathit{callback}$ specifies an α-Nexus IPC channel to be used to contact that process.

[16]In principle, locking can be used with both revocable and irrevocable credentials. However, the label store only implements locking for revocable credentials. This limitation stems from the design decision to delete each lock when the corresponding label store entry is deleted. In contrast, an irrevocable credential might outlive the label store entry, so any lock an irrevocable credential uses would also need to outlive the label store entry.

- `label_unregister`$(lid, callback)$ removes the pair $\langle P', callback\rangle$ from the $holders$ list for the label store entry identified by unique ID lid, where P' is identity of the process invoking the system call. A process invokes this system call to notify the kernel that the process no longer holds a copy of the credential. Processes are not permitted to store revocable credentials between executions, so when a process P' exits, the kernel automatically removes any pair $\langle P', \cdot\rangle$ from the $holders$ list for every label store entry.

- `label_lock`$(lid, mode)$ acquires lock ℓ_{lid}, if that lock exists, where $mode$ specifies either `shared` or `exclusive`. The return value of this system call conveys the credential

$$K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p) \textsf{ says } \texttt{locked}(mode, \ell_{lid}).$$

  This credential is analogous to (2.24), but with the kernel, represented by principal $K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p)$, taking the place of $LockMngr$. An analog to (2.25) holds, because the kernel speaks for all α-Nexus processes.

- `label_unlock`(lid) releases a previously acquired lock ℓ_{lid}, if that lock exists and is held by the invoking process. Processes are not permitted to hold locks after they have exited, so when a process exits the kernel automatically releases any label store locks held by the process.

- `label_revoke`(lid) requests that the kernel revoke the credential for the label store entry identified by unique ID lid, then delete that label store entry. Lock ℓ_{lid} is also deleted, if that lock exists. To handle this request, the kernel first acquires `exclusive` access to ℓ_{lid} if that lock exists, then enqueues a notification message on each IPC channel listed in $holders$ for that label store entry. A process receiving such a notification will invoke `label_unregister`$(lid, \cdot)$ once any stored credential that process holds has been deleted. Thus, the kernel waits until $holders$ becomes empty before proceeding to delete the label store entry and any associated lock.

α-Nexus Guard Library

We implemented a library for constructing credentials-based authorization guards for α-Nexus applications and the kernel. An overview of the interface implemented by this *guard library* is given in Appendix B. The guard library provides a set of helper routines for obtaining, managing and validating sets of credentials, including routines for

- checking a cryptographic signature for a credential that is represented as a signed message;
- obtaining certain credentials from the α-Nexus kernel, given a description of the desired credential. Recall, the kernel serves as an authority for `clock` and certain accessible parts of the system state.
- obtaining a credential conveying A says $\mathcal{F}$ over an IPC channel from an authority that is an α-Nexus processes, given an IPC channel identifier, the principal A, and the desired formula $\mathcal{F}$; and
- obtaining credentials from the α-Nexus label store, given unique ID lid for a label store entry.

Credentials are obtained from the label store by invoking `label_fetch_unsigned`(lid) or, should that fail, `label_fetch_revocable`$(lid, \cdot)$. Credentials are obtained from an authority that is an α-Nexus process by sending a request containing the specified formula $\mathcal{F}$ to the specified IPC channel and then awaiting a response from the IPC channel. IPC channels in α-Nexus are authenticated, so a guard can verify whether the response was sent by the given principal A.

When obtaining credentials over an IPC channel or from the label store, the helper routines register a *callback* in the form of a α-Nexus IPC channel identifier that the source (e.g., an authority) can use to notify the guard library should a credential need to be deleted. In the event that a credential obtained from an authority becomes invalidated, the authority (e.g., the label store or some process) will enqueue a notification message to the IPC channel. Upon receiving such a message, the guard library deletes any stored copies of the credential and notifies the authority of this fact, either by sending a message to the IPC channel for that authority or by invoking the $\texttt{label\_unregister}(\mathit{lid}, \mathit{callback})$ system call, as appropriate.

The guard library helper routines include code to accommodate credentials that convey leases. Specifically, the helper routines check if a credential has the form

$$A \textsf{ says } (((\texttt{clock}\ \mathit{op_1}\ \mathit{val_1}) \land (\texttt{clock}\ \mathit{op_2}\ \mathit{val_2}) \land \ldots) \Rightarrow \mathcal{P}),$$

where each $\mathit{op_i}$ is one of $=$, $>$, $<$, etc., and each $\mathit{val_i}$ is an integer constant. The guard library checks that all conjuncts in the antecedent are satisfied for some pre-specified interval, e.g., at the time the guard library was invoked, or the interval from that time and spanning some specified duration.[17]

The guard library helper routines also include code to accommodate credentials that refer to locks. For a credential obtained by reading the label store entry with unique ID lid, the guard library checks if the credential has the form

$$A \textsf{ says } (\texttt{locked}(\texttt{shared}, \ell_{\mathit{lid}}) \Rightarrow \mathcal{F}).$$

If so, the guard library helper routines invoke $\texttt{label\_lock}(\mathit{lid}, \texttt{shared})$ to acquire the lock and $\texttt{label\_unlock}(\mathit{lid})$ to release the lock. For a credential obtained from some other authority A, the guard library checks if the credential has the form

$$A \textsf{ says } (\texttt{locked}(\texttt{shared}, \mathit{lock\_id}) \Rightarrow \mathcal{F}),$$

where $\mathit{lock\_id}$ is any constant term. If so, the guard library helper routines acquire and release the lock by sending an IPC requests containing $\mathit{lock\_id}$ to the IPC channel for that

[17]The guard library helper routines include code for automatically checking leases only for leases that refer to `clock`, which is the α-Nexus system clock.

authority.[18] When checking a set of credentials, the helper routines acquire all locks before the start of the pre-specified interval and release the locks after the end of the interval.

2.4.3 Sources of Derivations

Since NAL includes terms whose axiomatization is undecidable (e.g., integers or rich data structures), we cannot hope to build a universal guard—a guard that, for any choice of authorization policy $\mathcal{G}$ and set of credentials $\{c_1, c_2, \ldots, c_n\}$, derives $\mathcal{G}$ from $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ if and only if such a NAL derivation exists. This undecidability result, however, does not preclude building guards that automatically generate a NAL derivation for some particular authorization policy when given credentials in some pre-specified form. The filesystem example above illustrates this, because authorization policy (2.17) can be derived automatically from a request A says $\texttt{read}(\texttt{foo})$ when given a credential that conveys (2.18).

An alternative to having a guard perform the derivation of an authorization policy $\mathcal{G}$ would be to accompany each request with a NAL derivation of $\mathcal{G}$ [6, 11] or for the guard to solicit the derivation from trusted third parties [14]. In either case, the guard checks a NAL derivation rather than generating its own. This check is a decidable task, because NAL derivations are finite in length and inference rule applications are mechanically checkable.

α-Nexus supports guards by implementing a NAL proof checker. (Appendix B describes the interface and operation of this proof checker). In practice, a proof in α-Nexus contains two parts: a schema for a set of credentials, with each credential conveying a formula to be taken as a premise; and a schema for deriving from these premises a conclusion using the axioms and inference rules of NAL. Guards invoke helper routines in the guard library to obtain and validate the specified credentials and to check the proof schema; the guard allows a request to proceed only if the proof is correct and has the

[18]The guard library helper routines include code for automatically acquiring locks only for credentials from authorities, where the authority is assumed to manage the lock.

desired authorization policy $\mathcal{G}$ as its conclusion.

Having each request be accompanied by a derivation is not a panacea. For a principal to produce a derivation of an authorization policy $\mathcal{G}$, that principal must know what $\mathcal{G}$ is. Yet sometimes $\mathcal{G}$ must be kept secret so that, for example, various principals do not learn that different authorization criteria apply to each. Also, having each requester independently derive $\mathcal{G}$ makes changing $\mathcal{G}$ difficult, since principals that will submit requests to the guard must either be programmed to accommodate such changes (which might not be possible for the same reason that universal guards cannot exist) or must be found and manually updated whenever $\mathcal{G}$ is altered.

2.4.4 Authenticated Channels in α-Nexus

Thus far we have presumed authenticated channels, so that the recipient of a message containing statement $\mathcal{F}$ can correctly attribute $\mathcal{F}$ to the sending principal P. Thus the recipient adds belief P says $\mathcal{F}$ to its worldview. A straightforward way to implement authentication for IPC channels would be for the kernel to augment each IPC message with the process ID (or, equivalently, the NAL principal name) of the sender. And for the label store, we might allow process P to create label store entries only using its own NAL principal name. If a recipient has a credential for P says $\mathcal{F}$ and can separately derive $P \to A$, for some other principal A, then the recipient can derive A says $\mathcal{F}$. Thus P can, in effect, send IPC messages and create label store entries on behalf of principal A so long as $P \to A$.

We implemented a more flexible approach to authenticated channels in α-Nexus. For a process P, if $P \to A$ holds then the kernel allows P to send IPC messages on behalf of A directly, with each such IPC message augmented with a representation of NAL principal A rather than P. Similarly for the label store, if $P \to A$ holds then the kernel allows P to invoke `label_say_irrevocable`$(A, \mathcal{F}, pf)$ or `label_say_revocable`$(A, \mathcal{F}, l, pf)$ to insert a new label store entry on behalf of A. These system calls and those for sending IPC messages take the principal name A and a proof pf as parameters. On each such system call, the kernel invokes a guard with the specified proof to check the authorization policy $P \rightarrow A$. Thus, some work that would otherwise be performed by the recipient of the IPC message or label store credential (i.e., ensuring that P, the sender or issuer, actually speaks for the principal A they purport to speak for), is performed by kernel guards.

α-Nexus Alias Tables

Proofs (or sub-proofs) of $P \rightarrow A$, where P denotes an α-Nexus process and A is some principal, are common, because many requests that a process makes are on behalf of just one principal, such as a user, or at most a small set of principals. As a convenience and an optimization, α-Nexus keeps, for each process P, an *alias table*.[19] The ith entry of the alias table is a pair $\langle A_i, pf_i \rangle$, where principal A_i is called an *alias* for P and pf_i contains a proof schema and a schema for a set of credentials. The proof schema and the credentials purportedly constitute a proof of $P \rightarrow A_i$. However, because credentials obtained from an authority (i.e., over an IPC channel or from the label store) might be invalidated at any time, the kernel does not check that proof until alias A_i is used.

α-Nexus implements the following system calls in connection with the alias table abstraction.

- `authenticate_alias`(A, pf) inserts a new entry $\langle A, pf \rangle$ into the invoking process's alias table and returns the index of the newly created entry. Note that, unlike system calls to send IPC messages or create label store entries, pf need not be checked when the alias table entry is created.
- `drop_alias`(i) removes the ith entry from the invoking process's alias table.
- `lookup_alias`(P, i) returns either: the alias A_i from the ith entry in P's alias table

[19]Nexus discarded the alias table abstraction implemented by α-Nexus in favor of a *decision cache* abstraction, which caches a wider variety of information than is stored in the alias tables.

along with a proof of $P \to A_i$ that has been checked by the guard library; or an error code if the kernel is unable to construct such a proof.[20]

One way to implement `lookup_alias`(P, i) would be simply to invoke the guard library with pf_i from the i^{th} entry in P's alias table, returning either A_i or an error code based on whether pf_i is a correct proof of $P \to A_i$. Because this is a common operation, α-Nexus amortizes some of the cost of checking pf_i by performing some checks during `authenticate_alias`$(\cdot, \cdot)$, when an alias entry is first created. The checks it factors out are ones that cannot be invalidated by subsequent execution. This includes (i) checking the proof schema to ensure that it is well formed and has $P \to A_i$ as its conclusion, and (ii) validating the signatures on signed credentials. Checks that involve contacting authorities (e.g., to acquire locks) are delayed until `lookup_alias`$(\cdot, \cdot)$ is invoked.

Some α-Nexus kernel guards invoke `lookup_alias`$(\cdot, \cdot)$ when authorizing requests from processes. For instance, when process P sends an IPC message, rather than passing a principal A and a proof pf to the kernel, P can instead provide an index i into its alias table. The kernel invokes `lookup_alias`(P, i) and, if an alias A_i and a proof that $P \to A_i$ are returned, then the kernel augments the IPC message with an encoding of A_i to represent the sender. Thus, P avoids having to pass a principal A with each IPC send system call and avoids having to construct and pass proofs for each IPC send. Invoking `lookup_alias`(P, i) is also more efficient than performing a complete proof check of $P \to A$ on each IPC send.

The label store makes similar use of the alias table by allowing P to omit the A and pf parameters when creating a label store entry. Instead, P provides an index i into its alias table and the kernel invokes `lookup_alias`(P, i). Only if that call returns an alias A_i does the kernel proceed to create a label store entry using A_i as the principal. The same approach is implemented by the α-Nexus `quote`$(\cdot)$ system call.

[20]α-Nexus implements discretionary access control for `lookup_alias`$(\cdot, \cdot)$, but we omit the details of alias table authorization guards and policies.

Our impetus for creating the alias table abstraction was to optimize the handling of proofs commonly encountered when we built α-Nexus applications. An additional, though unintended, benefit was that the alias table allowed some applications to implement proof-carrying authorization with very little effort. Consider, for example, a filesystem guard that enforces discretionary access control by implementing an ACL (see *access control lists*, in Section 1.2.1). In a traditional implementation, principal names on the ACL might be encoded as strings (e.g., user names), and for each request the guard compares the name of the requester, again encoded as a string, against each name on the ACL. (A more complete implementation would require the guard to examine group membership as well.) By just replacing the string encodings with a encodings of NAL principal names, the filesystem guard can support some types of credentials-based authorization. If an ACL entry encodes a NAL group G, for example, a requesting process can invoke system call `authenticate_alias`(G, pf), then provide the resulting index into its alias table whenever invoking the filesystem over an IPC channel. Group G could equally well be any of the α-Nexus principals or groups discussed in Section 2.3.3. All of the work of validating credentials and checking proofs is done by the kernel; the filesystem guard needs only compare the principal name accompanying the IPC message with the names on the ACL. In fact, because NAL formulas can be represented as strings, a simple string comparison is sufficient for this purpose, just as in the traditional filesystem guard implementation.

2.5 Discussion

2.5.1 Genesis of NAL

Our original plan for α-Nexus was to adopt—not adapt—prior work in credentials-based authorization. The Lampson et al. [78] account (which introduced says and $\rightarrow$ operators) seemed to offer a compelling framework for the kinds of authorization α-Nexus was going to support, had been formalized by Abadi et al. [3] as a logic, and was used in the Taos operating system [147]. There was the matter of generating proofs and checking them—Taos had implemented only a decidable subset of the logic. Appel and Felten's [6] proof-carrying authentication addressed that, suggesting that all requests be accompanied by proofs and that guards perform only proof checking. Moreover, proof-carrying authentication employed a higher-order logic, so it supported application-specific predicates; and it was implemented in Twelf [112], so a proof checker was available.

A clear focus of this prior work was authentication for the varied and nuanced principals found in distributed systems. Operators to construct new principals (e.g., roles, quoting, etc.) were central to that enterprise. In α-Nexus, system state and properties of principals were going to be important inputs to authorization, too. We embarked on a series of design exercises to see how well those needs would be served by the prior work.

Our attempt to design a simple digital rights management system was particularly instructive. We sought flexibility in what should count as an access to the managed content (e.g., accessing any part versus accessing a majority versus accessing all of the content). A system designer would presumably record accesses by changing the system's state. So we concluded that a logic for credentials and authorization policies ought to include state predicates.

However, adding arbitrary state predicates to an authentication logic is subtle. If stand-alone state predicates can be formulas then inconsistencies would have far-reaching effects by allowing false to be derived, hence any authorization policy to be satisfied. We thus restricted state predicates to appearing only in worldviews of principals. Since it is unrealistic to expect that every principal could evaluate every state predicate or that different principals evaluating the same state predicate at different times would compute the same value, we needed a way for one principal to include in its worldview a state predicate $\mathcal{P}$ evaluated by some other principal.

- One approach [3, 6, 68] is to use SAYS-I (2.2) along with a new inference rule

$$\text{CNTRL:}\ \frac{A\ \textsf{says}\ \mathcal{P},\quad \textsf{controls}(A, \mathcal{P})}{\mathcal{P}}$$

  where $\textsf{controls}(A, \mathcal{P})$ holds if A is considered a trusted source regarding the truth of $\mathcal{P}$.

- The other approach [1, 2, 81] is to postulate a local-reasoning restriction and require that principals use delegation to import and reason about beliefs from others.

We rejected the first approach because it offers fewer guarantees about the propagation of inconsistencies, and it also requires characterizing sets of state predicates $\mathcal{P}'$ covered by $\textsf{controls}(A, \mathcal{P})$: if $\textsf{controls}(A, \mathcal{P})$ holds and $\mathcal{P} \Rightarrow \mathcal{P}'$ is valid then is A necessarily also trusted on $\mathcal{P}'$? Is A necessarily trusted on $\neg\mathcal{P}$?

CDD [1, 2], which had embraced a local-reasoning restriction and been subjected to careful analysis, then became an obvious candidate for the foundation of NAL. Moreover, CDD left unspecified details about principals and beliefs, so it offered us freedom to define principals that would match what α-Nexus provided and to use state predicates in beliefs (with theories that interpret these state predicates).

NAL sub-principals are derived from named roles in Alpaca [81]. Prior proposals (e.g., SDSI/SPKI [117] and Taos [147]) restrict the term τ used in defining a sub-principal $A.\tau$ to being a fixed string, which meant that only static roles could be supported. By allowing τ to be any term, the identity of a NAL sub-principal can be state-dependent.

Groups in NAL are a special case of the *dynamic unweighted threshold structures* defined by Delegation Logic [85]. And Delegation Logic was the first to suggest that group membership be specified intensionally, although no proof rules were given (nor were they needed) there. Our approach to authorization requires proof rules for satisfying authorization policies from credentials; with inference rules MEMBER (2.13) and $\rightarrow$GROUP (2.14), NAL appears to be the first logic for reasoning about such groups. The deductive closure semantics we selected for NAL groups was first proposed in [3] along with an axiomatization for extensionally defined instances of such groups.

Other semantics for groups have been proposed. With the *or-groups* of Syverson and Stubblebine [132], which are also supported in proof-carrying authentication [6], a belief is considered to be in the worldview of a group if and only if that belief is in the worldview of some[21] group member; or-groups are not sound with respect to IMP-E (2.4) and therefore would require different proof rules from other NAL principals. In groups with *conjunctive* semantics (sometimes called *conjunctive principals* [3,47,49,85] or *and-groups* [132]), a belief appears in the worldview of a group if and only if that belief appears in the deductive closure of the intersection of the worldviews for all members. We conjecture that conjunctional groups could be supported in NAL as the following abbreviation:

$$\langle\!\langle v : \mathcal{P} \rangle\!\rangle \textsf{ says } \mathcal{F}: \quad (\forall \textsf{v} : \mathcal{P} \Rightarrow (\textsf{v says } \mathcal{F}))$$

Finally, various proposals (e.g., [49] and [85]) have been made for groups that exhibit *t-threshold* semantics, whereby a belief is in the worldview of the group if and only if that belief is in the worldviews of at least t group members. This construct is quite expressive, difficult to axiomatize, and (fortunately) has not been needed for the applications we have explored.

We were not the first to see a need for state in an authentication logic. As soon as support for revocation or expiration of credentials is contemplated, the need for state-dependent credentials and policies becomes apparent. In Becker and Nanz [16], credentials and policies can have side effects that involve the addition or removal of assertions from the local rule base; Cassandra [17] represents state in terms of the activation and deactivation of roles; and linear logics [28, 54] encode state information in terms of how many times an axiom can be used. These encodings all duplicate in the logic state that al-

[21]Some authors refer to such as groups as implementing *disjunctive* semantics, but this term is used by other authors to describe groups that have the semantics defined by NAL, which also requires a deductive closure.

ready exists in a system. Expressiveness is often lost in the translation, preventing certain policies from being formalized. Moreover, in this prior work, either some sort of globally available state is being assumed, which becomes difficult to implement in a distributed system, or the state is local to a guard, which limits what authorization policies could be implemented.

2.5.2 Other Related Work

PolicyMaker [24, 26, 27] was the first authorization scheme to focus on considerations of trust as an input to authorization decisions.[22] Policies, credentials, and trust relationships are expressed in PolicyMaker as imperative programs in a safe language; a generic compliance checker interprets these programs to determine whether a policy is satisfied given the provided credentials and trust assumptions. REFEREE [35], designed to support Web applications, extends this approach by supporting policies about credential-fetching and signature verification; KeyNote [25] adds restrictions to make compliance checking efficient; and Delegation Logic [85] replaces PolicyMaker's imperative programs with D1LP, a monotonic version of Datalog that has declarative semantics and can be compiled into ordinary logic programs (e.g., Prolog).

SD3 [73], Binder [47], the RT family of logics [86], Cassandra [17], Soutei [113], and SecPAL [15] all employ languages based on Datalog; the result is a tasteful compromise between the efficient decision procedures that come with PolicyMaker's imperative programs and the declarative elegance of the Abadi et al. [3] access control calculus.

SecPAL, which targets grid computing environments and has also been used for authorization in a weakly-consistent peer-to-peer setting [148], is quite expressive despite limitations inherent in Datalog. It supports delegation credentials that are contingent on the evaluation of predicates over a guard's local state. And, unlike other authorization

[22]However, considerations about trust are the basis for the definitions of groups and roles in prior work on access control.

schemes based on logic programming, SecPAL allows negations of the form $\neg(A\ \textsf{says}\ \mathcal{F})$ to appear within policies (but not credentials); syntactic constraints on credentials and policies nevertheless guarantee policy checking is sound, complete, and always terminates, under the assumption (which unfortunately can be violated by a denial of service attack) that all credentials are available whenever a policy is evaluated. A tractable decision procedure for authorization was obtained by translating from SecPAL into a Datalog variant (viz. Datalog with Constraints).

DKAL [63] introduces a new dimension to credentials-based authorization by extending SecPAL to prevent any sensitive information carried in credentials and authorization policies from leaking, even when users that have different clearances share the same underlying authorization policies, database of credentials, and implementation.

Alpaca [81], like NAL, builds on proof-carrying authentication [6]. However, the domain of applications for Alpaca—unifying and generalizing public key infrastructures (PKIs) to support authentication—is quite different from NAL's goal of supporting authorization. And that explains differences in focus and function. Alpaca *authorities* (different from NAL authorities), for example, provide a structure to localize reasoning associated with a given logical theory; this turns out to be convenient in Alpaca for dealing with the mathematical operations and coercions used in authentication protocols. NAL and other logics that are dependent on signatures and hashes for attributing beliefs to principals, do not provide support for reasoning about these operations within the logic. Another important point of difference is that Alpaca—unlike NAL—has only limited support for stateful protocols. Nonces can be used in Alpaca to achieve one-use or limited-use credentials; there is no way, however, to use Alpaca for protocols that depend in general on history, as would be required (and is supported in NAL) for a digital rights management system or even as needed for implementing many authentication protocols.

Relatively few systems—most, research prototypes—support credentials-based authorization, but none do so in anything that approaches the generality needed for using analytic or synthetic bases in authorization. This prior work includes Taos and SecPAL, which were already mentioned; the W3C Web Services WS-Security [107] standard (in particular, WS-Policy [150]) is also rooted in this general approach, and that could bode well for the future. Bauer [11] used proof-carrying authorization for access control to Web pages. The Grey Project [12, 13] integrates a linear logic and proof-carrying authentication on a smart phone platform, and it has been used for authorizing access to offices and shared labs. And Howell and Kotz [69] implemented a credentials-based approach for use within and between applications running in Java virtual machines; that logic is an extension of SPKI [49].

CHAPTER 3

A DOCUMENT-VIEWER SUITE USING NAL

We designed and implemented on α-Nexus three document viewer applications in order to evaluate the use of NAL for enforcing authorization policies that concern integrity and confidentiality of electronic documents. This exercise informed the design of NAL as well as showing that surprisingly broad functionality can be seen as forms of authorization if one considers authorization in terms of predicting trustworthiness of principals.

In each viewer application, we posit a one-to-one correspondence between documents and principals. The principal for the document to be displayed—not the human user viewing the document—is the principal whose requests are authorized by a guard. This unconventional set-up allows us to benefit from analytic and synthetic bases for authorization. Had the viewer applications instead been designed to process requests from human users wishing to view documents, then we would have been limited to employing an axiomatic basis for authorization, since humans are hard to analyze and do not take kindly to transformations.

3.1 $\mathit{TruDocs}$: Analytic and Axiomatic Bases for Authorization

$\mathit{TruDocs}$ is a document-viewer application that ensures excerpts attributed to a document are consistent with policies that document specifies. We start with the observations that documents convey beliefs and that excerpts derived from a document also convey beliefs. For d_i some document, NAL provides a natural way to formalize which beliefs d_i conveys. We identify d_i with a principal $\mathit{Prin}_{doc}(d_i)$ and write a NAL formula $\mathit{Prin}_{doc}(d_i)$ $\textsf{says}$ $\mathcal{P}$ for each belief $\mathcal{P}$ that d_i conveys.

We represent an excerpt e appearing in a document d as a 4-tuple $e = \langle \chi, d, l, d' \rangle$, where χ is the text of the excerpt, d' is a *source* document to which the excerpt is being attributed, and l is the location where the excerpt appears in d. Notice, distinct appearances of text

χ in d are considered to be different excerpts. As with documents, each excerpt e_i can be identified with a NAL principal $Prin_{ex}(e_i)$, where $Prin_{ex}(e_i)$ says $\mathcal{P}$ holds for every belief $\mathcal{P}$ that except e_i conveys. Define $src(e)$ to be the source document (i.e., d' above) from which e was purportedly derived; $Prin_{doc}(src(e))$ is therefore the principal corresponding to $src(e)$.

The reader of an excerpt e and the author of source document $src(e)$ would expect that beliefs conveyed by e are also conveyed by $src(e)$: $\omega(Prin_{ex}(e)) \subseteq \omega(Prin_{doc}(src(e)))$ or equivalently $Prin_{ex}(e) \rightarrow Prin_{doc}(src(e))$ holds. But whether $Prin_{ex}(e) \rightarrow Prin_{doc}(src(e))$ actually holds will depend on how e was derived from $src(e)$. Quoting too few words, quoting out of context, redaction, elision of words and clauses, all can produce an "excerpt" that conveys different beliefs than are conveyed in the source. We define a document d to have *integrity* if and only if, for every excerpt e appearing in d, the beliefs e conveys are also conveyed by $src(e)$. This property can be formalized in NAL as a credential

$$TruDocs \text{ says } (\forall \texttt{e}\colon \texttt{e} \in d \Rightarrow (\texttt{Prin\_ex(e)} \rightarrow \texttt{Prin\_doc(src(e)))}) \tag{3.1}$$

that $TruDocs$ issues about d, where relation $e \in d$ holds if and only if document d contains excerpt e.[1] The principal-valued function $\texttt{Prin\_ex}(e)$ and document-valued function $\texttt{src}(e)$ appearing in this credential are defined, for any excerpt e, to be equal to $Prin_{ex}(e)$ and $src(e)$, respectively. Similarly, the principal-valued function $\texttt{Prin\_doc}(d)$ is defined, for any document d, to be equal to $Prin_{doc}(d)$. For instance, given an excerpt e for which $src(e) = d'$, then we have

$$TruDocs \text{ says } \texttt{src}(e) = d',$$

or more generally we have

$$TruDocs \text{ says } (\forall \mathsf{x}, \mathsf{d}, \mathsf{l}, \mathsf{d'}\colon \texttt{src}(\langle \mathsf{x}, \mathsf{d}, \mathsf{l}, \mathsf{d'} \rangle) = \mathsf{d'}),$$

since each excerpt e is represented by some 4-tuple $\langle \chi, d, l, d' \rangle$.

[1]Definition (3.1) treats nested excerpts as if each appears directly in d. Other treatments are possible.

The author of a document d' cannot be expected to enumerate all possible excerpts e that convey beliefs found in d'. So authors (or the organizations they work for) associate *use policies* with documents they produce. To be eligible for inclusion in another document d, an excerpt e must comply with the use policy associated with $src(e)$. $TruDocs$ limits use policies to those that can be specified as syntactic criteria or as other computable checks whose compliance implies $Prin_{ex}(e) \rightarrow Prin_{doc}(d')$, meaning the beliefs expressed by excerpt e are from document d'.

We can associate a use policy with a source document d' by issuing a credential that conveys the NAL formula

$$\begin{aligned} Prin_{doc}(d') \textsf{ says } (\forall \texttt{e}:\ & (Prin_{doc}(d') = \texttt{Prin\_doc}(\texttt{src}(\texttt{e})) \wedge usePol_{d'}(\texttt{e})) \\ & \Rightarrow\ (\texttt{Prin\_ex}(\texttt{e}) \rightarrow Prin_{doc}(d'))) \end{aligned} \tag{3.2}$$

where $usePol_{d'}(e)$ is a predicate satisfied if excerpt $e = \langle \chi, d, l, d' \rangle$ appearing in d is consistent with the use policy associated with d'. The first conjunct in the antecedent of (3.2), $Prin_{doc}(d') = \texttt{Prin\_doc}(\texttt{src}(\texttt{e}))$, restricts the quantification to those excerpts e that purport to come from d'.[2] Credentials like (3.2) enable (3.1) to be derived by checking each excerpt e in a document d against the use policy for $src(e)$:

$$TruDocs \textsf{ says } usePol_{src(e)}(e) \tag{3.3}$$

Thus, a guard handling a request for the display of a document d can mechanically derive (3.1) or, conversely, deny a display request if d does not have integrity. Note that (3.3) is an analytic basis for trust because authorization depends on a form of analysis: checking a use policy.

$TruDocs$ can also handle copyright's "fair use" and other non-computable use policies by employing an axiomatic basis for trust. One or more human authorities H_i for which

[2]Under the assumption that the principals identified with documents are unique, that conjunct is equivalent to $d' = \texttt{src(e)}$. But the conjunct used in (3.2) has the benefit that only $Prin_{doc}(d')$ need appear in the credential rather than d'. As will become clear below, $Prin_{doc}(d')$ is a concise name for the document whereas d' contains the actual text of the document.

$\mathit{TruDocs}$ has issued a credential that conveys

$$\mathit{TruDocs}\ \textsf{says}\ (H_i \rightarrow \mathit{TruDocs}) \tag{3.4}$$

are solicited to check the use policy for each excerpt e. H_i in turn provides credentials that convey

$$H_i\ \textsf{says}\ (\mathit{Prin}_{ex}(e) \rightarrow \mathit{Prin}_{doc}(\mathit{src}(e))) \tag{3.5}$$

only if the use policy is satisfied for excerpt e. Receipt of such a credential for each excerpt e in d is all that is needed for $\mathit{TruDocs}$ to derive (3.1). So this approach corresponds to deriving (3.3), where $\mathit{usePol}_{\mathit{src}(e)}(e)$ is satisfied if and only if $\mathit{TruDocs}$ has credentials (3.4) and (3.5).

Implementation Details

$\mathit{TruDocs}$ comprises an editor TDed for use by document authors, a viewer TDview for displaying documents, and some additional support software.

- TDed allows a document d that contains excerpts to be created, enables a use policy to be defined and associated with that document, and constructs a unique name $\mathit{nme}(d)$ for the document. By construction, $\mathit{nme}(d)$ embodies a validated set of *document particulars*, such as title, author, publication venue, publication date, etc.
- TDview implements a guard to authorize display requests from documents; a display request for d is granted only if (3.3) can be derived for each excerpt e in d, since (3.1) can then be derived from that. Whenever TDview displays a document, it displays after each excerpt e the document particulars embodied in $\mathit{nme}(\mathit{src}(e))$, thereby giving the reader a human-intelligible description for the source document from which e was derived.

TDed and TDview were obtained by modifying the OpenOffice software suite [105]. We added 739 lines of Visual Basic code and 5066 lines of C code to OpenOffice. TDed

and TDview also use the NAL guard library (an additional 4677 lines of C code) and third party libraries: OpenSSL [106] (for hashing, signature generation, and signature verification) and XOM [153] (for document manipulation and canonicalization). Because α-Nexus does not yet support sophisticated user interfaces, TDview was implemented as two separate components. One component is trusted and runs on the α-Nexus kernel; it executes the NAL proof checker and an analysis engine. The other component is not trusted and runs on a Linux platform; it displays information and implements the user interface. TDed runs primarily on an untrusted Linux platform.

The TDview guard enumerates the excerpts in d and processes each excerpt e as follows.

(i) Determine the predicate $\mathit{usePol}_{\mathit{src}(e)}(e)$ that applies for excerpt e.

(ii) Check $\mathit{usePol}_{\mathit{src}(e)}(e)$ and, if it holds, issue

$$\mathit{TruDocs}\ \textsf{says}\ \mathit{usePol}_{\mathit{src}(e)}(e). \tag{3.6}$$

Step (ii) is implemented with assistance from the NAL proof checker and built-in support for text matching, as follows:

- TDview checks to see if the display request was accompanied by credentials like (3.5) from some human authority H_i and/or a NAL proof that discharges (3.6), and if so, TDview checks that proof, issuing a credential conveying (3.6) if the proof is correct;
- if not, TDview determines if it has built-in support to validate $\mathit{usePol}_{\mathit{src}(e)}(e)$, attempts that validation, and if successful TDview issues a credential conveying (3.6);
- otherwise, TDview displays an error message that details the use policy that it could not satisfy, requesting additional credentials and/or a NAL proof be provided.

TDview currently supports only digitally signed credentials—credentials are not read from the α-Nexus label store, for example—and credentials must be written using a fragment of NAL (described below).

Note that some trust assumptions are required, because of NAL's local-reasoning restriction. First, $\mathit{TDview} \rightarrow \mathit{TruDocs}$ must be assumed, so that credentials issued by TDview can contribute to the derivation of (3.6), a statement being attributed to $\mathit{TruDocs}$. TDview is an α-Nexus process, and the kernel issues a credential

$$K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(p) \text{ says } \mathtt{pgm\_hash}(\mathit{TDview}, h_{\mathit{TDview}}),$$

where h_{TDview} is the hash of TDview's program manifest. Thus assumption $\mathit{TDview} \rightarrow \mathit{TruDocs}$ can be discharged if we take $\mathit{TruDocs}$ to be

$$\{\!|\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \text{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{\mathit{TDview}}))|\!\},$$

making every TDview process executing on the kernel given by platform configuration register values $\overline{h}$ and running on the machine having a Trusted Platform Module (TPM) with public key K_{CPU} a constituent of $\mathit{TruDocs}$. Alternatively, we could take $\mathit{TruDocs}$ to be a public key K_{TruDocs} chosen for this purpose by the user or an administrator; we would then have to arrange for the distribution of signed credentials that convey

$$K_{\mathit{TruDocs}} \text{ says } \mathit{TDview} \rightarrow K_{\mathit{TruDocs}}$$

for each execution of TDview, or

$$K_{\mathit{TruDocs}} \text{ says } \{\!|\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \text{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{\mathit{TDview}}))|\!\} \rightarrow K_{\mathit{TruDocs}}$$

to capture all TDview instances in a single credential.

A second trust assumption we require is that for each credential H_i says $\mathcal{F}$ provided by an human authority H_i and used in step (ii), there must be a credential

$$\mathit{TDview} \text{ says } (H_i \rightarrow \mathit{TDview})$$

that signifies H_i is trusted by TDview and, therefore, TDview says $\mathcal{F}$ can be derived by

TDview from H_i $\textsf{says}$ $\mathcal{F}$. The name of each such trusted human authority H_i is communicated to TDview at startup.[3]

Limits in on-line storage or concerns about confidentiality are just two reasons TDview might not have access to certain source documents. So TDview is not always able to validate $\mathit{usePol}_{\mathit{src}(e)}(e)$ directly and might instead have to import credentials from human or other external authorities. In fact, having TDview import credentials can improve performance by undertaking an expensive analysis once rather than each time a document display is requested. For example, when creating a document d, TDed has access to all documents from which excerpts appearing in d are derived. TDed is therefore an obvious place to perform some analysis and issue credentials that later aid TDview in deriving (3.6). TDed invokes $\mathit{TDanalyze}$, an analysis engine that runs on α-Nexus; $\mathit{TDanalyze}$ performs some or all of the required analysis and invokes the α-Nexus $\texttt{quote}(\cdot)$ system call to issue a credentials attesting to the results of this analysis, and then TDed stores these credentials with the document. This implementation does require an additional trust assumption: $\mathit{TDanalyze} \rightarrow \mathit{TDview}$. $\mathit{TDanalyze}$ might execute on a different machine than TDview, and only some machines may be trusted, so we discharge this assumption using a credential of the form

$$\begin{aligned}\mathit{TDview}\ \textsf{says}\ \{\!\!|\mathsf{v} : (\exists \mathsf{k}, \mathsf{p}:\ & \texttt{certified\_hardware}(\mathsf{k}) \wedge \\ & \mathsf{k}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(\mathsf{p})\ \textsf{says}\ \texttt{pgm\_hash}(\mathsf{v}, h_{\mathit{TDanalyze}}))|\!\!\} \rightarrow \mathit{TDview},\end{aligned} \tag{3.7}$$

where $h_{\mathit{TDanalyze}}$ is the hash of the program manifest for $\mathit{TDanalyze}$ and the predicate $\texttt{certified\_hardware}(K)$ holds only if K is the public TPM key for a trusted machine.

[3]In *TruDocs*, integrity policies are meant to protect the interests of the reader of a document, and these policies are enforced at the reader's discretion. Documents are stored in plain text, for example, and a reader can examine those documents using some other viewer. Thus, it is reasonable for TDview to issue credentials of the form TDview $\textsf{says}$ $(H_i \rightarrow \mathit{TDview})$ at the request of the reader. This is in contrast to a MAC policy (see *mandatory access control*, in Section 1.2.2) or digital rights management systems, which aim to protect the interests of other principals, such as copyright holders, system administrators, or the reader's employer.

`excerpts`(d)	—The set of all excerpts in document d (i.e., excerpts e such that $e \in d$ holds).
`nme`(d)	—The validated document particulars for document d, encoded as a set of key-value pairs. Keys are strings such as `"title"`, `"author"`, `"publication date"`, etc., and values are strings.
`body`(e)	—The text of excerpt e.
`src`(e)	—The source document from which excerpt e purports to have been derived.
`dst`(e)	—The document in which excerpt e appears.
`numWords`(χ)	—The length in words of text χ.
`txtSrch`(χ_1, χ_2, m)	—The text χ_1 appears within text χ_2 modulo m. Here, m is a subset of { `"ignore case"`, `"ignore whitespace"`, `"allow eliding"`, `"allow editorial inserts"`, ...}.

Figure 3.1: A subset of *TruDocs* credential and use policy terms.

TruDocs relies on a *platform authority* (named by public key K_{PA}), which issues a credential that conveys

$$K_{PA} \text{ says } \texttt{certified\_hardware}(K)$$

for each trusted machine's TPM key K. This requires a further trust assumption,

$$\mathit{TDview} \text{ says } K_{PA} \xrightarrow{\texttt{k: certified\_hardware(k)}} \mathit{TDview}.$$

The identity of the trusted platform authority is communicated to *TDview* at startup.

Use policies enforced by *TDview* and (partially) checked by *TDanalyze* are currently written in a fragment of NAL that includes simple connectives ($\wedge$, $\vee$, and $\neg$), a variety of arithmetic and relational operators ($+$, $*$, $>$, $<$, $\in$, etc.), and the terms shown in Figure 3.1. Our prototype implements only what is required to support a few example use policies; a complete implementation would likely support a much richer policy language. For instance, one might include support for invoking automated sentiment analysis [109, 139]. Still, our prototype's policy language is rich enough for *TDview* and *TDanalyze* to support: matching an excerpt and source text verbatim or allowing for change of case, replacing fragments of text by ellipses, inserting editorial comments enclosed within square brackets, and limiting the length of individual excerpts or the aggregate length or number of the excerpts from a given document. For instance, consider the use policy $usePol_{d'}(e)$ defined by

$$\begin{aligned}&\texttt{txtSrch}(\texttt{body}(e), \texttt{body}(d'), \{\texttt{"ignore case"}\})\\&\quad\wedge \texttt{numWords}(\texttt{body}(e)) \geq 20\\&\quad\wedge \texttt{numWords}(\texttt{body}(e)) * 4 < \texttt{numWords}(\texttt{body}(\texttt{dst}(e))).\end{aligned} \tag{3.8}$$

For excerpt e to be attributed to source document d', this use policy requires that the text of e match some text in d' (ignoring case), have at least 20 words, and constitute less than one quarter of the document in which e appears. *TDview* and *TDanalyze* also can validate compliance with a use policy that stipulates excerpts not appear in documents having certain document particulars—for example, that excerpts not appear in documents authored by a given individual or published in a given venue. So the author of d' in the above example might augment policy (3.8) with a restriction

$$\langle \texttt{"author"}, A\rangle \notin \texttt{nme}(\texttt{dst}(e)).$$

A name $nme(d)$ that lists document particulars would prove problematic if we want to use an ordinary filesystem and store d as a file named $nme(d)$. So *TruDocs* associates with each document d a principal named $Hnme(d)$, as follows. Each document d is represented in extensible markup language (XML), and we define $Hnme(d) = H(x_d)$ where x_d is the XML representation (using the DocBook [48] standard) for d and where $H(\cdot)$ is a SHA1 hash. $Hnme(d)$, because it is relatively short, can serve as the name for a file storing x_d in a filesystem or Web server. For each excerpt e appearing in d , *TruDocs* stores in x_d name $nme(src(e))$, which provides the document particulars for $src(e)$, and name $Hnme(src(e))$, which provides direct access to the file storing $x_{src(e)}$.[4]

[4]If only name $Hnme(src(e))$ were stored in x_d, then after d has been created, an attacker could change what is stored in file $Hnme(src(e))$, thereby invalidating the consistency of the information from

A binding between principals $Hnme(d)$ (i.e., $H(x_d)$) and $nme(d)$ is made by $TruDocs$ principal Reg (named by public key K_{Reg}); Reg runs on a separate machine from $TDed$ and $TDview$. Reg creates bindings, validates document particulars, and disseminates the existence of $Hnme(d)$ to $nme(d)$ bindings by issuing credentials. In particular, a document d created with $TDed$ becomes eligible for viewing only after the user invokes the `publish` operation; `publish` causes pair $\langle x_d, nme(d) \rangle$ to be forwarded to Reg, which checks that

(i) $nme(d)$ is unique,

(ii) $nme(d)$ is consistent with document particulars (e.g., author, title, publication venue, publication date) conveyed in x_d, and

(iii) each document particular in $nme(d)$ is valid according to relevant external authorities (e.g., the authoritative reprints repository maintained by the journal where d is purported to have been published).

If (i)–(iii) hold, then $nme(d)$ is considered validated and Reg issues a credential

$$K_{Reg} \ \textsf{says} \ (Hnme(d) \rightarrow K_{Reg}.nme(d)), \tag{3.9}$$

which is returned by Reg to $TDed$, where it is piggybacked[5] on x_d. Notice that if we define $Prin_{doc}(d)$ to be $K_{Reg}.nme(d)$, we can derive

$$Hnme(d) \rightarrow Prin_{doc}(d), \tag{3.10}$$

a binding between $Hnme(d)$ and $Prin_{doc}(d)$: from (3.9), SUBPRIN (2.10) derives $Hnme(d) \rightarrow K_{Reg}.nme(d)$ and then use the above definition of $Prin_{doc}(d)$ to substitute for $K_{Reg}.nme(d)$.

The principal name for an excerpt, $Prin_{ex}(e)$, never appears explicitly in credentials, and it is never shown to the reader of a document. So here, there is no need for more than a simple hash. For the excerpt $e = \langle \chi, d, l, d' \rangle$, we define $Prin_{ex}(e)$ to be $H(\langle \chi, d, l, d' \rangle)$, where $H(\cdot)$ is again a SHA1 hash.

$nme(src(e))$ that gets displayed at the end of e with the document particulars for $src(e)$.

[5]Credential (3.9) cannot be stored in x_d, because that would change name $H(x_d)$ for that principal, rendering credential (3.9) useless.

Finally, as noted above, when TDed creates a document d', it stores a use-policy credential as part of $x_{d'}$. The credential stored is actually a variant of (3.2), now that two different principals are associated with each document:

$$\begin{aligned} &\mathit{Hnme}(d') \textsf{ says } (\forall \mathtt{e}\colon\ (\mathit{Hnme}(d') = \mathtt{sha1}(\mathtt{src}(\mathtt{e}))) \wedge \mathit{usePol}_{d'}(\mathtt{e})) \\ &\qquad\qquad \Rightarrow\ (\mathtt{Prin\_ex}(\mathtt{e}) \rightarrow \mathit{Hnme}(d'))), \end{aligned} \tag{3.11}$$

where $\mathit{TruDocs}$ says $\mathtt{sha1}(d) = h$ if and only if h is the SHA1 hash of document d. For excerpt e, $\mathtt{Prin\_ex}(e) \rightarrow \mathit{Hnme}(d')$ derives $\mathtt{Prin\_ex}(e) \rightarrow \mathit{Prin}_{doc}(d')$, since (3.10) can be derived from the instance of (3.9) piggybacked on $x_{d'}$. This means that from (3.9) and (3.11), TDview can always automatically derive:

$$\begin{aligned} &H(x_{d'}) \textsf{ says } (\forall \mathtt{e}\colon\ ((\mathit{Hnme}(d') = \mathtt{sha1}(\mathtt{src}(\mathtt{e}))) \wedge \mathit{usePol}_{d'}(\mathtt{e})) \\ &\qquad\qquad \Rightarrow\ (\mathtt{Prin\_ex}(\mathtt{e}) \rightarrow \mathit{Prin}_{doc}(d'))) \end{aligned} \tag{3.12}$$

And the NAL derivation of (3.1) from (3.12) is virtually the same as the derivation of (3.1) from (3.2), again remaining independent of document d and thus not something the guard of TDview must regenerate to authorize each display request.

3.2 *ConfDocs*: A Synthetic Basis for Authorization

ConfDocs implements multilevel security (MLS) [111,143] for accessing documents comprising *text elements*. The policy that *ConfDocs* enforces for each text element is similar to the MLS policy described in Section 1.2.2. Each text element χ in a document is assigned a *classification* by some trusted *classification authority*; each human user H is assigned a *clearance* by some trusted *clearance authority*. Classifications and clearances are selected from a set of *security labels* on which a partial order relation $\preceq$ has been defined.

Each document d is identified with a unique principal $\mathit{Prin}(d)$. A document d comprising a set $txt(d)$ of text elements is authorized for display to a user H if and only if

$$Prin(d) \textsf{ says } (\forall \mathsf{x} : \mathsf{x} \in txt(d) \Rightarrow \texttt{cls}(\mathsf{x}) \preceq \texttt{clr}(H)) \tag{3.13}$$

holds, where $\texttt{cls}(\chi)$ denotes the classification assigned to text element χ, and $\texttt{clr}(H)$ denotes the clearance assigned to user H. Policy (3.13) makes d—or, rather, the publisher of d—the ultimate authority on which users can read d, by leaving the choice of classification authority and clearance authority with d. In particular, the choice of classification authority determines the value of $\texttt{cls}(\chi)$ and the choice of clearance authority determines the value of $\texttt{clr}(H)$, so these choices (albeit indirectly) effect whether H satisfies (3.13).

ConfDocs is agnostic about the set of security labels and partial order relation $\preceq$. The system simply requires the means (internally built-in or by appeal to an external authority) to determine whether $\ell \preceq \ell'$ holds for any pair of security labels ℓ and ℓ'. *ConfDocs* has built-in support for security labels structured as pairs [44, 120], comprising a sensitivity level and set of compartments, as described in Section 1.2.2.

If a document d does not satisfy authorization policy (3.13) for a given user H, then it is often possible to derive a document that does.

- Deleting text from d narrows the scope of the universal quantification in (3.13) by removing a text element χ from $txt(d)$, thereby eliminating an obligation $\texttt{cls}(\chi) \preceq \texttt{clr}(H)$ that could not be discharged.
- Modifying d (say, by changing certain prose in a text element χ to obtain χ') could change the contents of $txt(d)$ in a way that transforms an obligation that could not be discharged (i.e., $\texttt{cls}(\chi) \preceq \texttt{clr}(H)$) into one that can be (i.e., $\texttt{cls}(\chi') \preceq \texttt{clr}(H)$).

Each implements a synthetic basis for authorization, and our *ConfDocs* prototype supports both.

Implementation Details

ConfDocs provides a program *CDview* for viewing documents and provides some shell scripts for creating and managing documents. *CDview* is 5787 of C code that runs on α-Nexus and uses the NAL guard library and proof checker. Shell scripts (175 lines of Bash) that invoke the OpenSSL library to perform encryption allow a user (as detailed below) to attach policies to documents and then encrypt the result for subsequent use by *CDview*. The shell scripts can run on either α-Nexus or Linux.

Each *ConfDocs* document d is represented using XML according to the DocBook standard, and we define $Prin(d)$ to be equal to the SHA1 hash of that XML representation. The representation for a document d includes set $txt(d)$ of text elements, as well as credentials that give a classification ℓ_χ for each text element $\chi \in txt(d)$:

$$Prin(d) \textsf{ says } (\texttt{cls}(\chi) = \ell_\chi)$$

or:

$$CA_T \textsf{ says } (\texttt{cls}(\chi) = \ell_\chi)$$

Here, CA_T is a classification authority; credentials it issues must be accompanied by a suitable restricted delegation

$$Prin(d) \textsf{ says } CA_T \xrightarrow{\mathsf{v_1,v_2}:\ \texttt{cls}(\mathsf{v_1})=\mathsf{v_2}} Prin(d) \tag{3.14}$$

attesting that the publisher of d trusts CA_T to assign classifications to text elements.

The representation of d optionally may include *sanitization credentials*

$$San \textsf{ says } (\texttt{cls}(\texttt{edit}(\chi, s)) = \ell_{\chi,s}) \tag{3.15}$$

that give a classification for the text element produced by executing a built-in edit function to modify χ according to script s. Here, *San* is either $Prin(d)$ or some classification authority CA_T for which restricted delegation (3.14) appears in the representation of d.

Script s comprises standard text editor commands like $\texttt{replace}(\texttt{x}, \texttt{y})$, which replaces all instances of character string x with string y, and so on.

Credentials like (3.15) define a *sanitization policy*. Such a policy characterizes ways to transform a document containing information that readers are not authorized to access into a document those readers are. The hard part is resolving the tension between hiding too much and indirectly leaking classified information. Sanitization of paper documents, for example, often involves replacing fragments of text with whitespace, but a document sanitized in this manner might still leak information to a reader by revealing the length of a replaced name or the existence of an explanatory note.

A user H attempting to view a document d invokes CDview, furnishing a credential signed by some clearance authority CA_U that attests to $\texttt{clr}(H)$:

$$CA_U \textsf{ says } \texttt{clr}(H) = \ell_H$$

Not all clearance authorities are equivalent. The publisher of d controls whether a clearance authority CA_U is trusted to assign clearances and, thus, can participate in determining which users have access to d. Specifically, the publisher includes a credential

$$\mathit{Prin}(d) \textsf{ says } CA_U \xrightarrow{\mathrm{v}_1,\mathrm{v}_2:\ \texttt{clr}(\mathrm{v}_1)=\mathrm{v}_2} \mathit{Prin}(d)$$

in the $\mathit{ConfDocs}$ representation of d for each clearance authority CA_U that is trusted.

α-Nexus Sealed Bundles

To ensure that CDview is the only way to view confidential documents, they are stored and transmitted in encrypted form. α-Nexus, in conjunction with a TPM secure co-processor, implements a storage abstraction that is ideal for this task.[6] An α-Nexus *sealed*

[6] α-Nexus and TPM terminology differ here: *sealing* in α-Nexus is a generalization of the TPM's *binding* interfaces, which enforce confidentiality policies specified in terms of the TPM's platform configuration register values. α-Nexus does not use TPM *sealing* interfaces, which provide for both authentication and confidentiality but only for data consisting of certain symmetric keys generated by that TPM.

bundle b comprises (i) a payload $\mathit{payload}(b)$ stored in encrypted form and (ii) a NAL group $\mathit{Group}(b)$ of constituents authorized to decrypt $\mathit{payload}(b)$.

The α-Nexus kernel implements a system call $\texttt{decrypt}(b, \mathit{pf})$, which a process P can invoke to request that b be decrypted. Just as for other α-Nexus system calls, the pf parameter is either a NAL proof or an index into P's alias table; its purpose will be described shortly. By invoking $\texttt{decrypt}(b, \mathit{pf})$, process P is seen by the α-Nexus kernel to be providing the credential

$$P \textsf{ says } \texttt{decrypt}(b).$$

α-Nexus responds by decrypting and returning $\mathit{payload}(b)$ to P if and only if authorization policy

$$\mathit{Group}(b) \textsf{ says } \texttt{decrypt}(b)$$

can be derived. To allow an access thus requires the kernel to verify a proof of $P \rightarrow \mathit{Group}(b)$, thereby establishing that P is among $\mathit{Group}(b)$ constituents. One way the kernel can discharge this obligation is by checking whether P satisfies a NAL formula $\mathcal{P}_b[\mathsf{v} := P]$, where characteristic predicate $\mathcal{P}_b$ was originally provided for defining $\mathit{Group}(b)$ and saved in the bundle; $P \rightarrow \mathit{Group}(b)$ then follows due to MEMBER (2.13). α-Nexus also allows P to provide a proof of $P \rightarrow A$, where A is some other principal; the kernel would validate that proof and then check that $\mathcal{P}_b[\mathsf{v} := A]$ is satisfied. Process P can provide such a proof by passing it in the pf parameter to the $\texttt{decrypt}(\cdot, \cdot)$ system call. Or P can create an alias table entry for A and pass the index for that entry in the pf parameter instead. Notice, the set of principals satisfying $\mathcal{P}_b$ is not necessarily static if $\mathcal{P}_b$ depends on state, and therefore the $\mathit{Group}(b)$ constituents may be dynamic.

Each $\mathit{ConfDocs}$ document d is stored using an α-Nexus bundle b_d, where $\mathit{Group}(b_d)$ is a set of principals corresponding to valid instances of CDview. A process is considered a valid instance of CDview if and only if the hash of its program manifest equals the hash h_{CDview} of some pre-determined correct program manifest for CDview, that process is running on a valid α-Nexus kernel, and the α-Nexus kernel is itself executing on a trusted processor with associated TPM. We again rely on a platform authority that issues credentials like (3.7) attesting to the public key of each trusted processor's TPM. Thus the group $\mathit{Group}(b_d)$ of principals for each document d is defined in NAL as:

$$\{\!\!\{\mathsf{v} : (\exists \mathsf{k}, \mathsf{p}\colon\ \mathtt{certified\_hardware}(\mathsf{k}) \wedge \\ \mathtt{k.pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \textsf{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{CDview}))\}\!\!\}$$

3.3 *CertiPics*: Axiomatic and Synthetic Bases for Authorization

The integrity of an image is often based on trust in an image provider [90], an axiomatic basis for trust. For example, today consumers trust images found in reputable newspapers, and newspapers in turn trust reputable on-line image repositories. And a court might require photographs used as evidence to come from digital cameras [32] that sign each image they produce. Some academic journals perform automated analysis on images in an attempt to avoid fraud in scientific publishing [51, 155], an example of an analytic basis for trust. Synthetic bases for trust in images are also common, though typically they are enforced only in an ad-hoc manner. For instance, newspapers might only publish images that comply with editorial policies regarding what constitutes an acceptable series of image transformations.

CertiPics formalizes these approaches by allowing a definition of image integrity to be specified as a NAL policy, and enforcing such policies on images. *CertiPics* comprises a set of image manipulation tools that run on α-Nexus, each taking some parameters and one or more images as input and producing a single output image and several NAL credentials. We also implemented an image viewer containing a guard to ensure that displayed images satisfy a user-specified image integrity policy. This involved several domain-specific proof generators for such policies; the proof generators take as input the credentials issued by *CertiPics* image manipulation tools.

Just as the text of a document conveys beliefs, so too does the contents of an image. So, to model an image integrity policy in NAL, we identify each image i with a principal $\mathit{Prin}(i)$, and we write a NAL formula $\mathit{Prin}(i)$ $\textsf{says}$ $\mathcal{P}$ for each belief $\mathcal{P}$ that i conveys. $\mathit{CertiPics}$ enables different principals to define different image integrity policies. We define an image i to have *integrity* according to principal A (named by a private key K_A) if and only if the beliefs i conveys are also conveyed by some designated principal $\mathit{Imgs}(A)$ trusted by A. Thus a guard enforcing A's image integrity policy allows a display request for image i to proceed only if the following holds:

$$\mathit{Prin}(i) \rightarrow \mathit{Imgs}(A). \tag{3.16}$$

3.3.1 Axiomatic Bases for Image Integrity

In the simplest case, A might declare some images to *a priori* have integrity. For each such image i, A issues a credential conveying

$$K_A \ \textsf{says}\ \mathit{Prin}(i) \rightarrow \mathit{Imgs}(A). \tag{3.17}$$

If we take $\mathit{Imgs}(A)$ to be

$$K_A.\texttt{imgs} \tag{3.18}$$

for any principal A, then SUBPRIN (2.10) derives $K_A \rightarrow \mathit{Imgs}(A)$.[7] Together with (3.17), this discharges (3.16) for requests to display images declared by A to *a priori* have integrity. In order that credentials like (3.17) remain small, we define $\mathit{Prin}(i)$ to be equal to $H(i)$, the SHA1 hash of the image data.

Using $\rightarrow$ in (3.16) to encode the notion of image integrity makes it simple to introduce

[7]Since we require A be able to create delegations for $\mathit{Imgs}(A)$, one might be tempted to define $\mathit{Imgs}(A)$ to be K_A. Our approach using a sub-principal $K_A.\texttt{imgs}$ seems more prudent on Least Privilege grounds, because it separates beliefs conveyed by images (hence found in the worldview of $\mathit{Imgs}(A)$) from beliefs held by A proper. An alternative approach would be to use restricted delegation in (3.17), but our approach also allows us to define sub-principals to represent other sets of images. We take advantage of this flexibility in the image integrity policies discussed in the remainder of this section.

a level of indirection so that A need not issue a separate instance of credential (3.17) for each image. Consider a camera C that is equipped with a key K_C and that digitally signs each image i it produces. Because it represents the camera's declaration that i has integrity, the resulting credential conveys the NAL formula

$$K_C \text{ says } \mathit{Prin}(i) \rightarrow \mathit{Imgs}(C). \tag{3.19}$$

Given definition (3.18) of $\mathit{Imgs}(\cdot)$, from an instance of (3.19) we obtain $\mathit{Prin}(i) \rightarrow \mathit{Imgs}(C)$ for each image i that C produces. So A can declare all images produced by camera C to have integrity by issuing a credential:

$$K_A \text{ says } \mathit{Imgs}(C) \rightarrow \mathit{Imgs}(A).$$

This derives $\mathit{Imgs}(C) \rightarrow \mathit{Imgs}(A)$. Together with an instance of (3.19) we obtain $\mathit{Prin}(i) \rightarrow \mathit{Imgs}(A)$ for each image i produced by camera C, which discharges (3.16) for requests to display that image.

CertiPics supports further levels of indirection. For instance, if a manufacturer M (named by key K_M) issues a credential conveying

$$K_M \text{ says } \mathit{Imgs}(C) \rightarrow \mathit{Imgs}(M) \tag{3.20}$$

for each camera C that it manufactures, A can declare all images by all such cameras to have integrity using a single credential that conveys K_A says $\mathit{Imgs}(M) \rightarrow \mathit{Imgs}(A)$.

3.3.2 Synthetic Bases for Image Integrity

As an example of a synthetic basis for authorization in *CertiPics*, consider the image integrity policy in Figure 3.2, which we dub the *NYT* policy.[8] The first rule of the *NYT*

[8]The policy we discuss here was inspired by clarifications made by the New York Times [104] subsequent to the widespread publishing—on many news Web sites, including the New York Times—of a doctored photograph of an Iranian missile test. The New York Times in fact does have a policy (though not machine checkable) on image integrity [102].

(1) Images produced by any camera from manufacturer M (named by key K_M) are considered *validated source images*.

(2) A derived image has *integrity* if it derives from a validated source image by a series of (zero or more) transformations that crop, redact, or add captions.

Figure 3.2: NYT policy—an example image integrity policy.

(1) Images produced by any camera from manufacturer M (named by key K_M) are considered *validated source images*.

(2) A derived image has *integrity* if it derives from some set of validated source images by a series of (zero or more) transformations and the series satisfies the constraints implied by the following rules.

    (a) Cropping is allowed.

    (b) Scaling is allowed, but only once.

    (c) Color balancing is allowed, but only before scaling.

    (d) Tiling (e.g., joining two images side by side) is allowed, but only if a visible border is inserted between the tiled images.

Figure 3.3: JCB policy—an example image integrity policy.

policy defines a set of *validated source images*. The second rule concerns the *chain-of-custody* of an image by defining conditions under which an image derived from validated source images itself has integrity. The NYT policy permits transformations to be applied to a validated source image any number of times (and in any order), and each image that has integrity derives from some single validated source image.[9]

More generally, an image integrity policy might define constraints on the history of transformations that have been performed, and the policy might permit transformations that combine multiple input images to produce an output image. Consider the JCB policy shown in Figure 3.3, inspired by the policies of the Journal of Cell Biology and de-

[9]Note, the NYT policy, as presented in Figure 3.2, is lax about the distinction between a validated source image and a derived image that merely has integrity. In fact, the distinction is immaterial in the NYT policy: if image i derives from image j by some otherwise acceptable transformation, and j has integrity according to NYT—but is not a validated source image—then there must be some validated source image j' from which both i and j are derived by a series of acceptable transformations. Shortly, we discuss a policy in which the distinction is important.

signed to preserve the integrity of scientific data represented in published images. Here, some of the chain-of-custody constraints (i.e., (2a) and (2d) of Figure 3.3) are *local*: they concern only information about a single step in a series of transformations. Other chain-of-custody constraints (i.e., (2b) and (2c) of Figure 3.3) are *global*: they concern the entire history of transformations performed.

In practice, $\mathit{CertiPics}$ represents the NYT and JCB policies as a set of credentials issued by principals NYT and JCB, respectively. Here we use the generic principal A (named by key K_A) as a placeholder. We define an image i to be a *validated source image* according to A if and only if $\mathit{Prin}(i) \rightarrow \mathit{Srcs}(A)$, where $\mathit{Srcs}(A)$ is some designated principal trusted by A. So to define the set of validated source images, A might issue a credential conveying NAL formula

$$K_A \text{ says } \mathit{Imgs}(M) \rightarrow \mathit{Srcs}(A). \tag{3.21}$$

We take $\mathit{Srcs}(A)$ to be $K_A.\texttt{srcs}$ so that (3.21) derives $\mathit{Imgs}(M) \rightarrow \mathit{Srcs}(A)$. Together with suitable instances of credentials (3.19) and (3.20), we obtain that $\mathit{Prin}(i) \rightarrow \mathit{Srcs}(A)$ holds—meaning that i is a validated source image according to A—for each image i produced by a camera C manufactured by M.

Consider some process P that instantiates an image transformation xform and produces an output image i. $\mathit{CertiPics}$ characterizes such a transformation by its input parameters, which comprise input images j_1, …, j_n along with additional (e.g., string or integer) parameters $p_1, \ldots, p_m$. The derivation of i is described by a credential conveying the NAL formula

$$P \text{ says } \mathit{Prin}(i) = \mathit{xform}(\mathit{Prin}(j_1), \ldots, \mathit{Prin}(j_n), p_1, \ldots, p_m). \tag{3.22}$$

In $\mathit{CertiPics}$, one instance of this credential is associated with each derived image i.

Each instance of (3.22) concerns information local to one step in a series of image transformations. So in addition to identifying each image i with a principal $\mathit{Prin}(i)$, $\mathit{CertiPics}$

identifies a pair $\langle i, \hbar \rangle$ with a principal $Prin_{tag}(\langle i, \hbar \rangle)$, where i is a derived image and $\hbar$ is a *tag* that itemizes the relevant history of image i. Depending on the image integrity policy, $\hbar$ could be simply a list of (parameterized) descriptions of each transformation that has been applied, or $\hbar$ might have to be a tree of such descriptions.

We say that $\hbar$ is a *valid tag* for i if and only if

(i) $\hbar$ accurately describes the derivation of i from some set of source images, and

(ii) those source images are validated source images.

CertiPics formalizes this by defining $\hbar$ to be a valid tag for i, according to A, if and only if

$$Prin_{tag}(\langle i, \hbar \rangle) \rightarrow Tagged(A) \tag{3.23}$$

holds, where $Tagged(A)$ is some designated principal trusted by A. We take $Tagged(A)$ to be K_A`.tagged` to allow A to create such delegations (hence, to define which tags are valid for an image). And we define $Prin_{tag}(\langle i, \hbar \rangle)$ to be $H(Prin(i)||\hbar)$, or equivalently, $H(H(i)||\hbar)$, the SHA1 hash of the image (itself represented by a SHA1 hash) concatenated with a canonical encoding the associated tag.[10]

The empty set is, by definition, a valid tag for a validated source image. Thus A issues a credential:

$$K_A \textsf{ says } (\forall \mathsf{i} : (\mathsf{i} \rightarrow Srcs(A)) \Rightarrow (\mathtt{sha1}(\mathsf{i}||\emptyset) \rightarrow Tagged(A))). \tag{3.24}$$

This credential represents the base case of an inductive definition. The inductive step is

[10]We could alternatively define $Prin_{tag}(\langle i, \hbar \rangle)$ to be $H(\langle i, \hbar \rangle)$ to avoid nested applications of SHA1, but that would be slightly less convenient for the formalization we describe below.

represented by a second credential:

$$
\begin{aligned}
K_A \text{ says } (\forall \mathsf{i}, \mathsf{h}, \mathsf{j}_1, \mathsf{h}_1, \ldots, \mathsf{j}_n, \mathsf{h}_n, \mathsf{p}_1, \ldots, \mathsf{p}_m :& \\
&(\mathsf{i} = \mathit{xform}(\mathsf{j}_1, \ldots, \mathsf{j}_n, \mathsf{p}_1, \ldots, \mathsf{p}_m) \wedge \\
&\ \mathsf{h} = \mathit{tag}(\mathit{xformname}, \mathsf{h}_1, \ldots, \mathsf{h}_n, \mathsf{p}_1, \ldots, \mathsf{p}_m)) \Rightarrow \\
&\ (\mathtt{sha1}(\mathsf{i}||\mathsf{h}) \rightarrow \mathtt{sha1}(\mathsf{j}_1||\mathsf{h}_1) \vee \ldots \vee \mathtt{sha1}(\mathsf{i}||\mathsf{h}) \rightarrow \mathtt{sha1}(\mathsf{j}_n||\mathsf{h}_n)))
\end{aligned}
\tag{3.25}
$$

for each transformation relevant to the image integrity policy to be enforced. Here, $tag(\cdot)$ is a policy-specific function that constructs a new tag $\hbar$ given the name $xformname$ of an image transformation, the tag $\hbar_1$, …, $\hbar_n$ for each input image, and the transformation's additional parameters p_1, …, p_m.

Note that we use disjunction, rather than conjunction, in the consequent of the implication in (3.25). This is because instances of (3.25) are ultimately used to derive (3.23) for an image i and its tag $\hbar$. With conjunction, if the antecedent of the implication in (3.25) is satisfied, then from the consequent we would obtain $Prin_{tag}(\langle i, \hbar\rangle) \Rightarrow Prin_{tag}(\langle j_1, \hbar_1\rangle)$, meaning that if $\hbar_1$ is a valid tag for input image j_1, then so too would $\hbar$ be a valid tag for i regardless of whether tags $\hbar_2$, …, $\hbar_n$ are valid for input images $j_2, \ldots j_n$. This would not provide the desired semantics: with conjunction we would be able to derive (3.23) for an image i and its tag $\hbar$ so long as *any* input image had a valid tag. With disjunction in the consequent, (3.25) derives $Prin_{tag}(\langle i, \hbar\rangle) \Rightarrow Tagged(A)$ only if $Prin_{tag}(\langle j_k, \hbar_k\rangle) \rightarrow Tagged(A)$ holds for *every* input image j_k and associated tag $\hbar_k$.

Some trust assumptions are required for credential (3.22), issued by P, to be useful in combination with credential (3.25), issued by A. In particular, part of an image integrity policy is a definition of which processes are trusted to instantiate each image transformation. Thus A might issue a credential conveying NAL formula

$$K_A \text{ says } P \xrightarrow{\mathsf{i,j,x,y}\,:\,\mathsf{i}=\mathtt{scale}(\mathsf{j,x,y})} K_A \tag{3.26}$$

to designate process P as trusted on the $\mathtt{scale}(\cdot,\cdot,\cdot)$ image transformation, which produces an image i given one input image j and two integer parameters x and y.

Given the above definitions, we can now define which derived images have integrity: derived input image i has integrity according to A if and only $\hbar$ is a valid tag for i and $\hbar$ satisfies some predicate $custodyPol(\hbar)$ representing policy-specific chain-of-custody constraints. Thus we have a credential:

$$\begin{aligned} &K_A \text{ says } (\forall \mathsf{i}, \mathsf{h} : \\ &\qquad ((\mathtt{sha1}(\mathsf{i}||\mathsf{h}) \rightarrow \mathit{Tagged}(A)) \wedge \mathit{custodyPol}(\mathsf{h})) \Rightarrow (\mathsf{i} \rightarrow \mathit{Imgs}(A))) \end{aligned} \tag{3.27}$$

Thus a complete image integrity policy in *CertiPics* comprises the following elements.

- Instances of credential (3.21) to define the set of valid source images.
- A definition for the $tag(\cdot)$ function used in instances of (3.25) to construct tags for derived images.
- Instances of credential (3.26) to define which processes are trusted to implement image transforms.
- A definition for the $custodyPol(\cdot)$ predicate used in (3.27) to represent chain-of-custody constraints.

Notice, *CertiPics* relies both on synthetic bases (for chain-of-custody constraints) and axiomatic bases (for validated source images) for authorizing the display of a derived image.

3.3.3 Implementation Details

CertiPics comprises a set of image transformation programs, an image viewer *CPview*, and a set of domain-specific proof generators for image integrity policies.

- Each image transformation program runs on α-Nexus. When executed, the program issues credentials conveying (3.22) to describe the transformation. In our prototype, all such credentials are stored in a *credential database* to facilitate the construction of proofs that involve credentials from multiple images. Also stored in the credential database are credentials issued by α-Nexus about the processes (and kernel and hardware) that execute *CertiPics* image transformation programs.

- *CPview* implements a guard to authorize display requests from images. *CPview* is configured with an image integrity policy, but *CPview* does not create proofs. Instead, *CPview* invokes an external domain-specific proof generator to obtain a proof for each display request, and the guard in *CPview* invokes the NAL guard library to check that the returned proof is correct and discharges (3.16).
- We implemented domain-specific proof generators, including one for a more complete variant of the NYT policy (Figure 3.2) and one for the JCB policy (Figure 3.3). Each proof generator is invoked with the SHA1 hash $H(i)$ of the image to be displayed. The proof generator then accesses the credential database to obtain credentials associated with i (or with images from which i purports to have been derived) and credentials associated with the processes that performed the transformations that derived i. The proof generators need not be trusted, because their output—a proof—is checked by *CPview*. They can be executed wherever and whenever is most convenient—at the time an image is published, just before viewing, or some other time.

CertiPics was implemented in C, requiring 1274 lines of code for image transformation programs, 1887 lines of code for *CPview*, and 1554 lines of code for domain-specific proof generators. The image transformation programs additionally use the libpng [123] image parsing library, and *CPview* uses the NAL guard library.

Figure 3.4 shows a subset of the image transformation programs implemented by *CertiPics*. The credentials issued by processes executing these programs can be inferred from their signatures. For instance, a process P that executes the `tile`(j_1, j_2, b) program

$\texttt{crop}(j, p_1, p_2)$	—Image j cropped to the box given by points p_1 and p_2.
$\texttt{scale}(j, x, y)$	—Image j scaled by a factor of x horizontally and y vertically.
$\texttt{balance}(j, m, r, g, b)$	—Image j with colors balanced according to method m (a string) and integer parameters r, g, and b.
$\texttt{tile}(j_1, j_2, b)$	—Images j_1 and j_2 tiled with a border of b pixels.
$\texttt{overlay}(j_1, j_2, p)$	—Image j_1 overlaid on image j_2 at point p.
$\texttt{redact}(j, p_1, p_2)$	—Image j with the box given by points p_1 and p_2 redacted.
$\texttt{blur}(j, p_1, p_2, b)$	—Image j with the box given by points p_1 and p_2 blurred by b pixels.
$\texttt{caption}(j, t, p)$	—Image j overlaid with caption text t at point p.
$\texttt{arrow}(j, p_1, p_2)$	—Image j overlaid with an arrow from points p_1 to p_2.

Figure 3.4: A subset of *CertiPics* image transformation programs.

to produce image i will issue a credential conveying NAL formula

$$P \textsf{ says } H(i) = \texttt{tile}(H(j_1), H(j_2), b),$$

where $H(\cdot)$ is a SHA1 hash.

The image transformation programs are agnostic about policy. In particular, the credentials they issue do not contain information about global image history, since *CertiPics* encodes history for images in a policy-specific manner. Actually, we implemented most of the transformation programs while working only with the *NYT* policy example, which does not require any history. This gives us a small measure of confidence that *CertiPics* is flexible enough to accommodate other image integrity policies or even other bases for trust, since we then extended our work to accommodate the *JCB* policy example, which does require global history.

In general, constructing proofs is hard. But, because each proof generator in *CertiPics* need only work for a subset of image integrity policies, we can implement a straightforward and efficient proof construction strategy that takes advantage of domain-specific knowledge about the structure of *CertiPics* credentials and proofs for that subset of poli-

(1) Enumerate credentials conveying NAL formulas of the form

$$P \text{ says } H(i) = \mathit{xform}(H(j_1), \ldots, H(j_n), p_1, \ldots, p_m),$$

where $H(i)$, $H(j_1)$, …, $H(j_n)$ are SHA1 hashes of images.

(2) For each such credential, check whether

$$P \xrightarrow{H(i)=\mathit{xform}(H(j_1),\ldots,H(j_n),p_1,\ldots,p_m)} K_A$$

holds, where K_A is either K_{NYT} or K_{JCB} as appropriate. If not, discard the credential.

(3) Starting with the target image as the root, reconstruct image relationships by constructing a tree having images as nodes, and with output images being the parent of the input images from which they were derived, as evidenced by credentials from step (2). The leaves of the resulting tree are taken to be *source images*.

(4) Assign tags to images by traversing the resulting tree in depth-first order: leaf nodes are assigned tag $\emptyset$; tags for parent nodes are constructed from the tags of their children in accordance with the $\mathit{tag}(\cdot)$ function defined by the image integrity policy.

(5) Check that $\mathit{custodyPol}(\hbar)$ holds, where $\hbar$ is the tag for the target image (i.e., at the root of the tree).

(6) Check that $H(j) \rightarrow K_A.\texttt{srcs}$ holds for each source image j, where K_A is either K_{NYT} or K_{JCB} as appropriate.

Figure 3.5: Outline of *CertiPics* proof generator strategy for NYT and JCB when executed for some target image.

cies. The proof generators we implemented for the NYT and JCB policies rely on a graph search to build a tree[11] of image transformations representing the derivation of a given image. The structure of a proof for (3.16) largely mirrors this tree. We implemented a backwards-chaining search strategy to analyze the credential database and generate proofs (if they exist). A sketch of this strategy is shown in Figure 3.5.

The NYT policy (shown in Figure 3.2) is represented in *CertiPics* by a set of digitally signed credentials. The principal NYT (named by key K_{NYT}) takes the place of the prin-

[11] Although is possible to derive a single image using many different sequences of transformations, this is unlikely to occur often in practice. For simplicity of presentation, we assume each derived image has a unique derivation from source images.

cipal A used in Sections 3.3.1 and 3.3.2. NYT issues a credential conveying (3.26) only for the image transformation programs mentioned in the policy, i.e., `crop`$(\cdot,\cdot,\cdot)$, `redact`$(\cdot,\cdot,\cdot)$, and `caption`$(\cdot,\cdot,\cdot)$, as follows.

$$K_{NYT} \textsf{ says } G_{crop} \xrightarrow{\mathsf{i,j,p_1,p_2:\ i{=}crop(j,p_1,p_2)}} K_{NYT}$$
$$K_{NYT} \textsf{ says } G_{redact} \xrightarrow{\mathsf{i,j,p_1,p_2:\ i{=}redact(j,p_1,p_2)}} K_{NYT}$$
$$K_{NYT} \textsf{ says } G_{caption} \xrightarrow{\mathsf{i,j,t,p:\ i{=}caption(j,t,p)}} K_{NYT}$$

Here, G_{xform} is a NAL group of processes trusted by K_{NYT} to execute image transformation program $xform$. We take G_{xform} to be the NAL group

$$\{\!\!\{\mathsf{v} : (\exists \mathsf{k},\mathsf{p}:\ \mathtt{certified\_hardware}(\mathsf{k}) \wedge$$
$$\mathsf{k.pcrs}(\overline{h}).\mathsf{epoch}(\mathsf{p}) \textsf{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{xform}))\}\!\!\},$$

where h_{xform} is the hash of a program manifest for the image transformation program corresponding to $xform$. Thus the NYT policy allows $CertiPics$ transformation programs to execute on any trusted α-Nexus platform.

Two additional elements complete the NYT policy: a definition for the $tag(\cdot)$ function, and a definition of the $custodyPol(\cdot)$ predicate. Since delegations were created only for acceptable image transformations (cropping, redacting, and adding captions) and the NYT policy imposes no additional constraints on image derivations, we simply take $tag(\cdot)$ to be $\emptyset$ and $custodyPol(\cdot)$ to be identically true.

The JCB policy is quite similar to the NYT policy, but using key K_{JCB} and allowing a wider set of image transformations. The JCB policy also uses a more involved $tag(\cdot)$ function and $custodyPol(\cdot)$ predicate. Here, image integrity depends on history, so we take $\hbar$ to be a tree of parameterized descriptions of transformations. In NAL, such a tree can be represented by a nested list $\hbar = [xformname, \hbar_1, \ldots, \hbar_n, p_1, \ldots, p_m]$, where $xformname$ is the name of the transformation (a string), $\hbar_k$ is the tag for the k^{th} input image to that transformation (a list), and p_k is the k^{th} non-image parameter for the transformation (an integer, string, etc.). Thus, for the $\texttt{tile}(\cdot,\cdot,\cdot)$ image transformation, we use an instance of (3.22) that conveys:

$$
\begin{aligned}
K_{JCB}\ \textsf{says}\ (\forall \mathsf{i}, \mathsf{h}, \mathsf{j}_1, \mathsf{j}_2, \mathsf{h}_1, \mathsf{h}_2, \mathsf{b}\ :\\
&(\mathsf{i} = \texttt{tile}(\mathsf{j}_1, \mathsf{j}_2, \mathsf{b}) \wedge\\
&\ \mathsf{h} = [\texttt{"tile"}, \mathsf{h}_1, \mathsf{h}_n, \mathsf{b}])\ \Rightarrow\\
&\ \ (\texttt{sha1}(\mathsf{i}||\mathsf{h}) \rightarrow \texttt{sha1}(\mathsf{j}_1||\mathsf{h}_1)\ \ \vee\ \ \texttt{sha1}(\mathsf{i}||\mathsf{h}) \rightarrow \texttt{sha1}(\mathsf{j}_2||\mathsf{h}_2)))
\end{aligned}
\tag{3.28}
$$

Similar credentials conveying instances of (3.22) are issued by JCB for each other image transformation. Together, these define the $tag(\cdot)$ function for JCB.

The $custodyPol(\hbar)$ predicate for JCB defines whether a series of image transformations satisfies constraints implied by rules (2a)–(2d) of Figure 3.3. The predicate can be defined inductively as follows.

$$
K_{JCB}\ \textsf{says}\ \forall \mathsf{h}, \mathsf{h}_1, \mathsf{h}_2, \mathsf{b}\ :
$$

$$
\texttt{custodyPol}(\mathsf{h}) = \begin{cases}
\textsf{true} & \text{if } \mathsf{h} = \emptyset\\
\texttt{custodyPol}(\mathsf{h}_1) & \text{if } \mathsf{h} = [\texttt{"crop"}, \mathsf{h}_1, \cdot, \cdot]\\
\texttt{custodyPol}(\mathsf{h}_1) \wedge \texttt{unscaled}(\mathsf{h}_1) & \text{if } \mathsf{h} = [\texttt{"scale"}, \mathsf{h}_1, \cdot, \cdot]\\
\texttt{custodyPol}(\mathsf{h}_1) \wedge \texttt{unscaled}(\mathsf{h}_1) & \text{if } \mathsf{h} = [\texttt{"balance"}, \mathsf{h}_1, \cdot, \cdot, \cdot, \cdot]\\
\texttt{custodyPol}(\mathsf{h}_1) \wedge \texttt{custodyPol}(\mathsf{h}_2) & \text{if } \mathsf{h} = [\texttt{"tile"}, \mathsf{h}_1, \mathsf{h}_2, \mathsf{b}] \wedge \mathsf{b} > 0\\
\textsf{false} & \text{otherwise}
\end{cases}
$$

The first case corresponds to an empty sequence of image transformations, which satisfies the JCB policy. The next four cases encode rules (2a)–(2d) of Figure 3.3, respectively. Two of the cases—for scaling and color balancing—refer to $\texttt{unscaled}(\hbar)$, a predicate that holds if and only if no scaling image transformations appear in $\hbar$. It is defined inductively as follows.

K_{JCB} says $\forall \mathsf{h}$:

$$\texttt{unscaled}(\mathsf{h}) = \begin{cases} \text{true} & \text{if } \mathsf{h} = \emptyset \\ \texttt{unscaled}(\mathsf{h}_1) & \text{if } \mathsf{h} = [\texttt{"crop"}, \mathsf{h}_1, \cdot, \cdot] \\ \text{false} & \text{if } \mathsf{h} = [\texttt{"scale"}, \mathsf{h}_1, \cdot, \cdot] \\ \texttt{unscaled}(\mathsf{h}_1) & \text{if } \mathsf{h} = [\texttt{"balance"}, \mathsf{h}_1, \cdot, \cdot, \cdot, \cdot] \\ \texttt{unscaled}(\mathsf{h}_1) \wedge \texttt{unscaled}(\mathsf{h}_2) & \text{if } \mathsf{h} = [\texttt{"tile"}, \mathsf{h}_1, \mathsf{h}_2, \cdot] \\ \text{false} & \text{otherwise} \end{cases}$$

CHAPTER 4
A MUTUAL-SUSPICION FILESYSTEM

We built the MSFS filesystem to gain insight into issues not addressed in designing and implementing $\mathit{TruDocs}$, $\mathit{ConfDocs}$, and $\mathit{CertiPics}$. Is NAL and our underlying credentials-based authorization approach compatible with security principles thought to engender trustworthiness? Also of concern was whether using NAL leads to unacceptable performance compared to more conventional approaches to authorization. We chose to study a filesystem because the structure of conventional filesystems are well known and well understood and because a trustworthy filesystem would be of great benefit to α-Nexus applications.

MSFS instantiates the security principles discussed in Section 1.4—Mutual Suspicion, Least Privilege, Complete Mediation, and Minimization of Trusted Computing Bases (TCBs)—and it enforces the following discretionary access control policy using ACLs (see *access control lists*, in Section 1.2.1).

MSFS DAC Policy:

- Every sequence of bytes stored by MSFS on disk has an *owner*.
- Every sequence of bytes stored by MSFS on disk has an ACL, specified by the owner or by some principal acting on the owner's behalf, where the ACL names principals and gives either `read` or `read/write` privileges.
- A request to read or write disk data—whether on disk or in memory—made on behalf of some principal is allowed to proceed only if the principal and corresponding privilege appear on the ACL for that data.

MSFS is designed to enforce this policy while defending against an adversary that launches software-based attacks on the filesystem. Our threat model includes users who may or may not have legitimate access to the machine as well as processes that attempt unauthorized access to filesystem data. But our threat model excludes hardware-based attacks; MSFS does not defend against adversaries with physical access to disks, memory, processors, or other hardware.[1]

MSFS runs on α-Nexus, and MSFS relies on services provided by the α-Nexus kernel. We assume that the kernel does not attempt unauthorized access to filesystem data and is trustworthy in other respects. In particular, we assume:

(i) The currently-executing kernel code and configuration is trustworthy.

(ii) The adversary cannot modify the kernel's code or configuration to cause it to become untrustworthy.

(iii) The adversary cannot reboot the hardware with modified kernel code or configuration that is untrustworthy.

These assumptions ensure that MSFS DAC Policy is not violated when α-Nexus is composed with MSFS.

The design goals for α-Nexus were chosen specifically[2] to discharge assumptions (i)–(iii). Assumption (i) implies that there are no bugs in the kernel. This is a difficult standard for any operating system to meet; achieving it is outside the scope of this dissertation. To address assumption (ii), α-Nexus prohibits dynamic extensions or device driver code from running within the kernel, and it prohibits changes to the kernel's configuration unless the resulting configuration is known to be safe. For an adversary without physical access to violate assumption (iii) would require the adversary to modify kernel code or configuration data stored within the MSFS filesystem.[3] So assumption (iii) is reasonable if assumptions (i) and (ii) hold and if the filesystem enforces a suitable authorization policy for requests to modify such data. A straightforward approach, and one we adopt in MSFS,

[1]Many prior filesystems (e.g., [23,61,65,93]) do focus on defending against physical attacks, particularly off-line attacks against stolen disks. Most such filesystems encrypt data stored on disks or make use of cryptographic hashes to monitor data integrity; MSFS does not, but could.

[2]α-Nexus is written in C and was not intended to be a production system. Thus it is unlikely that α-Nexus discharges assumptions (i)–(iii) with any significant degree of assurance.

[3]Another way an adversary without physical access could violate assumption (iii) is to modify the machine's firmware. α-Nexus prohibits firmware modifications unless the modifications are first verified to be safe.

is to designate the α-Nexus kernel as the owner of this data and specify an ACL that prohibits all write accesses.

A *filesystem administrator* principal is responsible for configuring an MSFS instance. In conventional filesystems, the filesystem administrator owns the meta-data that specifies the owner of every file and directory. So in conventional filesystems, the filesystem administrator can change the owner and, by implication, can modify ACLs to gain access to files and directories. Some conventional filesystem implementations even grant filesystem administrators unrestricted access to all data stored on disks regardless of DAC policies.

In MSFS, filesystem administrators are governed by DAC just like ordinary users. Filesystem administrators are allowed to configure the filesystem, but they do not own user files or directories, nor do they own most of the filesystem meta-data stored on disks. So they are not able to read or write user files or meta-data, nor are they able to change owners or ACLs for user files.

Filesystem administrators in MSFS can reformat disks, so a filesystem administrator can compromise the availability of MSFS user data. Such power seems necessary for a filesystem administrator to manage the system.

To build MSFS, we might have just modified a conventional filesystem or made use of a conventional operating system. That approach faced two challenges.

- In conventional filesystem implementations, the bulk of filesystem code executes in supervisor mode within the operating system kernel. The behavior of such code is largely unconstrained, so it would be difficult to enforce DAC on that code.
- Conventional operating systems grant system administrators, and all processes that execute on their behalf, the power to circumvent DAC policies. We deemed that unwisely liberal.

So we designed MSFS to execute outside of the operating system kernel. And because

MSFS runs on α-Nexus, system administrators are prevented from modifying MSFS code or circumventing MSFS DAC enforcement.

The rest of this chapter is organized as follows. In Section 4.1, we describe features of α-Nexus on which MSFS depends. We provide a brief overview of heuristics for decomposing the filesystem into components in Section 4.2. Section 4.3 details the design and implementation of MSFS, with specific attention to how the security principles of Section 1.4 are instantiated, including trade-offs that entailed. Section 4.5 evaluates the performance impact of various security principles we instantiated in the design of MSFS. We close with a discussion of related work on secure filesystems and security principles for building trustworthy systems.

4.1 Use of the α-Nexus Operating System

Several key features of α-Nexus were particularly useful for MSFS.

- α-Nexus processes are isolated from each other by default, making Mutual Suspicion the norm rather than the exception.
- Processes in α-Nexus interact with each other and with the kernel over channels that have simple and straightforward semantics, chosen to support Complete Mediation. For example, it is trivial in α-Nexus to implement a guard that mediates all incoming inter-process communication (IPC) messages to a process (see Sections 2.4.4 and 4.1.2 for details on α-Nexus IPC).
- NAL and credentials-based authorization supported by α-Nexus are useful for implementing guards to instantiate Least Privilege.
- α-Nexus executes device drivers and various system services outside of the operating system kernel, enabling smaller TCBs.

In this section, we briefly describe these features of α-Nexus and how they are used in MSFS.

A notable feature of α-Nexus, discussed previously in Section 2.3.3, is the use of a Trusted Platform Module (TPM) [137] secure co-processor as a hardware-protected root of trust. α-Nexus relies on the attestation and secure storage facilities of the TPM to protect the confidentiality and integrity of certain data. However, our MSFS prototype does not depend on the presence of a TPM.[4]

We restrict our discussion in this chapter to a single installation of α-Nexus with a kernel denoted by NAL principal name $K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p)$, where K_{CPU} is the public key for the hardware TPM, $\overline{h}$ is replaced by the appropriate platform configuration register values for the α-Nexus kernel code and configuration, and p varies across reboots. For simplicity of presentation, in this chapter we use Kernel as an abbreviation for $K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(p)$. We also assume a designated user $\mathit{FSAdmin}$ for the filesystem administrator, along with an ordinary user Alice. As described in Section 2.3.3, these are sub-principals of the α-Nexus Login service, and the following hold:

$$\begin{aligned} \mathit{Kernel} &\rightarrow \mathit{FSAdmin} \\ \mathit{Kernel} &\rightarrow \mathit{Alice} \end{aligned}$$

4.1.1 Credentials-Based Authorization for MSFS

MSFS DAC is a generalization of traditional DAC implementations. Traditional filesystems typically require a file's owner to be a user and limit ACLs to enumerated sets of users. By contrast, owners in MSFS need not be users—a set of processes can be an owner, for example, where the processes are characterized by specifying public keys, hashes of program manifests, or any other property that can be validated by mechanical analysis.

[4]If the threat model had included certain physical attacks and the filesystem were to encrypt data or monitor data integrity using cryptographic hashes, as suggested previously, then these α-Nexus and TPM facilities would provide a means for storing the necessary keys and hashes.

1.	Kernel says $\texttt{pgm\_hash}(P, h_{pgm})$	Credential (4.2).
2.	$\mathit{Kernel} \rightarrow \mathit{FSAdmin}$	By definition of $\mathit{FSAdmin}$ (see Section 2.3.3).
3.	$(\forall \mathsf{x}:$ $(\mathit{Kernel}$ says $\mathsf{x}) \Rightarrow (\mathit{FSAdmin}$ says $\mathsf{x}))$	From step 2, using definition (2.6) of $\rightarrow$.
4.	$\mathit{FSAdmin}$ says $\texttt{pgm\_hash}(P, h_{pgm})$	Eliminating the implication with steps 1 and 3, using IMP-E (2.4) and substituting for x.
5.	$\mathit{FSAdmin}$ says $(\forall \mathsf{v}:$ $\texttt{pgm\_hash}(\mathsf{v}, h_{pgm}) \Rightarrow (\mathsf{v} \rightarrow \mathit{FSAdmin}))$	Credential (4.3).
6.	$\mathit{FSAdmin}$ says $(P \rightarrow \mathit{FSAdmin})$	By deduction on the beliefs of $\mathit{FSAdmin}$ with steps 4 and 5, using DEDUCE (2.5) and IMP-E (2.4) and substituting for v .
7.	$P \rightarrow \mathit{FSAdmin}$	From step 6, since principals are trusted on their own delegations, using HAND-OFF (2.8).
8.	$(\forall \mathsf{x} : (P$ says $\mathsf{x}) \Rightarrow (\mathit{FSAdmin}$ says $\mathsf{x}))$	From step 7, using definition (2.6) of $\rightarrow$.
9.	P says $\texttt{format}(\mathit{disk}, \ldots)$	Credential (4.4).
10.	$\mathit{FSAdmin}$ says $\texttt{format}(\mathit{disk}, \ldots)$	Eliminating the implication with steps 8 and 9, using inference rule IMP-E (2.4) and substituting for x.

Figure 4.1: A proof of authorization policy (4.1) for MSFS's `format` method with three credentials in evidence and justification provided for each step.

Moreover, MSFS enables any principal to define such groups, in contrast to traditional DAC implementations that permit only system administrators to define groups. And because MSFS employs credentials-based authorization using NAL and NAL groups are specified intensionally, the set of constituents of a group in MSFS can change over time or in response to changes in system state. A NAL group appearing on an ACL, for example, may use a characteristic predicate that refers to the system clock.

As an example of how NAL is used to express an authorization policy for MSFS, consider the guard in MSFS's API to format a disk. Only the filesystem administrator for MSFS is allowed to format a disk. In NAL, this policy is encapsulated by the formula

$$\mathit{FSAdmin} \text{ says } \texttt{format}(\mathit{disk}, \ldots), \tag{4.1}$$

which must be discharged before a request to format a disk is allowed to proceed. Here, $disk$ is a NAL encoding of the MSFS name for the disk to be formatted. When a process makes a request to invoke the filesystem's `format` method, a guard at the filesystem authorizes the request only if NAL formula (4.1) can be proved from accompanying credentials. Suppose that process P is making a request to format the disk, and P has a credential from the α-Nexus kernel attesting to the hash h_{pgm} of P's program manifest:

$$Kernel \textsf{ says } \texttt{pgm\_hash}(P, h_{pgm}). \tag{4.2}$$

And suppose further that $FSAdmin$ issues a credential conveying the NAL formula

$$FSAdmin \textsf{ says } (\forall \mathsf{v} : \texttt{pgm\_hash}(\mathsf{v}, h_{pgm}) \Rightarrow (\mathsf{v} \rightarrow FSAdmin)). \tag{4.3}$$

This credential attributes to the filesystem administrator the assertion that any principal v—likely a process running on α-Nexus—whose manifest has hash h_{pgm} speaks for the filesystem administrator. Then, for the request from process P, three pieces of evidence might be used by the guard:

- credential (4.2) from α-Nexus attesting to the hash of P's program manifest;
- credential (4.3) from the filesystem administrator;
- and P's request itself, which is represented by NAL formula

$$P \textsf{ says } \texttt{format}(disk, \ldots). \tag{4.4}$$

Figure 4.1 shows a complete proof of authorization policy (4.1) given the above evidence. The filesystem guard would therefore authorize P's request to invoke `format` for the specified disk.

Use of α-Nexus Alias Tables for MSFS

Many authorization policies in MSFS share a similar structure: the requesting principal must prove that it speaks for one of a set of prespecified principals, such as the filesystem administrator, the owner of some object, or a principal found on an ACL. Guards in MSFS are specialized for these cases, and MSFS leverages this specialization to amortize the cost incurred by guards for checking NAL proofs. α-Nexus alias tables, described in Section 2.4.4, are used for this purpose. Here, we provide details on how MSFS uses alias tables.

Consider process P from the example above, which is allowed to invoke `format` if $P \to \mathit{FSAdmin}$ holds. The MSFS guard enforcing this policy simply checks if the IPC message conveying the `format` request was augmented by the kernel with the NAL principal $\mathit{FSAdmin}$. Thus, a process that invokes `format` must add $\mathit{FSAdmin}$ as an entry in its alias table, and P must subsequently specify the index of that alias table entry when sending IPC messages to the filesystem.

MSFS makes similar use of alias tables when checking ACLs. For example, MSFS authorizes a request to read a file f only if the following NAL formula holds:

$$\mathit{owner}(f) \text{ says } \texttt{read}(f), \tag{4.5}$$

where $\mathit{owner}(f)$ denotes the principal that owns f. Suppose Alice configures an ACL to grant user U read-only access to a file f that Alice owns. In NAL, that ACL entry represents a restricted delegation:

$$U \xrightarrow{\texttt{read}(f)} \mathit{Alice}. \tag{4.6}$$

If a process P executes on behalf of U, then $P \to U$ holds. Together with (4.6), we derive $P \xrightarrow{\texttt{read}(f)} \mathit{Alice}$. And from P's request to read f, represented by NAL formula P says $\texttt{read}(f)$, we can then obtain (4.5), substituting Alice for $\mathit{owner}(f)$. So MSFS should authorize P's request to read *Alice*'s file f. As with the `format` method, the MSFS guard for the `read` operation simply checks if the IPC message conveying the `read` request was augmented by the kernel with the NAL principal U. Process P must add U as an entry in its alias table and then specify the index of that alias table entry when sending IPC messages to the filesystem.

NAL Groups for MSFS

In the previous example, *FSAdmin* places full trust in any process satisfying the criteria encoded in (4.3), a credential issued by *FSAdmin*. Similarly, $P \rightarrow U$ represents full trust between user U and a process P executing on behalf of U. With NAL, there are several possible approaches if full trust is inappropriate. For example, consider $P \xrightarrow{\texttt{read}(f)} \mathit{Alice}$, where P is any process whose program manifest has hash h_{pgm}.

- *Alice* could place the name of each such process P on the ACL for f with read-only privileges. But this is cumbersome: P's name includes a process ID that is not predictable before P executes, and the name changes at reboot.
- *Alice* could publish a credential similar to (4.3) but using NAL's restricted speaks-for operator:

  $$\mathit{Alice}\ \textsf{says}\ (\forall \mathsf{v} : \texttt{pgm\_hash}(\mathsf{v}, h_{pgm}) \Rightarrow (\mathsf{v} \xrightarrow{\texttt{read}(f)} \mathit{Alice})).$$

  But α-Nexus alias tables are no longer useful, because alias tables do not directly support a restricted delegation like $P \xrightarrow{\texttt{read}(f)} \mathit{Alice}$: P's alias table lists only principals A for which $P \rightarrow A$.[5]

[5]Two considerations informed this aspect of the design of α-Nexus alias tables. First, a restricted delegation can be quite narrow in scope: presumably, P would use a restricted delegation in which the term $\texttt{read}(f)$ appears only when sending to the filesystem an IPC message that encodes a read request for file f. The alias table was intended to instead cache only widely useful sub-proofs, such as those that might be used by P in requests to many different services. And second, the kernel does not interpret the contents of IPC messages. Indeed, encoding rules for IPC vary by application, and only the sender and recipient need to know how messages are encoded. Thus the kernel is not in a position to determine if a particular restricted delegation is suitable for use with a given IPC message. These considerations were revisited during the design of subsequent versions of the operating system, resulting in the previously mentioned—see footnote 19 of Chapter 2—Nexus decision cache as a replacement for α-Nexus alias tables.

We therefore rejected these approaches. Instead, MSFS achieves the desired effect by using NAL groups.

Consider the NAL group

$$\{\!\!\{\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \textsf{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{pgm}))\}\!\!\},$$

the constituents of which are processes whose program manifests, according to some instance of the kernel, have hash h_{pgm}. Recall, this principal name is stable across reboots because it is independent of the process IDs chosen by the kernel at run-time. So this group is suitable as a principal for use on an ACL. For instance, if $Alice$ places this group on the ACL for file f with read-only privileges, then $Alice$ has granted to the group's constituents privileges to read f. The filesystem guard enforcing that ACL would allow P's request to read f to proceed if it can be proved that P is a constituent of that group:

$$P \rightarrow \{\!\!\{\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \textsf{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{pgm}))\}\!\!\}. \tag{4.7}$$

Moreover, the cost of checking a proof of (4.7) can be amortized over multiple accesses, because $\{\!\!\{\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p}) \textsf{ says } \mathtt{pgm\_hash}(\mathsf{v}, h_{pgm}))\}\!\!\}$ can now be added to P's alias table.

4.1.2 α-Nexus Shared Memory and IPC Channels

MSFS relies on two kinds of α-Nexus channels for processes to communicate and synchronize with each other: shared memory regions and IPC channels. These are named using opaque numeric identifiers, chosen at the time the region or channel is created. α-Nexus provides mechanisms to restrict which processes can read and write shared memory regions and which processes can send and receive messages over IPC channels.

α-Nexus implements DAC for shared memory regions. Each shared memory region is owned by the process that created it, and that process controls read and write access to the region by specifying an ACL that contains NAL principal names for `read` or `read/write` privileges. A process invokes the `shm_read` and `shm_write` system calls to read or write data in a shared memory region. Also, a process can invoke the `shm_map` system call to create virtual memory mappings for the underlying memory—thereafter, the process can access that data directly, without kernel intervention. The `shm_read`, `shm_write`, and `shm_map` system calls take a parameter i, specifying an index into the requester's alias table. And before performing any action in response to any of these system calls, a kernel guard performs a lookup for entry i in the requester's alias table and, if the lookup succeeds, checks if the resulting alias and access modes appear on the appropriate ACL. The `shm_change_acl` system call allows the owner to modify the ACL for a shared memory region. If an owner requests changes to an ACL, then the kernel examines existing virtual memory mappings for all processes and deletes those found to be inconsistent with the new ACL. The kernel's shared memory guard is kept deliberately simple, because it is in the TCB for all security goals.

IPC channels in α-Nexus have a single recipient and many senders. A process that creates an IPC channel is the owner of that channel. Two types of messages are supported: IPC requests and IPC responses. The owner is the only process allowed to receive IPC requests, which it does by invoking the `ipc_recv` system call, or to send IPC responses, which it does by invoking the `ipc_reply` system call. The kernel implements a guard that allows the channel owner, for each IPC request from some process, to send at most one IPC response. Any process is allowed to invoke the `ipc_send` system call to send an IPC request over a channel and await the corresponding IPC response.[6]

Two processes participate in every IPC request-response exchange. Complete Mediation for IPC requests and IPC responses is achieved by implementing guards within

[6]In addition to the synchronous request-response semantics described here, IPC channels in α-Nexus also support an asynchronous semantics. Here, the sender of an IPC request is not blocked awaiting an IPC response, nor is the channel owner provided the means to send an IPC response. MSFS uses asynchronous semantics whenever possible, because the implementation exhibits better performance.

these processes. A channel owner runs a guard to authorize IPC requests arriving over the channel; other processes run guards to authorize IPC responses. To implement these guards, processes need a reliable way to attribute IPC requests and IPC responses to the principals that sent them. As described in Section 2.4.4, IPC channels in α-Nexus are authenticated channels: a process P that invokes `ipc_send` or `ipc_reply` specifies an index i into its alias table, and the kernel augments the message with the specified alias A_i after checking that the alias table contains a correct proof of $P \to A_i$. In MSFS, the recipient's guard simply examines the principal name accompanying each IPC message; it is the sender's responsibility to ensure the alias that was selected satisfies the recipient's guard.

4.1.3 α-Nexus Device I/O Privileges

Device drivers in α-Nexus run as processes above the kernel. Each physical I/O device is associated with a device driver process. The process and device interact using a set of (unique) I/O addresses associated with the device; device drivers are not allowed to access other I/O addresses. Device drivers request I/O to instigate direct memory access (DMA) transfers between system memory and devices, but they are not allowed to request I/O that causes DMA transfers to memory outside of the device driver process's virtual memory or to memory that is otherwise unsuitable for DMA transfers.[7]

Neither the kernel nor other processes place full trust in device driver processes. The kernel implements a guard, called the *device driver reference monitor* (DDRM) [146], that tracks relevant system state (and history) in order to distinguish between safe and unsafe I/O operations. Whether a particular I/O operation is permitted by the guard depends on system state including, for example, current DMA-compatible memory allocations, the history of I/O operations previously requested, and the identity and state of the device to which the I/O is addressed.

[7]Memory pages used in DMA transfers must be properly aligned, have virtual memory paging disabled (i.e., "pinning" the pages in memory), and be physically contiguous with low physical addresses.

Complete Mediation requires that the DDRM be invoked for every I/O request. To implement this, a device driver process makes system calls to request that the kernel execute I/O instructions on the process's behalf; α-Nexus disables native hardware I/O instructions for all processes. The α-Nexus kernel invokes the DDRM before performing any action in response to a device driver process's I/O-related system calls. In order that the DDRM can accurately check whether a requested I/O operation is safe, the kernel also notifies the DDRM of relevant changes to system state, such as when DMA-compatible memory is allocated or deallocated.

The DDRM is an example of Mutual Suspicion using an *external guard*—a guard that is implemented not by the intended recipient of requests (i.e., the hardware device) nor by the channel that conveys requests (i.e., the device I/O mechanism), but by some other principal. Using an external guard increases the costs associated with enforcing a policy. In this case, I/O operations incur extra latency by involving the kernel instead of being executed directly by the device driver process. Using an external guard was necessary here because the recipient, a physical device, does not implement a guard and can't be easily modified to do so.[8]

4.2 System Decomposition

How one instantiates Mutual Suspicion, Least Privilege, and Minimization of TCBs depends on how the system is decomposed into components, because this defines the unit of compromise and the granularity of privileges. There is often considerable flexibility about how to decompose a system into components. We should prefer small, fine-grained components since they offer more opportunities to instantiate Mutual Suspicion, Least Privilege and Minimization of Trusted Computing Bases. If the filesystem were implemented as a single large component, for example, then the entire filesystem would be in the TCB

[8]Some hardware platforms include support for enforcing policies in hardware similar to those enforced by the DDRM. These platforms partially obviate the need for the DDRM.

for all security goals. If, instead, the filesystem were decomposed into small components, each with only a small amount of state and code, then each component could conceivably be granted only a relatively small set of privileges. Some components might even be excluded from some of the TCBs.

Granularity of a decomposition affects performance, however. A run-time cost is associated with supporting isolation for a component, so with many finer-grained components the total costs for implementing isolation will be larger. System performance can also suffer due to overhead for supporting interaction between components. This overhead includes the cost of provisioning channels, the extra cost to communicate over such channels as compared to using shared memory, and the costs incurred by mechanisms for Complete Mediation and Mutual Suspicion concerning messages sent over channels. All things equal, a filesystem decomposed into fine-grained components ought to exhibit worse performance than one decomposed into coarse-grained components.

MSFS was decomposed into fine-grained components according to the following three heuristics.

- *Domain decomposition* [53]: Define a component for each cohesive subset of the system state, and include in the component all code necessary for managing and manipulating that state. In a filesystem, for example, a file descriptor table containing information about open file handles might be assigned to one component, while a partition table describing the disk layout is assigned to another component. If security goals concern only a portion of the system state (e.g., file descriptors or partition tables), then this decomposition helps minimize TCB size by isolating sensitive data and its associated code from unrelated parts of the system. However, a single task that spans much of the system state now involves interactions between many components.
- *Functional decomposition* [53]: Define a distinct component for each separate task. Such a decomposition often results in components whose boundaries coincide with

units of code. As with domain decomposition, functional decomposition helps minimize TCB size by isolating critical functionality from unrelated parts of the system. However, functional decomposition can require replicating or sharing data, if tasks being assigned to different components use the same data.

- *Privilege separation* [76, 114]: Decompose code into components according to privileges, placing code requiring similar privileges in the same component and placing code requiring different privileges in different components. This approach should result in designs that allow many opportunities to instantiate Least Privilege. But the approach presumes that privileges are defined before the system is decomposed.

These three heuristics can conflict. As an example, consider a data structure implementing a list of pending disk transfers. This data structure would likely be manipulated both by transfer-scheduling code and by disk driver code. Domain decomposition would suggest placing both the transfer-scheduling code and disk driver code in a single component, because they share state. But privilege separation would suggest decomposing this code into two separate components, one for the scheduler and another for the disk driver, because only the disk driver code requires privileges to perform disk I/O. In the end, decomposition of a system into components requires taste and experience to understand how best to resolve such conflicts.

4.3 Structure and Organization of MSFS

The basic design and some of the code for MSFS derive from the Linux filesystem [88]. MSFS is organized into three main layers, augmented by several modules for concerns that cut across multiple layers. The layers and major modules are shown in Figure 4.2.

- A bottom layer comprises *disk drivers*. These manage DMA transfers between system memory and disks. The disk drivers are based on the Linux libATA [58] driver library.

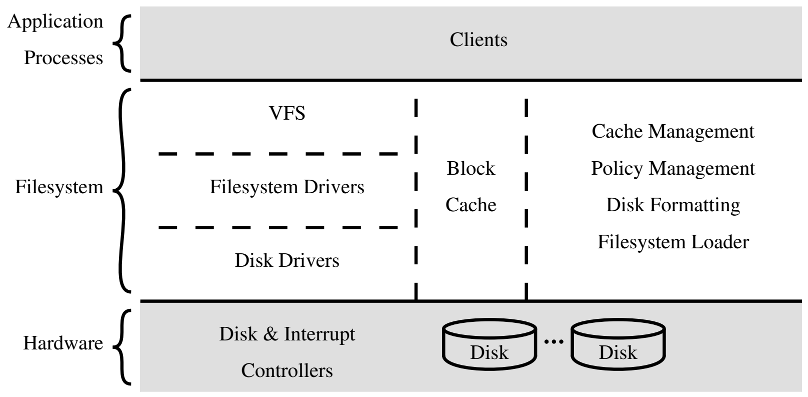


Figure 4.2: Structure of the MSFS filesystem.

- A middle layer comprises *filesystem drivers*, each implementing a standard file and directory tree view of the data stored in one partition of the disk. We implemented one filesystem driver for use in MSFS; it is an adaptation of an open source FAT32 filesystem driver [116].[9]
- A top layer creates a virtual filesystem (VFS). VFS presents a single file and directory namespace to clients, and it hides some details of filesystem drivers. The VFS interface for clients includes a *streams-oriented* interface, in which a client invokes `read` and `write` methods to access file contents, and an *mmap-oriented* interface, in which file contents are mapped into a client's virtual address space for subsequent direct client access.

MSFS runs on standard Intel x86-compatible hardware with dual Serial ATA (SATA) disk buses, each capable of controlling between one and four physical disks.

MSFS data is stored on disks in 4096-byte blocks; this size coincides with the memory page size on our hardware. The k^{th} block on the j^{th} disk of disk bus b is uniquely identified by the tuple $\langle b, j, k\rangle$, called a *block ID*. Block 0 of each disk stores a *partition table* for that disk. Each entry in the partition table defines a partition, specifying a *type code* and a range of block numbers. The type code identifies a filesystem driver that manages the

[9]We chose the FAT32 format for ease of implementation. Other filesystem formats, such as EXT3 or NTFS, would also be straightforward to deploy by adapting the available open source drivers.

corresponding range of blocks (our prototype only supports one type code, FAT32, because it only implements that one filesystem driver). An MSFS disk-formatting module writes block 0 once, during system installation, and the MSFS filesystem-loader module reads block 0 during each reboot.

The owner and ACL for each block are stored on disk and, therefore, persist across reboots. MSFS does not impose constraints on what principal names are used to specify owners or ACL entries. Typical principals include users and NAL groups whose constituents are processes or users. This authorization information is first written during initial system installation, and MSFS flushes changes to disk whenever a block's ACL is modified or the block's owner changes.

4.3.1 File Access Requests

Before describing MSFS design details, it is instructive to consider the high-level steps for a client to access a file. Consider a client that invokes VFS to read an open file. A file read request from the client specifies a *file handle* and a count of bytes to read.

1. Upon receiving the request, VFS looks up the specified file handle in a per-client `file_descriptor_table` to obtain: that client's current offset for reading and writing; and a unique ID for the file. The unique ID is a pair, comprising a reference to the underlying filesystem driver for that file and an ID chosen by that filesystem driver. VFS calculates the range of bytes requested, using the file offset and the specified count, then forwards a read request, with the calculated range of bytes and the file's unique ID, to the filesystem driver for the file.

2. A filesystem driver, upon receiving such a request, translates the file ID and byte range into a set of block IDs. This computation can require access to meta-data on the disk, or the meta-data might already be cached in memory. Once the block IDs are computed, the filesystem driver checks if the requested blocks are cached. If not,

the filesystem driver sends the block IDs in a data transfer request to the appropriate disk driver.

3. A disk driver translates each such request into a series of I/O requests. A hardware interrupt on completion of the transfer initiates a completion action at the disk driver. The disk driver then notifies the filesystem driver where the data can be found in the cache.
4. The filesystem driver sends a response to the VFS layer indicating where the data can be found in the cache.
5. The VFS layer updates the current file offset in the client's `file_descriptor_table` then passes the data from the cache to the waiting client.

Clients communicate with VFS over an α-Nexus IPC channel, and VFS in turn uses IPC to communicate with filesystem drivers. Filesystem drivers communicate with disk drivers through a shared `io_request_queue` data structure that contains data transfer requests to be serviced (asynchronously) by the disk driver. Filesystem drivers insert new requests into this data structure, and they poll to check whether previously inserted requests have completed. Requests are not always performed in first in, first out order because disk drivers re-order requests, combine overlapping requests, or aggregate requests for nearby blocks.

4.3.2 Caching

To mask the high latency and low throughput of physical disks, MSFS caches disk blocks in memory. An MSFS cache management module tracks the status of cached blocks, and it monitors global memory usage and block access patterns. A cache replacement strategy could be implemented by each filesystem driver, since filesystem drivers are well suited to making predictions about future accesses to blocks. Or the cache replacement strategy could be implemented by the cache management module, which has more comprehensive information about current and past cache usage across all filesystems. Our prototype implements most of the cache replacement strategy in the cache management module, but filesystem drivers provide hints about predicted future accesses.

MSFS avoids the expense of copying blocks when requests or responses move between layers of the filesystem. Thus, messages within and between layers refer to the unique copy of data stored in the block cache; all parts of the filesystem share access to the block cache. Disk hardware also shares access to cached blocks, performing DMA transfers directly between the block cache and a disk.

4.4 MSFS Implementation: Components and Privileges

Mutual Suspicion is supported if MSFS components are isolated from each other. Additionally, because the α-Nexus kernel is in the TCB for all security goals—it has access to all system state, for example—Minimization of TCBs dictates that MSFS code not be located in the kernel. So most MSFS components execute as separate processes above the kernel. In some cases, we judged the higher performance costs associated with supporting process isolation to outweigh its contribution to trustworthiness. One such case is discussed in Section 4.4.8. There, more than one MSFS component is executed in a single process, even though this means sacrificing some of the trustworthiness benefits. We examine the performance impact of that design decision in Section 4.5.3. In another case, discussed in Section 4.4.4, a small amount of MSFS code is situated within the kernel at the cost of enlarging the TCB for all security goals. We did not evaluate the performance impact of that design decision.

Certain MSFS components must be in the TCB for MSFS DAC Policy enforcement because they are responsible for protecting the integrity of the enforcement mechanisms for this policy. Consistent with Minimization of TCBs, we endeavored to reduce the number

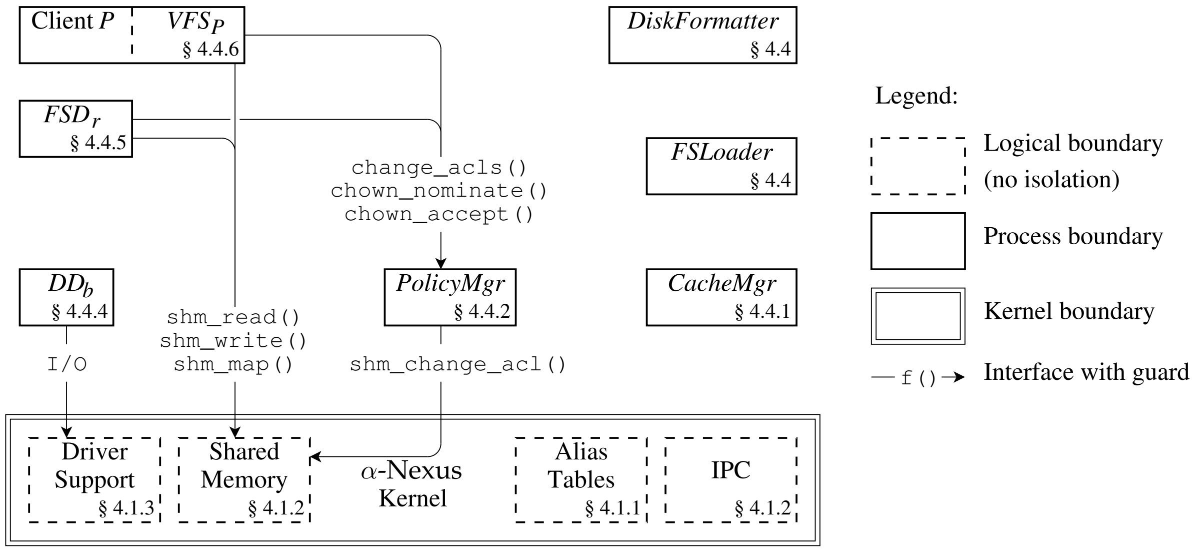


Figure 4.3: Final implementation of the MSFS filesystem, showing major components and some of the interfaces between components. Numbers refer to the sections of this chapter in which each component or interface is discussed.

and size of such components. Execution of all other MSFS components is subject to the enforcement mechanism. In particular, those components are prohibited from accessing blocks (cached in memory or stored on disk) that contain the contents of user files, since such blocks are owned by users and do not include MSFS components on their ACLs. For blocks that store other information, Least Privilege dictates that an MSFS component have access to the data only if its task requires such access. The owner of such a block is the MSFS component that creates and manages the block's data, and an MSFS component is included on the ACL for such a block only if that component needs to access that data. In fact, MSFS components rarely share access to block data, so most ACLs for these blocks contain only a single entry, which grants the owner of the block full access.

Several modules in MSFS are implemented as independent components executing as α-Nexus processes—an example of functional decomposition. These include: *FSLoader*, to execute the filesystem-loader module; *DiskFormatter*, to execute the disk-formatting module; *CacheMgr*, to execute the block cache management module; and *PolicyMgr*, to execute code for managing block owners and ACLs. Below, we describe some of these components in more detail, justify how the remainder of MSFS was partitioned into components, and describe the privileges each MSFS component holds. Figure 4.3 provides a guide for the discussion by illustrating the final structure we arrived at for the MSFS implementation, including the major components of MSFS and some of the interfaces between those components.

4.4.1 Cache Management Component

$CacheMgr$ has overall responsibility for managing the block cache. For each page of memory allocated for the block cache, $CacheMgr$ tracks: a block ID, a reference count, usage statistics, and a *page status flag*. The page status flag can be either `empty`, for a page that is allocated for a particular block but not yet filled with data, or `filled`, for a page that has been filled with data for the appropriate block from disk. In the later case, $CacheMgr$ also implements a *block status flag*, which can be either `dirty`, for a page containing cached block data that has been modified in memory and not yet flushed to disk, or `clean`, otherwise.

Block and page status flags are used to ensure that clients and MSFS components do not access pages in the block cache before those pages are filled with appropriate data. The flags are also used to ensure that changes made to cached blocks are flushed to disk before block cache memory pages are deallocated.

MSFS components and clients refer to cached blocks using *block references*. Block references are passed as parameters when invoking the $CacheMgr$ API, which includes interfaces for incrementing and decrementing reference counts, reading or updating usage statistics, and reading block and page status flags. Processes invoke these interfaces over an IPC channel. But because efficient access to cached blocks is so critical to system performance, MSFS stores the contents of cached blocks in α-Nexus shared memory regions. A single ACL controls access to all blocks in a single shared memory region, so blocks with different owners or different ACLs are stored in different memory regions. For example, a separate shared memory region is created for each file.

Block references are pairs comprising a shared memory region ID and a page offset within the shared memory region. A process accesses the contents of a block by first extracting the shared memory region ID from the block reference, then invoking interfaces in the α-Nexus shared memory API. For mmap-oriented access, a process invokes the `shm_map` system call. For streams-oriented access, a process invokes `shm_read` or `shm_write` system calls.[10]

When a process requests access to data in an α-Nexus shared memory region, the requester provides the absolute offset and length of the desired data. Each MSFS component that accesses cached blocks is responsible for calculating such offsets for its own requests, so MSFS components invoke shared memory system calls directly when they need to access data. Similarly, for a client that uses mmap-oriented access to files, the client calculates the needed offsets and the client invokes `shm_map`. But for a client that uses streams-oriented access to files, VFS calculates offsets on behalf of the client; in this case, VFS invokes `shm_read` and `shm_write` system calls, and VFS transfers data to or from the client over an IPC channel.

MSFS employs a level of indirection—block references rather than block contents typically appear in messages between processes—and each process accesses and manipulates blocks indirectly through system calls. This indirection improves performance by avoiding copying. And the interposition of an API for accessing and manipulating blocks, in effect, exposes only a limited set of operations (hence, privileges) compared to a design in which block data is copied between processes and accessed directly by processes. For example, $\mathit{CacheMgr}$ holds privileges to initiate pre-fetching and eviction for blocks, but $\mathit{CacheMgr}$ does not hold privileges to read or write cached blocks. And a disk driver can't

[10]MSFS use of α-Nexus shared memory regions is similar to a more traditional implementation's use of a kernel buffer cache or page pool.

directly read or write blocks stored in the cache or on disk, even though it holds privileges for initiating DMA transfers between the block cache and disks. Thus, our use of indirection enables MSFS to better instantiate Least Privilege.

4.4.2 Policy Management Component

The MSFS $PolicyMgr$ component manages *policy meta-data*, which includes the owner and the ACL for each block stored in MSFS.[11] For each installed disk, $PolicyMgr$ maintains a `policy_table` data structure to store policy meta-data for that disk. Each entry in `policy_table` encodes a range of block IDs, a NAL principal name for the owner of those blocks, and an ACL enumerating NAL principal names and the privileges those principals hold.

MSFS DAC Policy dictates that access to a block—whether stored in the block cache or on disk—is allowed only if the requester appears on the appropriate ACL. Blocks stored on disk can only be accessed using disk transfers, so MSFS prohibits disk transfers except in certain limited situations, e.g., to flush a cached block to disk or to load a disk block into the block cache. The mechanism for enforcing these restrictions is described in Section 4.4.4. For blocks stored in the block cache, the shared memory guard that the α-Nexus kernel provides is sufficient for enforcing MSFS DAC Policy; we need only configure each shared memory region's owner and ACL. $PolicyMgr$ creates the shared memory regions used for the block cache, hence the $PolicyMgr$ process is the owner for these shared memory regions and, consequently, $PolicyMgr$ can configure each shared memory region's ACL. $PolicyMgr$ determines the contents of that ACL by reading `policy_table` for the blocks that are expected to be stored in the shared memory region. All those blocks must have the same ACL, otherwise $PolicyMgr$ refuses to create the shared memory region.

[11]Many filesystem drivers contain code to manage policy meta-data for the files and directories that they manage. However, the FAT32 filesystem does not directly support DAC and makes no provision for storing policy meta-data within a FAT32 partition, so such code was not present in the filesystem driver we adapted.

The complete `policy_table` is stored on the corresponding disk, so policy meta-data persists across reboots. Portions of `policy_table` are cached in memory and used when $PolicyMgr$ creates a shared memory region for the block cache. $PolicyMgr$ uses the same mechanisms to read or write policy meta-data on disk as used for all other disk accesses in MSFS. So the cached `policy_table` is actually stored in the block cache. This architecture makes bootstrapping MSFS a bit tricky since, as previously described, MSFS prohibits reading disk blocks except to load the block cache, but $PolicyMgr$ must read `policy_table` when creating the shared memory regions that hold cached blocks. We resolve this circular dependency by storing `policy_table` at a fixed, predetermined location on disk. On reboot, $PolicyMgr$ knows which disk blocks store `policy_table`, so it creates a shared memory region to cache those blocks along with an ACL that grants only itself access to that shared memory region. $PolicyMgr$ can then request the blocks storing `policy_table` be loaded from disk into the block cache, as needed, before creating additional shared memory regions for other blocks.

Requests to change block ACLs. $PolicyMgr$ implements an API for changing the ACL associated with a range of block IDs. A process can invoke a `change_acls` method over an α-Nexus IPC channel, specifying the range of block IDs and how the ACLs should change, e.g., modify the privileges in an existing ACL entry, add a new ACL entry, or delete an existing ACL entry. In response to a `change_acls` request, $PolicyMgr$ updates the ACLs in the cached copy of `policy_table` in memory and flushes the changes to disk. If any of the modified ACLs concern blocks in the block cache, then $PolicyMgr$ also invokes the kernel to update the kernel-maintained ACL associated with the shared memory region for those cached blocks. Thus $PolicyMgr$ ensures all copies of an ACL are consistent.

MSFS DAC Policy allows only a block's owner—or some principal that speaks for the block's owner—to change an ACL. So $PolicyMgr$ implements a guard to authorize `change_acls` requests. For each block ID d in the specified range, the policy enforced by the guard is represented by the NAL formula

$$\mathit{owner}(d) \textsf{ says } \texttt{change\_acls}(d, \ldots), \tag{4.8}$$

where $\mathit{owner}(d)$ is the NAL principal name for block d's owner, as found in `policy_table`. A request from process P to change an ACL is represented as P $\textsf{says}$ $\texttt{change\_acls}(d, \ldots)$. The guard allows a `change_acls` request to proceed only if $P \xrightarrow{\texttt{change\_acls}(d,\ldots)} \mathit{owner}(d)$ can be derived, since (4.8) can then be derived from that. A process P invoking `change_acls` can augment that request with a proof of $P \xrightarrow{\texttt{change\_acls}(d,\ldots)} \mathit{owner}(d)$, and the guard invokes NAL's automated proof checker to check that proof.

Since each `change_owner` request is conveyed over an α-Nexus IPC channel, *PolicyMgr* can leverage α-Nexus alias tables to check the policy in cases where $P \rightarrow \mathit{owner}(d)$. Before invoking `change_acls`, P installs $\mathit{owner}(d)$ in its alias table along with a proof of $P \rightarrow \mathit{owner}(d)$; P then specifies an index for the resulting alias when sending IPC requests to *PolicyMgr*. The *PolicyMgr* guard compares the NAL principal name that accompanies each such IPC request against $\mathit{owner}(d)$ for each block ID d in the specified range. If any of these comparisons fail to match, then the `change_acls` request is denied. For instance, consider a client P, executing on behalf of user *Alice*, that makes a request to change the ACL for the blocks storing the contents of a file owned by *Alice*. P first invokes the kernel to add *Alice* to its alias table along with a proof of $P \rightarrow \mathit{Alice}$. From this system call, P obtains an index into its alias table. Subsequently, P specifies this index when invoking the kernel to send a `change_acls` request over an IPC channel to *PolicyMgr*. If the proof of $P \rightarrow \mathit{Alice}$ is correct, then the kernel will attach *Alice* to the request and this will allow the guard in *PolicyMgr* to derive (4.8).

Requests to change block owners. *PolicyMgr*'s API includes methods for changing the owner for a range of block IDs. In a traditional filesystem, the `chown` system call changes a file's owner, and typically only a filesystem administrator is allowed to invoke `chown` on a file. MSFS enforces DAC for the filesystem administrator and does not have the filesystem administrator speak for all users or processes. So $PolicyMgr$ implements a different policy for changing a block's owner, as follows. Before changing the owner of a block d to some new NAL principal name A, a guard in $PolicyMgr$ checks the policy represented by

$$(owner(d) \textsf{ says } \texttt{chown\_nominate}(d, A)) \land (A \textsf{ says } \texttt{chown\_accept}(d)), \quad (4.9)$$

where $owner(d)$ is the current owner of block d as found in `policy_table`. Policy (4.9) stipulates that the current owner and new owner both consent to the change in ownership. Discharging (4.9) requires coordination between the current owner, $owner(d)$, and the proposed new owner, A. This coordination is done out of band.

Policy (4.9) can be discharged by obtaining two credentials, one for each conjunct. So $PolicyMgr$ implements two methods—$\texttt{chown\_nominate}(d, A)$ and $\texttt{chown\_accept}(d)$—that processes can invoke over an IPC channel.[12] MSFS leverages the α-Nexus alias table for processing these requests. The protocol for changing a block's owner is as follows. First, a process executing on behalf of the current owner adds $owner(d)$ to its alias table and sends an IPC $\texttt{chown\_nominate}(d, A)$ request to $PolicyMgr$, specifying a block ID d and a proposed new owner A. Upon receipt of this request, a guard checks that the NAL principal name that accompanies the IPC request equals $owner(d)$. If so, then the first conjunct of policy (4.9) is satisfied, and $PolicyMgr$ records A in a temporary variable, $nomination(d)$, for later use. Subsequently, a process executing on behalf of the proposed new owner adds A to its alias table and sends an IPC $\texttt{chown\_accept}(d)$ request to $PolicyMgr$. Upon receipt of this request, a guard checks that the NAL principal name that accompanies the IPC request equals $nomination(d)$. If so, both conjuncts of policy (4.9) have been discharged, and $PolicyMgr$ records $nomination(d)$ as the block's new owner in $policy\_table$, flushing the change to disk. At most one instance of $nomination(d)$ is stored for each block d, and these variables are discarded on reboot.

[12]For simplicity of presentation, we showed these methods as accepting a single block ID d. In practice, the methods accept ranges of blocks IDs.

Complete mediation for policy meta-data. MSFS may store as many as three copies of policy meta-data: (i) encoded in `policy_table` on disk, (ii) encoded in a cached copy of `policy_table`, and (iii) in the ACLs for shared memory regions that form the block cache. Copies (i) and (ii) are accessed using the same mechanisms as used for all other blocks, so MSFS's normal DAC enforcement mechanisms suffice for implementing Complete Mediation: MSFS assigns $PolicyMgr$ to be the owner of blocks storing `policy_table`, and $PolicyMgr$ specifies an ACL containing only itself for those blocks. Copy (iii) is stored within the kernel, and the kernel's shared memory guard only allows the owner of the corresponding shared memory region to change it. And because $PolicyMgr$ is the owner of the shared memory regions that form the block cache, only $PolicyMgr$ can directly change copy (iii) of the policy meta-data. $PolicyMgr$ only makes changes to any of the copies in response to requests that its own guard has authorized, i.e., after checking policy (4.8) for a `change_acls` request or checking policy (4.9) for a request to change a block's owner.

4.4.3 TCB for MSFS DAC Policy Enforcement

$PolicyMgr$ and $CacheMgr$ are both involved in managing cached blocks. We considered incorporating all cache-related code into a single component, in order to reduce performance costs. But we rejected that design after considering its impact on the TCB for MSFS DAC Policy enforcement. Code for managing policy meta-data is in this TCB, but most cache management code need not be. For example, code that implements block cache prefetching does not need privileges to modify ACLs. Thus, the current MSFS decomposition leads to a smaller, simpler TCB. We predicted that the performance costs of having the two separate components would be acceptable, given the infrequency of block allocation. But we did not perform experiments to directly measure these costs.

Because $PolicyMgr$ is in the TCB for MSFS DAC Policy enforcement, we included in

$PolicyMgr$ the remaining MSFS code necessary for enforcing DAC. $PolicyMgr$ makes sure cached blocks are flushed before they are deallocated, for example, and $PolicyMgr$ ensures that each block can be found at most once in the block cache to avoid cache aliasing issues. The result is that, aside from the α-Nexus kernel, $PolicyMgr$ (2267 lines of C code) is the only MSFS component that is in the TCB for MSFS DAC Policy enforcement.

4.4.4 Disk Driver Components

MSFS uses a component DD_b to execute disk driver code for each disk bus b. This decomposition is an example of privilege separation, because components are being defined based on security-relevant privileges they require—in this case, privileges to request I/O for a given disk bus. We considered decomposing disk drivers into finer-grained components by using a separate component for each disk rather than for each disk bus. But this decomposition would require the DDRM to distinguish I/O requests on a per-disk basis, and α-Nexus did not support that.[13]

DD_b holds privileges for initiating DMA transfers between the block cache and disks on bus b. MSFS DAC Policy implies that only certain transfers be allowed. A straightforward, but ultimately unsatisfactory, approach to restricting access to blocks on disk is to include a guard in each disk driver DD_b. This implements Complete Mediation, because only DD_b can request I/O operations for disk bus b. But this approach would also put DD_b in the TCB for MSFS DAC Policy enforcement, a bad idea given the size, complexity, and (historically) high rate of bugs in device driver code [34].

The approach we instead implemented in MSFS was to extend the DDRM with checks to enforce MSFS DAC Policy. This implements Complete Mediation, because all device I/O requests are checked by the DDRM. The approach also helps to minimize TCBs, because the TCB for MSFS DAC Policy enforcement already includes the DDRM, and we

[13]Attributing I/O requests to specific disks introduces significant dependencies on the specific type of disk controller hardware being used.

now add only a small amount of code to the kernel (hence, to all TCBs).

In addition to the regular DDRM checks, for I/O operations that instigate a DMA transfer between block having ID d and memory page m, the DDRM also checks whether:

(i) page m is within some shared memory region r, owned by some process P;

(ii) P speaks for $\mathit{PolicyMgr}$;

(iii) page m stores, or is expected to store, block d;

(iv) if transferring to memory, then page m is `empty`; otherwise, if transferring from memory, then page m is `filled` and `dirty`.

We modified α-Nexus shared memory code so that shared memory allocations for the block cache satisfy hardware constraints for DMA transfers. And we modified the DDRM to update the associated page status flag to `filled` (if it was not already) and the block status flag to `clean`.

The DDRM implements (ii) by searching P's alias table for an entry encoding a NAL principal name for $\mathit{PolicyMgr}$ accompanied by a correct proof of $P \rightarrow \mathit{PolicyMgr}$. If no such alias table entry is found, then the DDRM denies the I/O request. For stability across reboots, we define $\mathit{PolicyMgr}$ to be the NAL principal name

$$\{\!|\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\mathtt{pcrs}(\overline{h}).\mathtt{epoch}(\mathsf{p})\ \textsf{says}\ \mathtt{pgm\_hash}(\mathsf{v}, h_{\mathit{PolicyMgr}}))|\!\},$$

where $h_{\mathit{PolicyMgr}}$ is the hash of a program manifest describing the MSFS code for $\mathit{PolicyMgr}$. This definition allows the DDRM to ensure $\mathit{PolicyMgr}$ owns the shared memory region, even though the NAL name for the process P that executes $\mathit{PolicyMgr}$ changes on each α-Nexus reboot.

Checks (iii) and (iv), above, depend on meta-data associated with each page in the block cache, including the page's block ID, the page status flag (`empty` or `filled`), and the block status flag (`clean` or `dirty`). We considered two implementations.

- The meta-data could be stored in an MSFS component, requiring the kernel to access some process's memory whenever the kernel needs to access the meta-data.
- When *PolicyMgr* invokes the kernel to create a shared memory region, it specifies the block IDs for each page (page status flags are always `empty` initially), and the kernel stores and manages the meta-data thereafter.

We chose the later, because a component that manages the meta-data would become part of the TCB for MSFS DAC Policy, whereas the kernel is already part of this TCB. Moreover, the kernel code to manage the meta-data is actually simpler, so likely less error-prone, than the code needed for the kernel to access meta-data that is stored and managed by a process. Finally, the kernel accesses this meta-data more frequently than other components, so it pays to locate the meta-data within the kernel.

4.4.5 Filesystem Driver Components

The filesystem driver layer manages file and directory meta-data for the filesystem stored by each disk partition. A single component implementing all filesystem drivers would have to hold privileges to access blocks in every disk partition, which is inconsistent with Least Privilege. So we used privilege separation and decomposed this layer into multiple components. In MSFS, each disk partition r has a separate component FSD_r. FSD_r executes the filesystem driver code given by that partition's type code, and FSD_r is configured with the range of block IDs that define that partition.

When a disk in MSFS is first formatted, the *DiskFormatter* component becomes owner of all blocks on the disk. *DiskFormatter* creates a partition table and, for each partition r in the partition table, invokes `chown_nominate` to propose a new owner, FSD_r, for the blocks in that partition. FSD_r then invokes `chown_accept` to become owner of these blocks. For stability across reboots, the NAL principal name used for FSD_r is a NAL group:

$$\{\!|\mathsf{v} : (\exists \mathsf{p} : K_{CPU}.\texttt{pcrs}(\overline{h}).\texttt{epoch}(\mathsf{p}) \textsf{ says } \texttt{pgm\_hash}(\mathsf{v}, h_{FSD_r}))|\!\}.$$

As with *PolicyMgr*, the group definition includes a hash h_{FSD_r} computed over a program manifest, where the program manifest describes the code being executed by FSD_r. The manifest here also includes the range of blocks IDs that define partition r, so that filesystem driver components configured to manage different partitions will have different NAL names, hence will hold different privileges, even if the components execute the same filesystem driver code.

The top layer of the filesystem, VFS, employs an IPC channel to invoke FSD_r for various file and directory operations. When a process P, executing on behalf of principal A (e.g., a user), requests that a file be created or enlarged, FSD_r invokes `chown_nominate` to propose that A become owner of blocks that will store contents of that file. Process P then invokes `chown_accept` to become owner of those blocks. These roles are interchanged when a file is truncated or deleted. Once the owner of the file becomes the owner of blocks storing the contents of the file, the α-Nexus shared memory guard and *PolicyMgr* enforce DAC for those blocks. Thus FSD_r is not in the TCB for MSFS DAC Policy enforcement.

Users and other principals own the contents of files, but FSD_r owns other blocks in partition r, including blocks that store directories. We could have further decomposed MSFS. One alternative would be to employ separate components for directory management. Or, using domain decomposition, we could have separated instances of directory management code for different directories into different components. These alternative architectures could allow users and other principals to own blocks storing directories, thereby permitting FSD_r to hold fewer privileges. The potential benefits of finer grained components, however, are offset by the disadvantage of creating dependencies on the particular filesystem format—FAT32—used in our prototype.

4.4.6 VFS Components

The VFS layer manages state on behalf of filesystem clients. A collection of one VFS_P component for each client P comprise the layer. Client P invokes VFS_P to perform various file and directory operations, and VFS_P implements a guard to ensure requests only from P are performed.

Client P always executes on behalf of some principal A, a user or a group of users, and P relies on this delegation when accessing the filesystem. In MSFS, the trust A places in client P is, by default, specified by the NAL formula $P \to A$. So P adds A to its alias table, along with a proof of $P \to A$. Client P also issues a credential that conveys NAL formula P says $(VFS_P \to P)$, from which $VFS_P \to P$ follows. Together with $P \to A$, this implies $VFS_P \to A$, because $\to$ is transitive. Thus VFS_P can also use A as an alias and make request to filesystem drivers and to α-Nexus shared memory regions on behalf of A.

If client P uses an mmap-oriented interface to access files, then P must invoke the `shm_map` system call directly, rather than having VFS_P make the request on its behalf. This is necessary on α-Nexus, because the kernel creates the virtual memory mappings in whichever process invoked the system call. In this case, P uses alias A to satisfy the kernel's shared memory guard, since A is presumably on the ACL for the blocks in question.

Using a separate VFS_P component for each client P helps reduce the TCB size for any security goal a client might have. For instance, if VFS_P becomes compromised, then security goals relating to client P may be violated (recall that $VFS_P \to P$ means that P has placed full trust in VFS_P). But some other client P' need not have placed any trust in VFS_P, so P' need not be affected by the compromise of VFS_P. Were the VFS layer a single component, then a compromised VFS layer could cause security goals to be violated for all clients that use MSFS.

VFS_P is responsible for specifying the identity of A to other components—by adding alias A to its alias table and by selecting that alias when sending IPC requests—yet VFS_P need not be trusted by the recipients of these requests, because the kernel checks that $P \rightarrow A$ holds during each IPC system call. So absent a proof of $VFS_P \rightarrow A'$, for some other principal A', VFS_P can't substitute a bogus identity A' when making requests, even if VFS_P is compromised. By contrast, were the entire VFS layer a single component, VFS would need to be trusted to chose the right identity from among many valid alias table entries when making a request on behalf of a client.

In our MSFS implementation, P places full trust in VFS_P. Least Privilege would favor a design in which a client P does not place full trust in VFS_P, but instead grants P only enough privileges to perform its task. These privileges would include only privileges to make requests to various MSFS components and α-Nexus shared memory regions on behalf of P. There are several ways to accomplish this in NAL, but we found it unnecessary to implement these restrictions in light of optimizations described in Section 4.4.8.

4.4.7 Data Replication in MSFS

In our decomposition of MSFS into components, certain data is used by multiple components. The VFS components share a `mount_table` data structure containing information about each mounted filesystem. And filesystem driver and disk driver components share `io_request_queue` data structures. A straightforward implementation would use α-Nexus shared memory to store these shared data structures. But this restricts authorization to only two types of privileges—read or read/write—and these privileges are enforced at a relatively coarse granularity—4KB memory pages. Least Privilege would favor a design where MSFS components hold fewer and more fine-grained privileges. For example, while VFS components executing on behalf of the administrator can create or delete entries in `mount_table`, other VFS components only read the entries and increment or decrement various reference counts. Similarly, only filesystem drivers insert entries into an `io_request_queue`, and only disk drivers mark those entries as completed.

We avoid the limitations concerned with authorizing access to a common data structure by replicating information across different data structures. Different privileges are then associated with the different replicas. MSFS employs this approach, using α-Nexus IPC channels to keep the replicas synchronized. For `mount_table` and `io_request_queue` data structures, straightforward coherence protocols suffice to accommodate the different privileges held by the components that need access. The performance cost of this approach to Least Privilege is the overhead for implementing a coherence protocol for the replicas.

4.4.8 MSFS Implementation Optimizations

In building MSFS, we explored a few techniques that improve performance without sacrificing many benefits to system trustworthiness that Mutual Suspicion brings. The optimizations we consider admit slightly less aggressive instantiation of security principles in return for large gains in performance.

Shortening Communication Paths

We can sometimes gain efficiency without much changing the privileges each component must hold and without enlarging TCBs much. For example, consider the steps involved in handling client P's request to access a cached file.

1. P sends a request to VFS_P.
2. VFS_P calculates the current file offset and sends a request to FSD_r.
3. FSD_r calculates a block reference for the cached data and responds to VFS_P.
4. VFS_P accesses the block by invoking the α-Nexus shared memory API with the shared memory region ID contained in the block reference, then responds to P.

But P can easily perform the offset calculations done by VFS_P in step 2. Moreover, the identity of FSD_r is known at the time a file is opened and is not secret from P. So we can eliminate VFS_P and use a different sequence of steps:

1′. P calculates the current file offset and sends a request to FSD_r.

2′. FSD_r calculates a block reference for the cached data and responds to P.

3′. P accesses the block by invoking the α-Nexus shared memory API with the shared memory region ID contained in the block reference.

This optimization leads to fewer IPC messages during file accesses, at the cost of duplicating in the client some VFS functionality (i.e., maintaining the current file offset). However, the new client code is not likely to change the TCB for client security goals, since the client already places full trust in VFS_P, which implements nearly identical code. Moreover, the duplicate functionality is straightforward to implement in the client, so even a small gain in performance justifies this optimization.

As a further optimization, we amortize the overhead of step 2′ when P makes multiple accesses to the same file. When a file is first opened, P invokes FSD_r to obtain a block reference for the first block of the file, and P caches the shared memory region ID contained within that block reference. During subsequent accesses to the same file, P uses the previously cached shared memory region ID rather than contacting FSD_r.

Notice, these optimizations do not give P additional privileges. It may seem that P gains the ability to make arbitrary changes to the current file offset or to influence the offset calculations, since these are now located within P. But in fact, P can already completely determine the output of the offsets calculations done by VFS_P—P need only invoke the `seek` method. P can also already access shared memory regions, as it does for mmap-oriented file access. Thus there is no way for P to violate MSFS DAC Policy. Even if P were to perform incorrect offset calculations or access the wrong shared memory region, the only consequence would be that P receives incorrect data or its request is rejected when it makes the system call to access the shared memory region.

Leveraging Fate Sharing

Fate sharing [158]—which occurs when failures are not independent—creates an opportunity to improve performance without sacrificing security. In particular, if components place full trust in each other, then isolating them from each other contributes nothing to security. By merging these components, we eliminate overhead. One simple example in MSFS is the DDRM, which executes in the kernel rather than as a process outside of the kernel. Even were they isolated from each other, the compromise of the kernel or of the DDRM could lead to the compromise of the other. So isolating these components would not increase trustworthiness.

We also leverage fate sharing by merging each client P with the corresponding MSFS component VFS_P. To achieve this, we incorporated VFS code into the standard C library used by α-Nexus programs—that change is transparent to application programmers. Eliminating the isolation boundary between P and VFS_P allows IPC calls to be replaced with more efficient local function calls. This optimization has two potential consequences: P can now cause the compromise of VFS_P; and VFS_P can now cause the compromise of P. The former is not a concern in MSFS, because P holds at least as many privileges as VFS_P. We accept the later, because VFS code in MSFS is quite small and simple, so it is unlikely to cause a compromise.

Relocating Guards

By changing how and when authorization checks are performed, we can change their performance overhead. One approach is to amortize some or all of the work done for checks, with a single check serving for multiple requests. A canonical example with filesystems is when access control checks are only performed when a file is first opened, rather than for every access.

MSFS amortizes the cost of checks done for I/O requests that initiate DMA transfers. Rather than have the kernel check whether the process that owns a shared memory region speaks for $PolicyMgr$ before each DMA transfer, the kernel performs this check only when shared memory regions for the block cache are allocated. The use of the α-Nexus alias table abstraction is also a form of amortization, because some of the proof checking done by the kernel is performed when an alias table entry is created rather than each time it is used.

4.5 Filesystem Evaluation

One way to quantify the consequences of various design decisions made in MSFS is by calculating TCB sizes of various prototype implementations and comparing that to traditional filesystem implementations. A second way is to measure and compare the performance of these systems. The results of both—detailed below—confirm that instantiating security principles for MSFS using NAL and credentials-based authorization did not result in measurably worse performance.

4.5.1 TCB Contributions

With so many guards and the need for communication across isolation boundaries, one might expect the MSFS code base to be larger than a monolithic kernel-mode filesystem. So we counted the lines of code that implement MSFS along with the α-Nexus code for system services closely related to supporting MSFS. These counts are shown in Figure 4.4 (a). Then, as a point of comparison, we examined two alternative designs, here called UFS and KFS, that support the same filesystem interface but with different components.

MSFS Kernel-mode LOC	
Shared memory	1900
DDRM	4658
MSFS User-mode LOC	
User-mode driver lib.	49839
SATA disk driver	28216
FAT32 fs driver	7230
VFS layer	1915
Policy management	2267
Other filesystem code	4562
MSFS Total LOC	100587

(a)

UFS Kernel-mode LOC	
Shared memory	1900
DDRM	4658
UFS User-mode LOC	
User-mode driver lib.	49839
SATA disk driver	28029
FAT32 fs driver	6442
VFS layer	771
Policy management	1689
Other filesystem code	4081
UFS Total LOC	97409

(b)

KFS Kernel-mode LOC	
Shared memory	1773
SATA disk driver	28029
FAT32 fs driver	6442
VFS layer	771
Policy management	1689
Other filesystem code	4081
KFS User-mode LOC	
n/a	0
KFS Total LOC	42785

(c)

Figure 4.4: Lines of code (LOC) for MSFS (a) and two alternative designs, UFS (b) and KFS (c). All filesystem code is counted. Some α-Nexus system services that are closely related to supporting these filesystems is also counted. KFS counts are estimates.

- Figure 4.4 (b) shows code sizes for UFS, an implementation that places most filesystem code in a single component executing as a process above the α-Nexus kernel. We obtained the UFS implementation from our MSFS implementation by eliminating some guards and replacing most IPC-related code with equivalent code using local function calls.

- Figure 4.4 (c) shows estimates for code sizes that would result from implementing a traditional kernel-mode filesystem design, KFS. In this design, all filesystem code executes within the α-Nexus kernel. We did not implement the KFS design, because α-Nexus does not support executing disk drivers within the kernel.

As expected, Figure 4.4 shows that instantiation of security principles in MSFS leads to increased total code size. This is seen in the 100587 total lines of code (LOC) for MSFS, compared to 97409 LOC for UFS and 42785 LOC for KFS, designs that instantiate Least Privilege and Mutual Suspicion less aggressively than MSFS.

Figure 4.4 likely exaggerates the impact on total code size that is caused by instantiating security principles. First, we did not include counts for the large body of kernel code common to all of the designs. If that code were included, then the relative LOC differences would be much smaller. Second, the large increase in total code size above KFS

	α-Nexus			Linux		
	MSFS (mutual suspicion)	UFS (monolithic user-mode)	KFS (monolithic kernel-mode)	(mutual suspicion)	FUSE (monolithic user-mode)	FAT32 (monolithic kernel-mode)
Contrib. to TCB for DAC	8825	97409	42785	-	78695	46801
Contrib. to TCB for isolation	6558	6558	42785	-	46609	46801

Figure 4.5: Lines of code (LOC) contributed by various filesystem designs to the TCB for two security goals. For all designs, enforcing DAC also requires enforcing isolation. So the TCB for DAC is a strict superset of the TCB for isolation. Counts for Linux do not include disk driver code.

is due mostly to the user-mode driver library (49839 LOC). The SATA disk driver used in MSFS and UFS requires this library, because the driver was originally programmed against Linux's kernel-mode driver API rather than against α-Nexus's user-mode driver API.[14] Much of the user-mode driver library code (roughly 85%) provides a compatibility layer that emulates Linux's kernel-mode driver API, and that code would not be necessary had we implemented a user-mode SATA disk driver for α-Nexus from scratch. The remainder of the user-mode driver library code (roughly 15%) implements helper routines for handling interrupts, I/O requests, and thread scheduling (e.g., locks and mutexes).

If we exclude the entire user-mode driver library, then the LOC for UFS is 4785 lines larger than for KFS. Most of this additional code for UFS is for the DDRM (4658 LOC), which implements checking for I/O requests, interrupt handling, and other device driver concerns. This code is present in MSFS as well. The LOC for MSFS is 3178 lines larger than for UFS. The additional code implements guards that check requests between MSFS components, code that is not needed for UFS.

If the larger size of MSFS made that system more vulnerable to compromise, comparing performance with smaller-size systems would be pointless. We calculated the contributions to the size of the TCB for MSFS, UFS, and KFS relative to each of two security

[14]For KFS, we assume the size of an kernel-mode driver for α-Nexus would be comparable to the kernel-mode SATA disk driver for Linux.

goals: enforcement of MSFS DAC Policy, and the integrity of kernel and process isolation boundaries. We also made estimates for two filesystems running on Linux [88].

- Linux FUSE: A Linux user-mode implementation of the EXT2 filesystem based on FUSE [133]. This system is a Linux analog to α-Nexus's UFS design.
- Linux FAT32: The standard Linux implementation of the FAT32 filesystem using a traditional kernel-mode design. This is the Linux analog to α-Nexus's KFS design.

For Linux implementations, we calculated the contribution to the TCB for Linux's DAC policy, rather than for the MSFS DAC Policy. We include only code relating to the filesystem and exclude the bulk of the kernel code base. We also excluded Linux disk driver code—Linux supports many such drivers, but they always execute within the Linux kernel. We thereby highlight the impact of different filesystem designs on the relative sizes of TCBs.

Figure 4.5 shows the results. Although from Figure 4.4 (a) we see that MSFS has the largest code base of the three α-Nexus designs (MSFS, UFS, KFS), Figure 4.5 shows that MSFS adds the least amount of code to TCBs. Only 6558 LOC for MSFS—about 7% of the MSFS code base—is in the TCB for integrity of the kernel or of other processes executing above the kernel, and only 8825 lines of MSFS code are part of the TCB for enforcing MSFS DAC Policy. KFS has the least amount of code (42785 LOC, from Figure 4.4 (c)) among the α-Nexus designs, but by situating all of this code within the kernel, KFS achieves only a moderately sized TCB for both security goals, as shown in Figure 4.5. UFS represents a partial instantiation of the security principles that drive the design of MSFS. And UFS achieves a smaller TCB than KFS for kernel and process isolation (6558 LOC and 42785 LOC, respectively, from Figure 4.5), but UFS requires a large amount of code—97409 lines—in the TCB for MSFS DAC Policy enforcement versus 42785 LOC for KFS.

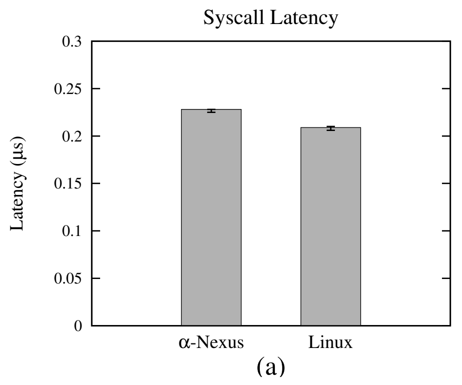
The TCB contributions shown in Figure 4.5 for the three α-Nexus designs indicate that pervasively instantiating security principles can reduce TCB size in comparison to a traditional kernel-mode design or a design that only partially instantiates security principles. Although we do not have a Linux filesystem analog to MSFS, the TCB contributions shown in Figure 4.5 for the two Linux implementations are consistent with this conclusion. Linux FUSE—an analog of UFS—has a smaller TCB for kernel and process isolation than Linux FAT32—an analog of KFS—though only slightly (46609 LOC and 46801 LOC, respectively, from Figure 4.5). And for DAC enforcement, Linux FUSE requires a large amount of code—78695 lines—in the TCB versus 46801 LOC for Linux FAT32.[15]

One might be concerned about our choice of FAT32 as the basis for MSFS. Would a more modern filesystem format like EXT3 change our conclusions, because FAT32 is substantially simpler than EXT3? The same concern arises when comparing Linux FUSE (which implements EXT2, the predecessor to EXT3) and Linux FAT32. So we measured the Linux EXT3 code base which, like Linux FAT32, follows a traditional kernel-mode design. We found that it contributes 53654 LOC to each TCB. This is indeed more than the 46801 LOC contributed by Linux FAT32. But this only means our use of FAT32 likely causes the α-Nexus results in Figure 4.5 to understate the case for design driven by Mutual Suspicion. If we replaced MSFS's FAT32 implementation with EXT3, the code base of MSFS would likely be even larger than it currently is, and the TCBs for UFS and KFS would also be larger. But MSFS would achieve the same small TCBs, since none of the code in MSFS TCBs depends on the choice of filesystem format.

4.5.2 The Cost of Isolation

Filesystem clients and MSFS components communicate with each other by sending IPC messages, accessing shared memory, and invoking system calls. So the performance of MSFS depends on the performance of these communications channels. We implemented four micro-benchmarks to quantify the performance cost of communication between iso-

[15]If we had counted device driver code for Linux, as we did for α-Nexus, all of the Linux TCBs shown in Figure 4.5 would appear larger by a constant amount. This does not change our conclusions.



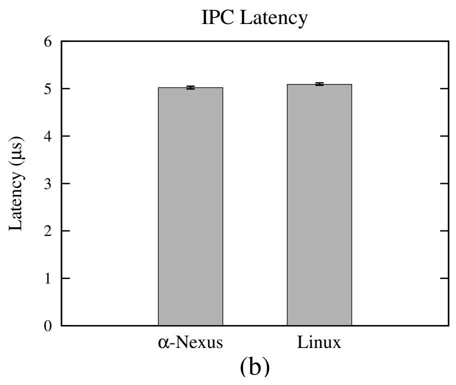
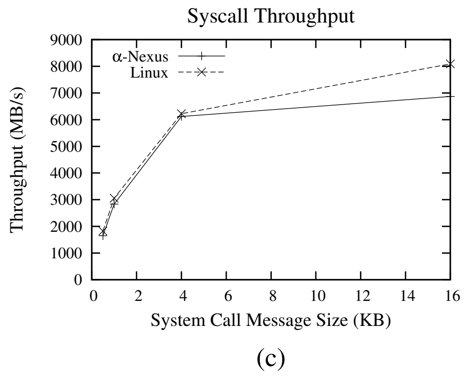


Figure 4.6: Syscall latency (a) and IPC latency (b) micro-benchmark results for α-Nexus and Linux, showing median result for each experiment. Error bars show the range in which the middle 80% of results fall.



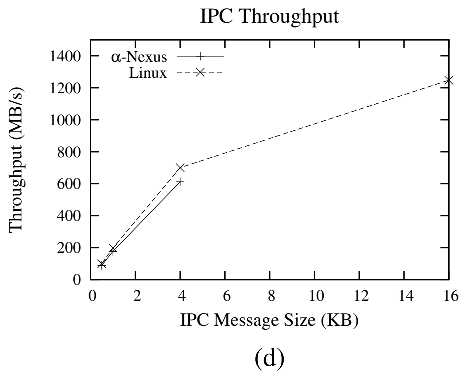


Figure 4.7: Syscall throughput (a) and IPC throughput (b) micro-benchmark results for α-Nexus and Linux, showing median result for each experiment. Variability (not shown) was small, with 80% of trials falling within $\pm 3\%$ of the median. Note that α-Nexus does not support IPC messages larger than 4 KB.

lated components in MSFS. These micro-benchmarks measure system call and IPC performance; we consider shared memory below, in Section 4.5.3 and Section 4.5.3. We ran the micro-benchmarks on α-Nexus and Linux, and we compared the results in order to confirm that α-Nexus system call and IPC performance is not unusual. The four micro-benchmarks we implemented are as follows.

- *Syscall latency*: Perform a single null system call.

- *IPC latency*: Send a single-byte request to another process over an IPC channel[16] and receive a single-byte response.
- *Syscall throughput*: Transfer 16 MB from the kernel to a process using system calls.[17]
- *IPC throughput*: Transfer 16 MB from one process to another using request-response pairs sent over an IPC channel.[18]

For the two throughput micro-benchmarks, we vary the amount of data transferred by each system call or request-response pair from 512 bytes to 16 KB. All experiments were performed on a 2.66 GHz Intel Core 2 platform with 3 GB of RAM. We performed 100 trials for each experiment. Results for the syscall and IPC latency micro-benchmarks are shown in Figure 4.6, and results for the syscall and IPC throughput micro-benchmarks are shown in Figure 4.7.

α-Nexus versus Linux Micro-benchmark Performance

Comparing the results for α-Nexus versus Linux across all four micro-benchmarks, the largest differences between the two operating systems occurs in the syscall latency micro-benchmark (0.228 µs median latency on α-Nexus versus 0.209 µs median latency on Linux, from Figure 4.6 (a)) and in the syscall throughput micro-benchmark (for 16 KB messages, 6869 $\mathrm{MB/s}$ median throughput on α-Nexus versus 8090 $\mathrm{MB/s}$ median throughput on Linux, from Figure 4.7 (a)). In the later case, the difference between α-Nexus and Linux is most pronounced when transferring 16 KB messages from the kernel to a process, and there is a much smaller difference in median throughput for α-Nexus versus Linux when

[16]We use pipes to implement IPC on Linux.

[17]On Linux, reads to `/dev/zero` are used for this micro-benchmark. On α-Nexus, an equivalent system call, unrelated to MSFS, is used instead. This micro-benchmark is meant to measure the speed at which kernel data can be written into a process's memory. So for each system call, the kernel zero-fills a message buffer specified by the micro-benchmark client, and the client does not copy or otherwise access the data returned from each system call.

[18]This micro-benchmark is meant to measure the speed at which one process's data can be copied into another process's memory over a α-Nexus IPC channel or Linux pipe. So for each send or receive system call, the kernel copies a request or response messages from one process's memory to the other's, but neither process copies or otherwise accesses the request or response messages.

transferring smaller messages. α-Nexus imposes a 4 KB maximum message size for IPC, so we were unable to compare α-Nexus and Linux IPC throughput using 16 KB messages. Therefore, to minimize confounding effects, in the remainder of this paper we limit transfers to at most 4 KB per system call and 4 KB per IPC message.

Despite some differences, the overall similarity between α-Nexus and Linux microbenchmark performance suggests that differences between α-Nexus and Linux system call and IPC mechanisms are unlikely to account for any large differences in performance for filesystems running on α-Nexus versus filesystems running on Linux.

System Call versus IPC Performance

The results in Figures 4.6 and 4.7 illustrate a familiar trade-off between isolation and performance. IPC allows senders and recipients to be isolated from each other, while the system call mechanism only isolates the kernel from processes. But system calls achieve better performance than IPC. On α-Nexus, for instance, system calls have lower latency than IPC—0.228 µs median syscall latency, from Figure 4.6 (a), versus 5.02 µs median IPC latency, from Figure 4.6 (b). And α-Nexus system calls have higher throughput than IPC—6116 MB/s median syscall throughput, from Figure 4.7 (a), versus 612 MB/s median IPC throughput, from Figure 4.7 (b), both using 4 KB messages.

Compared to a monolithic kernel-mode filesystem, such as Linux FAT32, MSFS replaces some operations that involve system calls with operations that involve IPC, and it replaces some operations that require only local function calls with operations that require system calls or IPC. Consider, for instance, the file `open` and `close` operations. In MSFS, each of these operations requires at least one IPC message (5.02 µs median latency on α-Nexus, from Figure 4.6 (b)), whereas in Linux FAT32, each operation requires only a system call (0.209 µs median latency on Linux, from Figure 4.6 (a)). So we should expect the median latency of `open` or `close` operations to be at least 5.02 µs $-$ 0.209 µs $=$ 4.81 µs slower in MSFS than in Linux FAT32.

4.5.3 Filesystem Performance

We measured the performance of MSFS and Linux FAT32 implementations using four benchmarks:

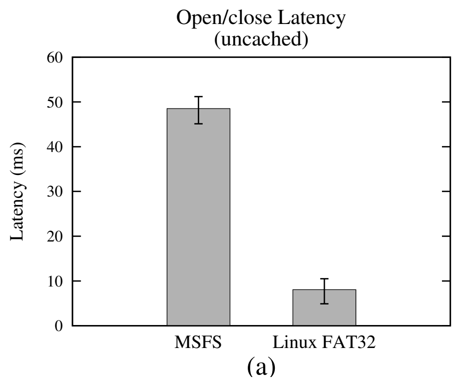
- *Open/close latency*: Open and close a 1 MB file.
- *Read latency*: Read a single byte from an already-open 16 MB file using the streams-oriented filesystem interface.
- *Read throughput*: Read all bytes from an already-open 16 MB file using the streams-oriented filesystem interface.
- *Enumerate latency*: Enumerate 4096 files and 1365 directories in a 5-level tree.

The benchmarks are implemented by a client that can run in two configurations. For the *uncached configuration*, the benchmark client requests that all caches (including the filesystem's block cache and the disk's internal data cache) be emptied before each trial. For the *cached configuration*, the benchmark client loads the caches before each trial so that all requests are satisfied by the filesystem's block cache and no disk I/O is performed. The experiments used a single 160 GB, 7200 RPM SATA disk with a FAT32-formatted partition. According to manufacturer specifications, the disk hardware can achieve 78 $\mathrm{MB/s}$ sustained read throughput and has an average seek latency of $4.16\ \mathrm{ms}$.

Open/Close Latency

Results for the open/close latency benchmark are shown in Figure 4.8.

Uncached open/close latency. For the uncached configuration of the open/close latency benchmark, Figure 4.8 (a) shows that MSFS achieves a median latency of $48.5\ \mathrm{ms}$, which is approximately $6x$ worse than the $8.07\ \mathrm{ms}$ median latency achieved by Linux FAT32. We conjectured that this difference is largely due to the difference in how MSFS and Linux



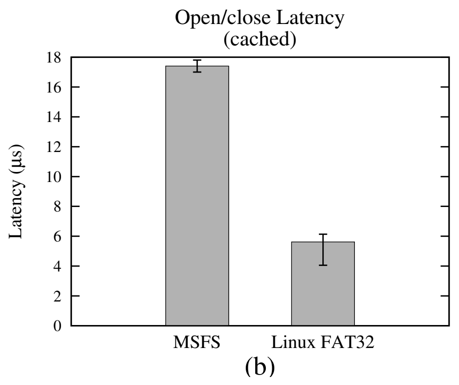


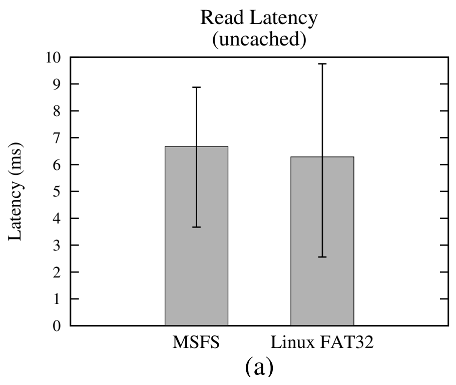
Figure 4.8: Open/close latency benchmark performance of MSFS and Linux FAT32 for the uncached (a) and cached (b) benchmark configurations, measured in milliseconds (ms) and microseconds (µs), respectively. Each box shows the median latency achieved for one experiment, and error bars show the range in which the middle 80% of results fall.

FAT32 enforce DAC. Linux FAT32 enforces DAC only for the contents of files and directories. And because of limitations of the FAT32 filesystem, Linux FAT32 does not retrieve policy meta-data from the disk. Instead, Linux FAT32 uses a single pre-configured owner and ACL for all files and directories. By contrast, MSFS enforces DAC for all data on disk. And for every block accessed by MSFS's FAT32 filesystem driver, FSD_r, MSFS's $PolicyMgr$ requires several additional accesses to retrieve that block's owner and ACL from the `policy_table` stored on disk. There is little cache locality in these accesses for a single benchmark trial, so the cost of these disk accesses adds up.

We performed two experiments to confirm our conjecture that MSFS DAC enforcement accounts for the observed differences in performance of MSFS and Linux FAT32 for the uncached configuration of the open/close latency benchmark. First, we counted how many disk access requests were initiated by MSFS and by Linux FAT32. For MSFS we counted 12 accesses for each uncached trial, 10 of which were initiated by $PolicyMgr$ and 2 of which were initiated by FSD_r. For Linux FAT32 we counted only 2 accesses for each uncached trial. If disk accesses are random, and if the disk's seek latency dominates performance costs in the uncached open/close latency benchmark, then, based on the $4.16\ \mathrm{ms}$ average seek latency claimed by the disk manufacturer, we should expect MSFS to require about $12*4.16$ ms $= 49.92$ ms and Linux FAT32 to require about $2*4.16$ ms $= 8.32$ ms. These numbers are close to the median observation (48.5 ms and 8.07 ms for MSFS and Linux FAT32, respectively, from Figure 4.8 (a)). This suggests that the difference in performance we measured is indeed due to the different extents to which the two implementations enforce DAC. It also suggests that MSFS performance could benefit by reducing the number of disk accesses it requires to retrieve policy meta-data.

To confirm, we performed a second experiment, using a modified version of MSFS. We replaced *PolicyMgr* with a different implementation, *ConstPolicyMgr*. *ConstPolicyMgr* performs no disk accesses. Instead, it assumes a single pre-configured owner and ACL for every block, thereby enforcing a DAC policy similar to what is enforced by Linux FAT32. The modified version of MSFS achieved 8.58 ms median latency for the uncached configuration of the open/close latency benchmark, a performance result that is much closer to Linux FAT32 (8.07 ms median latency, from Figure 4.8 (a)) than to MSFS (48.5 ms median latency, from Figure 4.8 (a)). This measurement adds further support to our conjecture about the source of the difference in uncached open/close latency benchmark performance observed between MSFS and Linux FAT32.

Cached open/close latency. Disk seek latency should not affect performance for the cached configuration of the open/close latency benchmark, because no disk I/O is performed in that configuration. Figure 4.8 (b) shows that MSFS achieves a median cached open/close latency of 17.4 µs. This is approximately $3x$ worse than the 5.62 µs median latency shown in the same figure for Linux FAT32. For comparison, MSFS with *ConstPolicyMgr* achieves a median latency of 17.1 µs for the cached configuration of the open/close latency benchmark, which is only a slight improvement over MSFS's 17.4 µs. This means DAC enforcement by *PolicyMgr* is not likely to be the source of the $3x$ difference between MSFS and Linux FAT32 cached open/close latency benchmark performance.



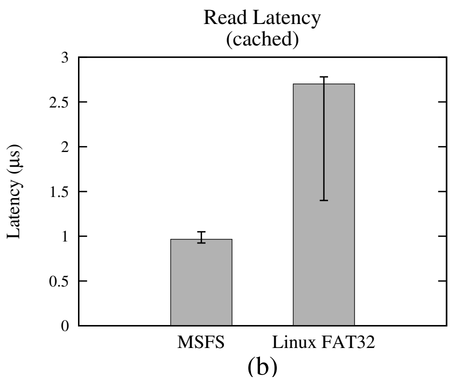
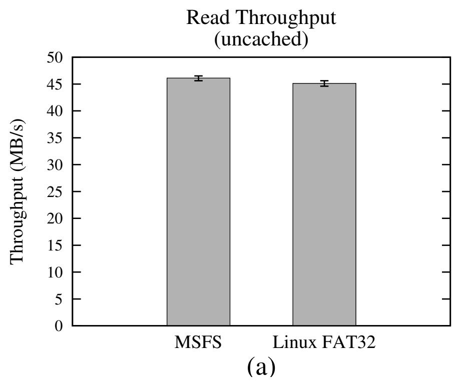


Figure 4.9: Read latency benchmark performance of MSFS and Linux FAT32 for the uncached (a) and cached (b) benchmark configurations, measured in milliseconds (ms) and microseconds (µs), respectively. Each box shows the median latency achieved for one experiment, and error bars show the range in which the middle 80% of results fall.

We conjectured that the $3x$ difference (17.4 µs for MSFS versus 5.62 µs for Linux FAT32) reflects the cost of IPC between isolated components in MSFS as compared with the cost of local function calls in Linux FAT32. For cached trials, `open` and `close` operations in MSFS each involve one IPC message, whereas these operations each involve only a single system call for Linux FAT32. From Figure 4.6 and the calculations in Section 4.5.2, the overhead of these IPC messages above system calls is about $2 * 4.81$ µs $= 9.62$ µs. This accounts for most of the 17.4 µs $- 5.62$ µs $= 11.78$ µs difference between MSFS and Linux FAT32 cached open/close latency benchmark performance.

Read Latency and Read Throughput

File reads in MSFS and in Linux FAT32 involve system calls and not IPC. A Linux client invokes the VFS layer, which resides in the Linux kernel, using the `read` system call. An α-Nexus client P reads a file in MSFS by invoking VFS_P using a local function call, since VFS_P resides within client P's address space, and VFS_P in turn accesses the MSFS block cache using the `shm_read` system call. So given the similarity between the system call performance of α-Nexus and Linux (as discussed previously in Section 4.5.2), we should



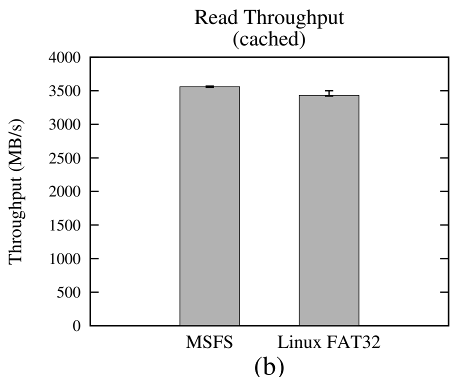


Figure 4.10: Read throughput benchmark performance of MSFS and Linux FAT32 for the uncached (a) and cached (b) benchmark configurations. Each box shows the median latency achieved for one experiment, and error bars show the range in which the middle 80% of results fall.

expect that MSFS and Linux FAT32 will have similar filesystem read performance, as well. Figure 4.9 shows results for the read latency benchmark, and Figure 4.10 shows results for the read throughput benchmark.

Uncached read latency and throughput. For the uncached configuration of the read latency benchmark, Figure 4.9 (a) shows a median latency of 6.67 ms for MSFS versus 6.29 ms for Linux FAT32, a difference of about 6%. Here, MSFS and Linux FAT32 both exhibit high variability—the slowest 10% of reads for Linux FAT32 take longer than 9.75 ms while the fastest 10% take less than 2.56 ms, for example. We attribute the high variability to variability in disk seek latency, which should be distributed uniformly (assuming random seeks) between about 0 ms and $2 * 4.16 \text{ ms} = 8.32$ ms, given the 4.16 ms average seek latency claimed by the disk manufacturer. For the uncached configuration of the read throughput benchmark, Figure 4.10 (a) shows a median throughput of 46.1 MB/s for MSFS versus 45.1 MB/s for Linux FAT32, a difference of about 3%. These results are not surprising since uncached read performance for both MSFS and for Linux FAT32 is likely constrained largely by disk performance.[19]

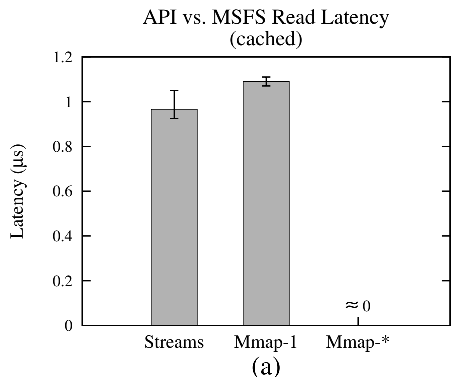
[19]Neither MSFS nor Linux FAT32 appears able to achieve the 78 MB/s sustained read throughput that is claimed by the disk manufacturer. We did not investigate this discrepancy further.

Cached read latency and throughput. Figure 4.9 (b) shows a marked difference between cached read latency achieved by MSFS versus Linux FAT32—MSFS achieves a median cached read latency of $0.996\ \mu\mathrm{s}$ versus $2.70\ \mu\mathrm{s}$ for Linux FAT32. However, latency for Linux FAT32 shows high variability, with 10% of trials measuring $1.40\ \mu\mathrm{s}$ or less, a value much closer to the median cached read latency in MSFS ($0.996\ \mu\mathrm{s}$). Such variability in Linux performance could be due to scheduling and lock contention between the filesystem and background processes, drivers, and interrupts. α-Nexus is a research prototype and, as such, executes few background activities that cause contention. This explanation is also consistent with the higher variability observed for the uncached configuration of the read latency benchmark for Linux FAT32 as compared to MSFS, shown in Figure 4.9 (a).

For the cached configuration of the benchmark, median read throughput for MSFS and Linux FAT32 differ by less than 4%—3560 $\mathrm{MB/s}$ median throughput for MSFS versus 3430 $\mathrm{MB/s}$ for Linux FAT32—as shown in Figure 4.10 (b). We suspect that the filesystems here are constrained only by the ability of the α-Nexus and Linux system call mechanisms to perform high throughput data transfers between the kernel and the benchmark client.[20]

Mmap-oriented file access. A filesystem client can access the contents of a file using the mmap-oriented interface, rather than the streams-oriented interface we have used thus far. With the mmap-oriented interface, a client first creates a virtual memory mapping, then directly accesses the data one or more times. So performance here depends on access patterns. A client that accesses the same data multiple times need only create the virtual memory mappings once, effectively amortizing the performance cost of this oper-

[20]Data is read from the filesystem using 4 KB transfers. An upper bound on cached read throughput for this message size can be obtained by considering the system call throughput micro-benchmark, discussed in Section 4.5.2 and shown in Figure 4.7 (a). In that micro-benchmark, α-Nexus and Linux both achieve a median throughput of about 6000 $\mathrm{MB/s}$ for 4 KB transfers. That result is higher than achieved for the cached configuration of the read throughput benchmark, but it does not include the cost of copying data from the filesystem block cache into a process's address space. Instead, it only measures the performance for the kernel to zero-fill 4 KB message buffers in response to system calls.



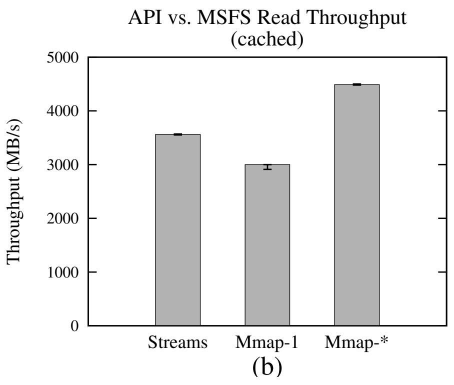


Figure 4.11: Cached read latency (a) and throughput (b) as measured by three filesystem benchmark clients. The Streams client uses a streams-oriented interface, as in previous experiments. The Mmap-1 client uses a mmap-oriented interfaces and includes the cost of creating a virtual memory mapping on every access. And the Mmap-* client uses a mmap-oriented interface but creates a single virtual memory mapping for use by all accesses. Each box shows the median latency achieved for one experiment, and error bars show the range in which the middle 80% of results fall.

ation across multiple accesses. In addition to the default Streams client, which we have used for running filesystem benchmark experiments, we created two new benchmark clients, called Mmap-1 and Mmap-*, each derived from the Streams client. Mmap-1 creates a virtual memory mapping and directly accesses file contents during each trial of the benchmark. Mmap-* creates a virtual memory mapping only once, during the first of 100 trials, then for each subsequent trial measures only the performance cost of directly accessing file contents. So Mmap-* models a filesystem client for which the cost of creating a virtual memory mapping is amortized over many data accesses.

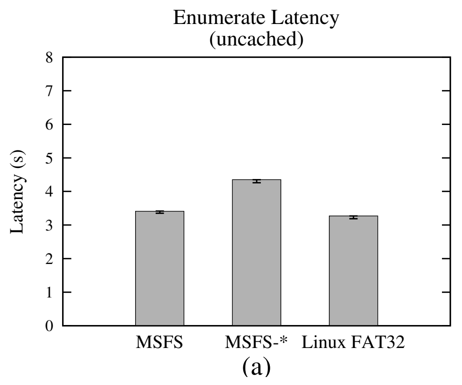
Figure 4.11 shows the results of running the read latency and read throughput benchmarks on MSFS for the three clients, Streams, Mmap-1 and Mmap-*. We show results only for the cached configuration of the benchmark, so disk performance has no influence. Figure 4.11 (a) shows that Mmap-1 has the worst median cached read latency of the three clients, 1.09 µs, versus 0.996 µs for Streams and nearly zero for Mmap-*. And Figure 4.11 (b) shows that Mmap-1 also has the worst median cached read throughput, 3000 MB/s, versus 3560 MB/s for Streams and 4490 MB/s for Mmap-*. So the cost of creating virtual memory mappings in each trial offsets any possible efficiency gains of subsequent direct accesses to file contents. By contrast, Mmap-*, which amortizes the costs of creating virtual memory mappings, achieves the lowest latency and highest throughput among the three clients. In fact, cached read latency for Mmap-* becomes too small to measure with our current benchmark infrastructure. In the best case, we expect read latency to be a few cycles (≈ 1 ns for our test platform).

The trade-offs between streams-oriented and mmap-oriented interfaces are not specific to MSFS or to α-Nexus. Linux FAT32 supports both interfaces, and we measured its performance using the modified filesystem benchmarks. We omit Linux FAT32 results because we observed only minor differences compared to the results in Figure 4.11 obtained for MSFS.

Enumerate Latency

When implementing MSFS components that access meta-data in the block cache, we had to chose either the streams-oriented interface or the mmap-oriented interface. For the MSFS FAT32 filesystem driver, we chose to use the streams-oriented interface, because we didn't expect this component to access any single piece of meta-data frequently enough to justify the cost of creating virtual memory mappings.

Each trial of the enumerate latency benchmark causes filesystem components to perform many access to meta-data—over one thousand directories are enumerated, and each directory operation requires access to meta-data stored in the block cache. So we implemented a version of MSFS, called MSFS-* to quantify the cost of this design decision. In MSFS-*, the FAT32 filesystem driver uses the mmap-oriented interface to the block cache. The cost of creating a virtual memory mapping for each piece of meta-data is amortized over all accesses to that meta-data. For the uncached benchmark configuration, amortization is only across a single trial of the benchmark, since caches are flushed between trials, while for the cached benchmark configuration, amortization is across all trials. Fig-



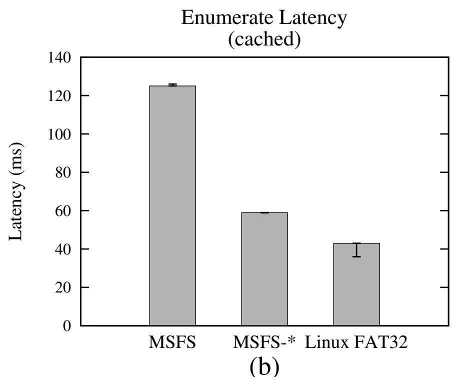


Figure 4.12: Enumerate latency benchmark performance of MSFS, MSFS-*, and Linux FAT32 for the uncached (a) and cached (b) benchmark configurations, measured in seconds and milliseconds (ms), respectively. MSFS-* is a modified version of MSFS that uses the mmap-oriented interface internally when accessing meta-data in the block cache. Each box shows the median latency achieved for one experiment, and error bars show the range in which the middle 80% of results fall.

ure 4.12 shows results for the enumerate latency benchmark for MSFS, MSFS-*, and Linux FAT32.

Uncached enumerate latency. For the uncached configuration of the enumerate latency benchmark, Figure 4.12 (a) shows that the median latency for MSFS and Linux FAT32 differ by 0.14 s, or about 4%, with 3.41 s for MSFS and 3.27 s for Linux FAT32. Here, as for other uncached filesystem benchmarks, performance is likely dominated by the cost of accessing the disk. The higher median latency for MSFS-* (4.35 s, as compared to 3.41 s for MSFS), indicates that there is insufficient spacial locality in access to meta-data during a single benchmark trial. Thus MSFS-* incurs the overhead of creating virtual memory mappings to access meta-data in the block cache, but MSFS-* can amortize this overhead across only a few meta-data accesses per block.

Cached enumerate latency. MSFS exhibits approximately $3.5x$ worse performance compared to Linux FAT32 for the cached configuration of the enumerate latency benchmark, as shown in Figure 4.12 (b), with MSFS achieving a median latency of 125 ms versus a median latency of 43 ms for Linux FAT32. However, the median latency for MSFS-* is 50 ms for the same benchmark, also from Figure 4.12 (b). Here, MSFS-* amortizes the cost of creating virtual mappings over all 100 benchmark trials, so the overhead for MSFS-* to access meta-data in the block cache is similar to direct access. By contrast, MSFS accesses meta-data using system calls to the block cache. It is not surprising then that the cached enumerate latency benchmark performance for MSFS is worse than for MSFS-*: 125 ms median latency for MSFS versus 59 ms for MSFS-*. And the median latency for MSFS-* (59 ms) is much closer to the median latency for Linux FAT32 (43 ms) than for MSFS (125 ms). This suggests that the cost of isolation, in the form of an increased cost for the FAT32 filesystem driver to access meta-data versus direct access, is the largest source of performance differences between MSFS and Linux FAT32 in this benchmark. Other reasons for a difference in performance include the performance costs for communication between isolated MSFS components, the overhead for guards on the channels between MSFS components, and costs incurred for DAC enforcement when accessing meta-data. None of these costs are incurred by Linux FAT32.

In hindsight, using the mmap-oriented interface within the FAT32 filesystem driver component to access meta-data, as done for MSFS-*, might have been a better implementation choice for MSFS. However, we can't reach a firm conclusion without a realistic model for client file and directory access patterns.

Merging Clients and VFS Components

Thus far, the filesystem benchmarks use MSFS implementations in which each VFS component, VFS_P, executes within the address space of the corresponding client P, instead of as an isolated process, and P invokes VFS_P using local function calls rather than IPC. In Section 4.4.8, we justified this optimization by arguing that the performance cost of IPC outweighs the benefits to trustworthiness that stem from isolation between P and VFS_P. Here we revisit that.

We implemented a version of MSFS without this optimization, and we then used the four filesystem benchmarks—open/close latency, read latency, read throughput, and enumerate latency—to compare the performance of the optimized and unoptimized versions. (For brevity, we omit graphs for these benchmarks and only summarize the results here.)

For the uncached configuration of the filesystem benchmarks, we observed only minor differences in performance between the optimized an unoptimized versions of MSFS. This was expected, since here, both versions of MSFS are constrained largely by disk performance, and the performance costs incurred for disk access far outweigh the overhead of IPC above local function calls.

For the cached configuration of some filesystem benchmarks, isolating VFS components caused a noticeable performance degradation. For the cached configuration of the open/close benchmark, we observed an increase of $10.2\ \mu\text{s}$ in the median latency achieved by the unoptimized MSFS as compared to the optimized MSFS. We attribute this increase to the overhead of IPC, because the increase is only slightly larger than twice the $5.02\ \mu\text{s}$ median α-Nexus IPC latency, from Figure 4.6 (b), and each trial for the unoptimized MSFS makes two additional IPC invocations—one IPC invocation from P to VFS_P for `open`, and a second for `close`—that were not present in the optimized implementation.

A similar analysis holds for the cached configuration of the enumerate latency benchmark, for which we also observed a performance decrease for the unoptimized MSFS versus the optimized MSFS.

For the cached configurations of the read latency and read throughput benchmarks, we expected the unoptimized MSFS to perform poorly, since each access to file contents now involves an IPC invocation from P to VFS_P versus a local function call in the optimized MSFS. Moreover, in the unoptimized MSFS, file data is now is transferred through both an IPC channel and system calls versus only system calls for the optimized MSFS. Results for the cached configurations of the read latency and read throughput benchmarks using the unoptimized MSFS implementation were $5.20\ \mu\text{s}$ and 530 MB/s, respectively, in the median case. This is substantially worse than the performance of the optimized MSFS for the same benchmarks (0.966 μs median latency, from Figure 4.9 (b), and 3560 MB/s median throughput, from Figure 4.10 (b)).

The poor file read performance for the unoptimized MSFS can be explained by considering the results from the micro-benchmarks discussed in Section 4.5.2. For the cached configuration of the read latency benchmark, the median latency (5.20 μs) is only 4% worse than the 5.02 μs median α-Nexus IPC latency (from Figure 4.6 (b)). Similarly, for the cached configuration of the read throughput benchmark, the median throughput (530 MB/s) is about 15% worse than the 612 MB/s median α-Nexus IPC throughput (from Figure 4.7 (b)) for 4 KB messages.

We have no simple metric to weigh the benefits of isolation for VFS_P and P against the degraded performance we observed for the unoptimized implementation. But the magnitude of the performance degradation suggests the optimization is worthwhile for common use cases.

4.6 Discussion

4.6.1 Secure Filesystems

We are not the first to consider the design and implementation of a trustworthy filesystem. Halcrow [64] provides a comprehensive overview of secure filesystems for Linux. Wright et al. [151] and Riedel et al. [115] examine the performance and trustworthiness of secure filesystems, in both distributed and non-distributed settings, for a variety of operating systems. Here we discuss only a few secure filesystems that are particularly germane to our work. We organize the discussion according to the trust (or, conversely, suspicion) between filesystem components, local and remote disks or storage services, clients, and users.

Suspicion of local disks. Many secure filesystems are intended to protect against theft of locally installed disks. Here, the filesystem employs encryption to prevent a disk from leaking filesystem data to an attacker who gains possession (hence, control) of the disk. One approach to implementing such a cryptographic filesystem places the filesystem code entirely within the operating system kernel (e.g., [65, 93, 122, 152, 156]). So, all processes place full trust in the filesystem code, which includes disk device drivers and code for implementing encryption and decryption. The entire filesystem is in the TCB for every system security goal.

A more common approach moves encryption and decryption code out of the kernel by executing some or all of the filesystem as a process above the kernel (e.g., [23,61,108,134]). These filesystems can, in principle, instantiate Mutual Suspicion and Least Privilege by granting filesystem processes only enough privileges to service client requests and to perform encryption and decryption. The result would be that filesystem code—at least, the portion executing as a user-space process—can violate only those security goals that depend on that filesystem. Consequently some filesystem code need not be in all TCBs. The user-space filesystem process would not be able to violate the isolation of client processes, for instance, even if the filesystem were to become compromised. In practice, however, user-space filesystems often execute with full administrative privileges and are trusted by the kernel (hence, by all processes). Therefore they are in the TCB for every system security goal. Rather than increased trustworthiness, the motivation for moving filesystem code out of the kernel instead appears to be programming convenience, administrative convenience, and the desire to avoid accidental compromise of the kernel due to bugs in filesystem code.

Untrusted remote storage. Cryptography can also be useful when disks or other storage services are accessed remotely (e.g., [10,59,74,84,129,130,134,154]). These filesystems encrypt data so that full trust need not be placed in a remote storage service. Consequently, the remote storage service holds few privileges beyond the ability to delete data or otherwise deny service. Filesystems that access data on a remote storage service do not rely on local disk device drivers, and in some cases (e.g., [10,154]) the filesystem is an application or library, rather than a system service. This makes Least Privilege easier to instantiate—an ordinary user can create and configure the filesystem without the cooperation of system administrators, and the resulting filesystems hold no more privileges than the user that created it. Eliminating disk device drivers could also lead to smaller TCBs.

Suspicion between filesystems and remote storage services. A remote storage service provides an opportunity for both the local filesystem and the remote storage service to be suspicious of each other. Saksha [75] instantiates Mutual Suspicion in an effort to ensure proper resource accounting and billing. Using signed transaction logs and other cryptographic techniques, Saksha ensures that neither the local filesystem nor the remote storage service holds privileges that can be used to violate the system's security goals.

Suspicion between users. Using cryptography as an isolation mechanism impedes the ability to share data. For instance, CFS [23] uses symmetric per-user encryption keys within the filesystem. Such keys must not be revealed to processes executing on behalf of other users, since any user that gains possession of another user's key can subsequently access any of that other user's data. This limits sharing files among users, unless the users place full trust in each other. Sharing files is also difficult if the filesystem places full trust in filesystem clients. pStore [10], for example, can't easily share files between users, because a substantial part of pStore executes as a library within each client's address space. The same problem arises in TrustedDB [9], which executes the client and filesystem together as a single component on a micro-kernel or hypervisor, without any isolation between client and filesystem.

Untrusted clients. In most filesystems, each client is a process that executes on behalf of some user, and a user is assumed to place full trust in such a client. Alcatraz [87] addresses the threat posed by filesystem clients that are not trustworthy, intercepting and suppressing any write request to the filesystem that was issued by an untrusted client. Solitude [72] queues write requests from a target client so that an administrator can first examine the requests and then, if desired, grant privileges to the process retroactively. These filesystems eliminate privileges that would normally be held by filesystem clients. Neither filesystem relies on encryption. VPFS [144, 145], by contrast, is a cryptographic filesystem that protects data stored by a single, trusted client from interference by all other clients. VPFS justifies this approach on Least Privilege grounds: no VPFS client holds privileges that can be used to access data stored by a different filesystem client.

Authorization and Authentication. Secure filesystems vary substantially in their support for authorization and authentication. Much work is this area is driven by the challenges of distributed authorization spanning multiple administrative domains. Closest to our work is a filesystem by Garg and Pfenning [56], called PCFS. That filesystem uses proof-carrying authorization [6] to support a wide variety of authorization policies. Clients provide proofs and credentials to the filesystem guard, which enforces authorization policies specified in a formal logic. PCFS includes mechanisms for automated proof construction, credential and proof caching, and revocation. MSFS guards rely on the α-Nexus alias table abstraction to achieve some of the same benefits as PCFS, though MSFS lacks a general proof search procedure that can be used by filesystem clients. Miltchev et al. [95] survey a large number of other secure distributed filesystems, both production systems and experimental prototypes, paying particular attention to the authentication and authorization mechanisms these filesystem support.

4.6.2 Security Principles

We based the design of MSFS on an interpretation of these security principles in the context of a system comprising many isolated components, each treated as a principal that acts independently of other components, but where some components can become compromised. This is not the only possible interpretation, and there is even debate about their usefulness in practice.

All of the security principles we use date to the 1970s, yet software appears to be no more secure today than it was then. This criticism of the principles is discussed, and largely dismissed, by Smith and Marchesini [127], who also discuss other concerns:

- The security principles are vague, easily misinterpreted, and likely impossible to fully achieve in practice.
- The security principles were developed in a very different context—commercial multi-user operating systems, in the case of Saltzer and Schroeder, and military and defense settings in which secrecy is (or was) paramount, in the case of Nibaldi. Thus one might worry that the principles address the wrong problems, present the wrong solutions, and address the wrong threats for today's pervasive, Internet-enabled, personal computing platforms.
- The security principles fail to address political, economic, and commercial realities. As a result, there are significant incentives for system designers to ignore these security principles, and few incentives to follow them.

These points have been raised by other authors as well. For instance, Viega and McGraw [141] concede that it is easier and likely more profitable to ignore Least Privilege than it is to instantiate Least Privilege with any degree of completeness.

Mutual Suspicion. Though it was formulated first, Mutual Suspicion is the least frequently cited of the three security principles we discuss here, and it is also seemingly the least controversial. The original formulation of Mutual Suspicion by Schroeder [124] was descriptive rather than prescriptive. Our interpretation of the principle is closer to Nelson et al. [101], who argue that each component of a system should be responsible for protecting itself by limiting reliance on information from external sources. Subsequent to that work, Woo and Lam [149] apply Mutual Suspicion in a similar way to users in an authentication system.

Minimization of Trusted Computing Bases. TCBs are often discussed in terms of a single set of system-wide security goals and a corresponding single TCB for the system. Rushby [118], for example, examines how to minimize the TCB for embedded systems that support only a single application. A general system, even one with a single user or application, will have many security goals and, therefore, it could have many different TCBs. Trade-offs arise when, in the course of reducing the size of one TCB (e.g., by moving functionality out of one component and into another), we inadvertently enlarge the size of some other TCB.

Bernstein [20] advocates for Minimization of TCBs as the key to building secure systems, and validates the benefits of this security principle by designing and implementing Qmail, a mail transfer agent. Qmail, which is now widely deployed, was designed with trustworthiness as an explicit goal, and it appears to have largely achieved that goal. It has an internal architecture similar to MSFS: functionality is decomposed across numerous fine-grained components, components are mutually suspicious of each other and hold as few privileges as possible, and guards are located on the communications channels between components.

Arbaugh et al. [7] argue that insufficient attention is paid to the lower layers of a system—hardware, firmware, BIOS, and bootstrapping code—when defining a TCB. Arbaugh et al. insist that the precise semantics of these lower layers should be fully understood and their implementations vetted, before they are included in a TCB. More recently, Smith and Marchesini [127] reiterate this point, but also argue that these lower layers, particularly hardware components, are increasingly too complex and too poorly understood to make a suitable foundation for building a secure system.

If a TCB is interpreted as being dynamic, rather than static, then the necessity of including all lower layers in a TCB can be avoided. Flicker [92], for instance, removes the operating system kernel from the TCB for certain application-level security goals, even though most of the application runs on top of the kernel. This is accomplished using new hardware mechanisms that support a dynamic root of trust [4, 100]: the kernel is temporarily suspended, and a small application-specified set of code is loaded into a secure, attested environment. The small piece of code thus runs directly on the lower-layer hardware and need not place full trust in the kernel.

Least Privilege. Least Privilege can be instantiated for users and the privileges they hold. Motiee et al. [96] provide empirical evidence that Least Privilege is rarely followed for Windows users. The authors of that study fault the authorization and authentication mechanisms supported by Windows for making Least Privilege impractical.

Interpreted broadly, Least Privilege applies not just to users, but to other principals as well. Much prior work concerns the application of Least Privilege to processes. Applying Least Privilege to a process typically means relying on `setuid` and similar system calls to change a process's privileges to some set different than what is held by the user that invoked the process. Tsafrir et al. [138] examine the poorly understood and poorly defined semantics of such system calls and the difficulties this raises for Least Privilege. Krohn et al. [77] investigate other ways in which common operating systems stand in the way of Least Privilege for processes.

Bernstein [20] argues that Least Privilege—at least, its most common interpretation—is "fundamentally wrong." Here, instantiating Least Privileges is interpreted to involve two steps. First, identify the components of the system and enumerate all privileges those components hold. Second, successively remove privileges from components as long as the system still functions properly. The resulting assignment of privileges to components might then be said to instantiate Least Privilege, because each component holds only privileges that are necessary for the system to function properly. In a compelling example of a system designer being misled by Least Privilege, Bernstein discusses the case of a DNS resolver that, rather than being decomposed into several mutually suspicious components that each hold only a few privileges, was simply "Least Privilege-ized" as a single component, to little effect.

One might instead take Least Privilege as a mandate to decompose a system into fine-grained components such that security-critical privileges are held by few of the resulting components. Here, the difficulty is in choosing the appropriate decomposition and arranging for the now-isolated components to communicate with each other. There have been several attempts to simplify and automate these tasks. Programmers can rely on a library [76] to make programming across isolation boundaries more convenient. Monolithic applications can also be decomposed into components automatically, based on program annotations [30, 114, 157]. Here, Least Privilege is taken to mean that the system should place full trust only in certain components, called *privileged* components, and that other, *unprivileged* components should hold few or no security-critical privileges. Thus, the automated approaches above decompose an application into two components—one privileged and one unprivileged—each executing as a processes on top of the kernel and communicating over an IPC channel. Swift [33] provides automatic decomposition for a more general case of Least Privilege, splitting an application across two or more mutually suspicious machines. Buyens et al. [31] discuss a variety of automated and manual program-restructuring techniques for instantiating Least Privilege. These techniques include splitting large components within an application into finer-grained components and splitting coarse-grained privileges into finer-grained privileges.

Finally, Smith and Marchesini [127] argue that the model of subjects, objects, and privileges, upon which both Mutual Suspicion and Least Privilege rest, may no longer be adequate: is a Web page an inactive object that is acted upon, or is it an active subject that makes requests and may hold privileges?

CHAPTER 5

CONCLUDING REMARKS

This dissertation describes NAL, a logic for specifying credentials and authorization policies. Instead of designing NAL from scratch, we started with an existing bare-bones authorization logic (CDD) that abstracts the essence of such logics, and we instantiated its notion of principals and its underlying predicate logic. Then, by building a suite of document-viewer applications and a filesystem, we demonstrated that NAL, despite its simplicity, is expressive and convenient enough to be a practical basis for implementing authorization in real systems.

NAL provided a vehicle for us to understand and bridge the gap between what authorization logics provide and what real systems need. The implementation of credentials and of principal names is one area where such a gap often exists. There are, for example, significant practical differences between credentials implemented by digital signatures, by hashes, and by ordinary messages on authenticated channels. These differences include the cost to create and validate credentials, whether secrets must be stored or shared, whether certain memory must be accessible to the credential holders, and whether the credential can be forwarded. Only in contemplating authorization for real applications, did these differences become apparent and did the design-trade-offs they enable become clear.

An imperative in the design of NAL and in our approach to supporting authorization was to empower system designers with flexibility for defining policies and implementing guards. This caused us to resist adding to NAL special-purpose constructs that shape policy by directly supporting revocation of credentials or by enforcing bounds on credential usage. Such constructs are only one way to create an authorization logic that is inherently non-temporal for use in a setting, like a computer system, where principals' beliefs actually do change over time. A system designer—informed by the semantics of an application—should know the best means for handling changes in principals' beliefs and, therefore, should be given the flexibility to implement that means. The state predicates that can appear in NAL formulas provide that flexibility, because changes in principals' beliefs are necessarily correlated with changes in system state.

NAL's flexibility also led us to take a very different view about the role of guards (or reference monitors) in systems. We propose in this dissertation that guards be seen as checking requester trustworthiness, where the authorization policy enforced by a guard defines criteria by which requester trustworthiness is evaluated. Identity-based and reputation-based authorization illustrate an axiomatic basis for deciding whether a requester is trustworthy, and this basis is widely used in practice. We argued in this dissertation, however, that analytic and synthetic bases are also worth supporting; and our document-viewer applications illustrated the power and convenience of these bases.

Our thesis that authorization be viewed not as an end to itself but rather as a proxy trustworthiness test is not limited to NAL or even to authorization logics. And there is reason to believe that guards that assess trustworthiness can implement a wider class of authorization policies than the more familiar "filtering" approach to building a guard, in which the guard is seen as only examining a series of requests and filtering out those requests that fail to satisfy certain checks. Clarkson and Schneider [38], for example, discuss security goals (i.e., hyperproperties) whose enforcement would require a guard to reason about the set of requests that a principal will not (or cannot) make. A guard that relies on analytic or synthetic bases for predicting trustworthiness can do just that.

Our MSFS filesystem, the most complex of the systems we implemented for this dissertation, makes extensive use of NAL for authorization between filesystem components, but primarily relies on an axiomatic basis for this authorization. One exception is disk driver code, which is authorized to perform I/O by virtue of the presence of a reference monitor (i.e., the DDRM) for that code, a synthetic basis for authorization. Some work has been done to relocate the synthesis of trustworthy disk driver code to compile time, by inlining the reference monitor. And in principle, compile-time analysis of that code can eliminate the need to add run-time checks. Here, α-Nexus's lack of support for high-level languages, for which program analysis is more tractable, is a limiting factor for using an analytic basis for authorization. Higher-level languages would also allow for finer-grained principals than are currently possible given the types of isolation boundaries provided by α-Nexus.

There are aspects of NAL that we did not fully explore. Restricted delegation, in particular, is used only occasionally in our document-viewer applications; MSFS replaces most uses of restricted delegation with a construction based on NAL groups. Axiomatizing restricted delegation can be subtle, an issue explored in some depth by Howell [68].

Although MSFS and our document-viewer applications enforce novel authorization policies, they rely on a fairly limited range of credentials to discharge those policies. Moreover, trust relationships between principals in these applications tend to be shallow: the reader of a document in $\mathit{ConfDocs}$ places trust in a guard implemented by $\mathit{ConfDocs}$, which in turn places trust in an analysis engine and certain hardware (e.g., a TPM)—but the analysis engine and hardware are trusted axiomatically. A richer instantiation of our approach to authorization would be more complex. An analysis engine for $\mathit{TruDocs}$ might be trusted, for example, because it was written in a way that is easily subjected to formal and even mechanical analysis. It is an open question whether NAL is expressive and convenient enough to specify authorization when principals rely on rich trust relationships. It is also unclear whether the benefits of such webs of trust outweigh the complexity they bring.

NAL does not specify a language for terms and predicates, and we have adopted various different language elements (e.g., lists, sets, numbers and arithmetic, and binary and text data) as convenient for implementing applications. This ad-hoc approach leaves unresolved several issues. Systems that uses a logic for authorization require careful attention to predicate and function naming, as noted in Section 2.2. α-Nexus and the applications described in this dissertation were all written by the same small group of programmers, so we had an easy time controlling names for predicates and terms used in policies and credentials. Even so, it has proved challenging to ensure names used in one application do not conflict with names used by other applications or the kernel.

Inconsistent use of predicate and function names is just one way in which a principal P can inadvertently issue contradictory credentials, i.e., credentials that allow P says false to be derived. Treating a group as a principal, as NAL does, admits contradictions from pooling credentials issued by multiple principals. And NAL's inference rules admit *ex falso quodlibet* (literally, "from falsehood, whatever you please"). P says $\mathcal{F}$ holds for any formula $\mathcal{F}$ whenever P issues a set of contradictory credentials.

From one perspective, adopting *ex falso quodlibet* is beneficial, because it exposes latent errors in the principals that issue credentials. But perhaps the machines, processes, and other principals we study in this dissertation are not (yet) ready for such a high standard, and an authorization mechanism should instead seek to minimize the consequences of contradiction. For instance, while system designers might rightly adopt *ex falso quodlibet* when analyzing the potential consequences of some set of credentials, guard implementations might instead use a paraconsistent logic [42]—unlike NAL, such logics do not admit *ex falso quodlibet*—so that contradictory credentials cannot be used in proofs that authorize requests. It is not clear, however, if such concerns outweigh the appeal of using a single logic as the basis both for design and for implementation.

The current NAL proof checker is hand-coded in C, so we have little assurance in its correctness. The run-time cost of executing the proof checker and the costs to store and transmit proofs were a concern, so we paid some attention to efficiency in the proof checker implementation and in the representation of proofs within the implementation. There is much prior work on efficient proof representation and validation (e.g., [43,94,98, 99]). Our proof checker employs only a few of these existing techniques; our reliance on C hinders further optimization. Initial work has been completed implementing a NAL proof checker in Coq [37]; that proof checker would certainly provide greater assurance than the current proof checker, and Coq may provide opportunities for more efficient proof representation as well.

A proof checker implemented in C has the advantage that application programmers—as opposed to logicians—are likely to be familiar with that programming language. Application programmers are unlikely to be familiar with the intricacies of writing proofs using Coq. Alpaca [81], by contrast, strives to make the construction of proofs simple for application programmers by providing a library with an intermediate programming-like environment for generating lemmas and proofs. That proof checker and library is implemented in Python, a language that is both familiar to application programmers and higher level than C.

One difficulty we encountered in writing NAL proofs involves a trade-off between shared context and the level at which proofs are written. Proofs can be easy to write and understand if clients and the proof checker share application-specific context, such as lemmas or proof fragments useful for constructing proofs for typical requests, or application-specific proof search strategies. An application-specific context can also help reduce the size of proofs. Credentials and cryptographic keys used in proofs, for example, are quite large. But if a proof checker maintains a database of credentials for use in proofs or of cryptographic keys used for credentials, then proofs sent to that proof checker would not need to include this information. But maintaining such application-specific context makes a proof checker more complex, and it requires coordination between clients and proof checkers. These considerations led us to share very little context between proof checkers and applications. But, proofs for our applications are consequently difficult to understand and difficult to generate. The proofs, however, are mostly self-contained, since they rely only on axioms and inference rules shared by all proof checkers.

We have noted how credentials-based authorization can aid in the creation of a comprehensive audit facility. Our applications create audit logs, but the simplistic approach they use to do so—writing the complete proof for each access to a file, for example—is inadequate. Such audit logs reveal both too much and too little information. In *ConfDocs*, the audit log reveals details about confidential documents and must, therefore, be protected from unauthorized access. More generally, since credentials convey attributes of principals, any approach to authorization that makes decisions based on credentials could impinge on the privacy of individuals.[1] However, guards in MSFS create audit logs with very little information that could impinge on the privacy of individuals. This is because MSFS relies on two levels of credentials-based authorization for every request:

- a first level, in which the kernel authorizes a process's request to send a message on behalf of a given principal name;
- and a second level, in which a guard in MSFS checks an access control list to determine if that principal is authorized to perform the requested operation.

This two-level scheme is beneficial for privacy, enabling principals to make requests or issue credentials that reveal only enough information for some constrained use, but it also hinders the ability to audit, because the credentials (and proofs) are not all located in one place. The same difficulties for audit arise whenever authorization for one operation depends on authorization of operations elsewhere in the system.

In conclusion, this dissertation continues a line of inquiry that began when the first authorization mechanisms were developed for time-shared and multi-programmed computer systems, nearly fifty years ago. Since then, authorization has been recognized as one of the key enablers for trustworthy systems, and numerous mechanisms have been proposed to address new and existing challenges. This dissertation proposes that authorization not be viewed as primarily concerning a set of mechanisms, but that it be viewed in terms of evaluating trustworthiness of principals. Our experience using NAL and credentials-based authorization gives reason to hope that taking this view is progress for better security as computer systems become more pervasive and interconnected.

[1]We define *privacy* to be the right of an individual to control the dissemination and use of information about that individual.

APPENDIX A

NAL INFERENCE RULES

NAL's axiomatization is similar to CDD [1,2], augmented with rules for sub-principals and groups, and with standard rules for quantification in a predicate calculus [136, 140]. The SAYS-I rule of NAL is called UNIT in CDD; Abadi shows that CDD's BINDM axiom is equivalent to NAL's SAYS-E (also known as C4) in the presence of SAYS-I and DEDUCE (both of which are present in NAL). We assume, but do not show in the axiomatization, rules for variable renaming and substitution.

The derivation of any NAL formula $\mathcal{F}$ can be represented as a proof tree whose nodes correspond to NAL formulas.

- Leaves correspond to axioms and assumptions. Each assumption has a unique label Λ_i.
- Each internal node in the tree corresponds to the conclusion $\mathcal{G}$ of some NAL inference rule. The formulas that correspond to the node's immediate predecessors are the premises needed to conclude $\mathcal{G}$ using the rule.
- The root of the tree corresponds to $\mathcal{F}$.

Rules FORALL-I and EXISTS-E below involve side conditions that refer to "uncanceled assumptions". In a proof tree that derives $\mathcal{F}$, an assumption $\mathcal{A}$ with label Λ_i is defined to be *canceled* if and only if any path from the node that corresponds to $\mathcal{F}$ to the node that corresponds to $\mathcal{A}$ passes through a node derived by applying inference rule IMP-I(Λ_i); otherwise $\mathcal{A}$ is considered *uncanceled*.

A.1 NAL Rules from Constructive Predicate Logic

$$\textsc{true}: \frac{}{\mathsf{true}} \qquad \textsc{imp-e}: \frac{\mathcal{F}\ ,\ \mathcal{F} \Rightarrow \mathcal{G}}{\mathcal{G}} \qquad \textsc{imp-i}(\Lambda_i): \frac{\begin{array}{c} \Lambda_i: \overline{\ \mathcal{F}\ } \\ \hline \vdots \\ \hline \mathcal{G} \end{array}}{\mathcal{F} \Rightarrow \mathcal{G}}$$

$$\textsc{and-i}: \frac{\mathcal{F}\ ,\ \mathcal{G}}{\mathcal{F} \wedge \mathcal{G}} \qquad \textsc{and-left-e}: \frac{\mathcal{F} \wedge \mathcal{G}}{\mathcal{F}} \qquad \textsc{and-right-e}: \frac{\mathcal{F} \wedge \mathcal{G}}{\mathcal{G}}$$

$$\textsc{or-left-i}: \frac{\mathcal{F}}{\mathcal{F} \vee \mathcal{G}} \qquad \textsc{or-right-i}: \frac{\mathcal{G}}{\mathcal{F} \vee \mathcal{G}} \qquad \textsc{or-e}: \frac{\mathcal{F} \Rightarrow \mathcal{H}\ ,\ \mathcal{G} \Rightarrow \mathcal{H}\ ,\ \mathcal{F} \vee \mathcal{G}}{\mathcal{H}}$$

$$\textsc{forall-i}: \frac{\mathcal{F}}{(\forall \mathsf{v} : \mathcal{F})} \quad \text{v is not free in any uncanceled assumptions in the derivation of } \mathcal{F}$$

$$\textsc{forall-e}: \frac{(\forall \mathsf{v} : \mathcal{F})}{\mathcal{F}[\mathsf{v} := \tau]} \quad \text{free variables of term } \tau \text{ are free for v in } \mathcal{F}$$

$$\textsc{prop-forall-e}: \frac{(\forall \mathsf{x} : \mathcal{F})}{\mathcal{F}[\mathsf{x} := \mathcal{G}]} \quad \text{free variables of formula } \mathcal{G} \text{ are free for x in } \mathcal{F}$$

$$\textsc{exists-i}: \frac{\mathcal{F}[\mathsf{v} := \tau]}{(\exists \mathsf{v} : \mathcal{F})} \quad \text{free variables of term } \tau \text{ are free for v in } \mathcal{F}$$

$$\text{PROP-EXISTS-I:}\ \frac{\mathcal{F}[\mathsf{x} := \mathcal{G}]}{(\exists \mathsf{x} : \mathcal{F})} \qquad \text{free variables of formula } \mathcal{G} \text{ are free for } \mathsf{x} \text{ in } \mathcal{F}$$

$$\text{EXISTS-E:}\ \frac{\mathcal{F} \Rightarrow \mathcal{G}\ ,\ (\exists \mathsf{v} : \mathcal{F})}{\mathcal{G}} \qquad \begin{array}{l}\mathsf{v} \text{ is not free in } \mathcal{G} \text{ or in any uncanceled assumptions} \\ \text{in the derivation of } \mathcal{F} \Rightarrow \mathcal{G}\end{array}$$

A.2 NAL Definitions and Rules Derived from CDD

false: $(\forall \mathsf{x} : \mathsf{x})$

$\neg\mathcal{F}$: $(\mathcal{F} \Rightarrow \mathsf{false})$

$A \rightarrow B$: $(\forall \mathsf{x} : (A\ \mathsf{says}\ \mathsf{x}) \Rightarrow (B\ \mathsf{says}\ \mathsf{x}))$

$A \xrightarrow{\overline{\mathsf{v}}:\mathcal{F}} B$: $(\forall \overline{\mathsf{v}} : (A\ \mathsf{says}\ \mathcal{F}) \Rightarrow (B\ \mathsf{says}\ \mathcal{F}))$ for $\overline{\mathsf{v}}$ not free in A or B

$$\text{DEDUCE:}\ \frac{A\ \mathsf{says}\ (\mathcal{F} \Rightarrow \mathcal{G})}{(A\ \mathsf{says}\ \mathcal{F}) \Rightarrow (A\ \mathsf{says}\ \mathcal{G})}$$

$$\text{SAYS-I:}\ \frac{\mathcal{F}}{A\ \mathsf{says}\ \mathcal{F}} \qquad\qquad \text{SAYS-E:}\ \frac{A\ \mathsf{says}\ (A\ \mathsf{says}\ \mathcal{F})}{A\ \mathsf{says}\ \mathcal{F}}$$

A.3 NAL Extensions

SUBPRIN: $$\frac{}{A \rightarrow A.\tau}$$

EQUIV SUBPRIN: $$\frac{\tau_1 = \tau_2}{A.\tau_1 \rightarrow A.\tau_2}$$

MEMBER: $$\frac{\mathcal{P}[\mathsf{v} := A]}{A \rightarrow \{\!|\mathsf{v} : \mathcal{P}|\!\}}$$ free variables of A are free for v in $\mathcal{P}$

$\rightarrow$ GROUP: $$\frac{(\forall \mathsf{v} : \mathcal{P} \Rightarrow (\mathsf{v} \rightarrow A))}{\{\!|\mathsf{v} : \mathcal{P}|\!\} \rightarrow A}$$ v is not free in A

A.4 NAL Derived Inference Rules

FALSE: $$\frac{\mathsf{false}}{\mathcal{F}}$$

$\rightarrow$ TRANS: $$\frac{A \rightarrow B\ ,\ B \rightarrow C}{A \rightarrow C}$$

$\xrightarrow{\bar{\mathsf{v}}:\mathcal{F}}$ TRANS: $$\frac{A \xrightarrow{\bar{\mathsf{v}}:\mathcal{F}} B\ ,\ B \xrightarrow{\bar{\mathsf{v}}:\mathcal{F}} C}{A \xrightarrow{\bar{\mathsf{v}}:\mathcal{F}} C}$$

HAND-OFF: $$\frac{B \text{ says } (A \rightarrow B)}{A \rightarrow B}$$

REST-HAND-OFF: $$\frac{B \text{ says } (A \xrightarrow{\bar{\mathsf{v}}:\mathcal{F}} B)}{A \xrightarrow{\bar{\mathsf{v}}:\mathcal{F}} B}$$

GROUP MONOTONICITY: $$\frac{(\forall \mathsf{v} : \mathcal{P} \Rightarrow \mathcal{P}')}{\{\!|\mathsf{v} : \mathcal{P}|\!\} \rightarrow \{\!|\mathsf{v} : \mathcal{P}'|\!\}}$$

APPENDIX B

NAL GUARD LIBRARY AND PROOF CHECKER

We implemented a NAL guard library in C.[1] Our goal was not to build a production-quality system but rather to ascertain whether a NAL guard—and in particular, the code for checking NAL proofs—could be fast enough to be on the critical path for accesses. However, we did not pay special attention to low-level performance optimizations.

Preliminary measurements suggest that our NAL proof checker achieved these performance goals. A typical authorization in *ConfDocs* requires proofs involving roughly 1000 inference rules, and our proof checker validates these proofs in approximately 300 milliseconds—fast enough so that a user of *ConfDocs* senses no delay. A significantly faster implementation would be required if the proof checker were invoked on the critical path for authorizing accesses to all α-Nexus kernel resources. However, the power of a general purpose proof checker is unlikely to be needed for that setting, because policies, credentials, and proofs there assume only a few simple forms. So special-purpose proof checkers, which can be considerably faster, would be deployed. In addition, the cost of invoking the NAL proof checker can be amortized over many authorization checks. The α-Nexus shared memory guard provides such an example; it relies on partially-checked proofs stored in the invoking process's alias table.

Proofs accepted by the NAL guard library are specified in two parts. The first part is a schema for obtaining and validating a set of credentials. Section 2.4.2 describes the types of credential specifications accepted by the guard library and how the guard library processes those credentials. We provide additional implementation details below. The second part of a proof is a proof schema, which acts as a program to transform a set of premises into a conclusion according to the axioms and inference rules of NAL. This appendix describes how proof schemas are represented in α-Nexus and how the guard

[1]C is the only high-level language α-Nexus supports, so we had little choice here. Admittedly, other languages would have been better suited to the task.

library's proof checker determines if a proof schema is well formed.

B.1 Formula Representations

NAL formulas and terms have various representations in the implementation.

- A text-based representation is used for display purposes and when necessary due to constraints on the objects in which formulas or terms are embedded. $\mathit{TruDocs}$ and $\mathit{ConfDocs}$, for example, use a text-based representation to embed formulas in text documents. It is the least compact encoding, and it is the most expensive to parse. We built the parser and lexical analyzer using GNU Flex [110] and Bison [128]. Unlike the other representations, we do not define a canonical form for the text-based representation. Thus, we cannot check if two formulas or terms in this text-based representation are identical.
- A serialized binary representation is used when a compact, machine-readable implementation is needed, such as when storing formulas or terms on an ACL, in a database on disk, in an IPC message, or in a parameter or return value for an α-Nexus system call. This representation is used by MSFS when storing principal names on disk, for instance. This representation is based on DER [71], a format widely used for cryptographic keys and X.509 digital certificates [67]. It is a more compact representation for formulas and terms than the text-based representation. It can be parsed relatively efficiently with a simple, special-purpose DER parser.
- An expanded representation using C data structures is used as an intermediate representation when manipulating formulas in memory. Because this representation is not flat, it is not suitable for sending over channels or embedding in objects. The expanded representation is convenient for manipulating programmatically, so it is used internally by the NAL guard library's proof checker and by most applications that use NAL.

B.2 Stack-Based Proof Checking

Proof schemas are trees, and they are conveyed to our proof checker using a postfix-like representation. The formula corresponding to the root of the tree is preceded by a list of inference rule names, axiom names, and meta-rules (which allow terser encodings for some constructions). We adopted this representation because it promised performance advantages over the more familiar Hilbert-style proof representation as a sequence of formulas, each accompanied by a justification (itself the name of an axiom or inference rule and list of previous line numbers that identify premises). In particular:

- Our postfix-like representation tends to be shorter than the Hilbert-style representation, because it does not contain intermediate formulas or references to lines in the proof. Our postfix-like representation only contains the formulas at the root and leaves of the tree, plus names (which are short) of the inference rules used to derive intermediate formulas.
- Our postfix-like representation allows proof-checking with a smaller footprint for its working memory. In the Hilbert-style representation, checking line i requires that lines 1 through $i-1$ be available in a convenient (i.e., expanded) form in working memory; only later lines can be stored in encoded form or on disk. In our postfix-like representation, the working memory need only contain a stack that stores premises for the next inference rule to be checked. Since each NAL inference rule involves only a few premises, this means the stack depth need only store a few elements (where each element stores an entire NAL formula in its expanded representation).

A NAL proof schema in our postfix-like representation specifies a series of reductions to be performed on *judgments*. Each judgment has the form of a *sequent*[2] $\Sigma \vdash \mathcal{F}$, where Σ is

[2]The NAL axiomatization in Appendix A does not use sequents; the proof checker does, because that is more convenient. There is a trivial translation from proofs in terms of the axiomatization of Appendix A into the sequent-based logic used by our NAL proof checker.

a set of formulas that represent uncanceled assumptions in the derivation of $\mathcal{F}$. Initially, there is an empty stack of judgments. Each line of the proof is read, then a reduction is initiated according to a rule selected based on that line. The available rules are stored in a *rule base*. Roughly speaking, a line corresponding to a rule requiring n premises is processed as follows. The proof checker:

(1) pops judgments $\Sigma_1 \vdash \mathcal{F}_1, \ldots, \Sigma_n \vdash \mathcal{F}_n$ from the stack,

(2) checks that $\mathcal{F}_1, \ldots, \mathcal{F}_n$ are suitable for use as premises for the rule,

(3) checks any side conditions for the rule,

(4) constructs the rule's conclusion $\mathcal{G}$ based on the premises $\mathcal{F}_1, \ldots, \mathcal{F}_n$, and

(5) pushes a judgment $\bigcup_{i=1}^{n} \Sigma_i \vdash \mathcal{G}$ on to the stack.

Once all input lines have been processed, the proof is deemed well-formed if the stack contains a single sequent $\emptyset \vdash \mathcal{F}$ where $\mathcal{F}$ matches the root of the proof tree being checked.

Our NAL proof checker has a built-in, hard-coded entry in the rule base for each NAL inference rule that involves checking a side condition or modifying a sequent. Each such rule can be characterized in terms of (i) the contents of the stack top when the rule is activated, and (ii) how executing the rule changes the stack. Built-in rules include the following.

- `assume` $\mathcal{F}$.

  Push $\{\mathcal{F}\} \vdash \mathcal{F}$.

- `imp-i` $\mathcal{F}$.

  Pop $\Sigma \vdash \mathcal{G}$ then push $\Sigma \setminus \{\mathcal{F}\} \vdash \mathcal{F} \Rightarrow \mathcal{G}$.

- `forall-i` v.

  Pop $\Sigma \vdash \mathcal{F}$ then push $\Sigma \vdash (\forall \mathsf{v} : \mathcal{F})$, provided v is not free in any formula of Σ.

- `forall-e` τ.

  Pop $\Sigma \vdash (\forall \mathsf{v} : \mathcal{F})$ then push $\Sigma \vdash \mathcal{F}[\mathsf{v} := \tau]$, provided free variables of term τ are free for v in $\mathcal{F}$.

- `prop-forall-e` $\mathcal{G}$.

  Pop $\Sigma \vdash (\forall \mathsf{x} : \mathcal{F})$ then push $\Sigma \vdash \mathcal{F}[\mathsf{x} := \mathcal{G}]$, provided free variables of formula $\mathcal{G}$ are free for x in $\mathcal{F}$.

- `exists-i` $(\exists \mathsf{v} : \mathcal{F})$.

  Pop $\Sigma \vdash \mathcal{F}[\mathsf{v} := \tau]$ then push $\Sigma \vdash (\exists \mathsf{v} : \mathcal{F})$, provided free variables of term τ are free for v in $\mathcal{F}$.

- `prop-exists-i` x.

  Pop $\Sigma \vdash \mathcal{F}[\mathsf{v} := \mathcal{G}]$ then push $\Sigma \vdash (\exists \mathsf{v} : \mathcal{F})$, provided free variables of formula $\mathcal{G}$ are free for v in $\mathcal{F}$.

- `exists-e`.

  Pop $\Sigma \vdash \mathcal{F} \Rightarrow \mathcal{G}$, and $\Sigma' \vdash (\exists \mathsf{v} : \mathcal{F})$, then push $\Sigma \cup \Sigma' \vdash \mathcal{G}$, provided v is not free in $\mathcal{G}$ or in any formula of Σ.

- `member` $\{\!\!\{\mathsf{v} : \mathcal{P}\}\!\!\}$.

  Pop $\Sigma \vdash \mathcal{P}[\mathsf{v} := A]$, then push $\Sigma \vdash A \rightarrow \{\!\!\{\mathsf{v} : \mathcal{P}\}\!\!\}$, provided free variables of A are free for v in $\mathcal{P}$.

- `group-sfor`.

  Pop $\Sigma \vdash (\forall \mathsf{v} : \mathcal{P} \Rightarrow (\mathsf{v} \rightarrow A))$, then push $\Sigma \vdash \{\!\!\{\mathsf{v} : \mathcal{P}\}\!\!\} \rightarrow A$, provided v is not free in A.

- `rename` $\mathcal{F}$.

  Pop $\Sigma \vdash \mathcal{G}$ then push $\Sigma \vdash \mathcal{F}$, if $\mathcal{F} =_\alpha \mathcal{G}$, where $(=_\alpha)$ denotes alpha equivalency .

All other NAL axioms and inference rules, because they do not involve checking side conditions or modifying sequents, are implemented using a routine that adds to the proof checker's rule base. This routine effectively extends the proof checker's logic by making new axioms, inference rules and derived inference rules available for use in a proof schema. As part of initialization for the NAL proof checker, this routine is invoked for each of the remaining NAL axioms and inference rules.

Finally, the NAL proof checker supports a few additional rules that manipulate the stack in simple ways. These do not correspond to NAL inference rules, and they do not derive any new formulas. They simply allow some proofs to be represented in a more compact form.

- `pushdown` n.

  Move the top element of the stack down n elements.

- `pullup` n.

  Move the n^{th} highest stack element up, to be at the top of the stack.

- `dup` n.

  Create n copies of the top element of the stack and push these duplicates on to the stack.

B.3 Implementation Details

The NAL proof checker is approximately 5000 lines of code, including about 2500 lines concerned with manipulating and parsing NAL formulas and converting between the various formula representations. The code for the proof checker can be instantiated in several ways:

Stand-alone service An α-Nexus process monitors a well-known IPC port for requests from clients. Three kinds of IPC requests are supported:

- `check`: Check if a given proof schema is well-formed.
- `add_derived_inference_rule`: Given a name and a proof schema for a derived inference rule, check whether the proof schema is well formed and, if so, add a new entry in the rule base for that derived inference rule. The proof checker maintains a separate rule base for each IPC client, and subsequent invocations

of the proof checker by the same IPC client can use the derived inference rule in a proof schema by using the given name. Proofs for other clients are unaffected.

- `add_inference_rule`: Add an arbitrary new axiom or inference rule to the rule base for that IPC client. The proof checker makes no effort to check soundness of the new rule, so care must be taken when making this kind of request—unsound axioms or inference rules might cause the proof checker to declare a proof schema well-formed even if it is not, hence a guard in that IPC client that invokes the proof checker may declare arbitrary requests to satisfy an authorization policy.

Library Routines The proof-checker can be compiled as a library and linked into a program. The entry points to the library are: `check`, `add_derived_inference_rule`, and `add_inference_rule`.

Interactive Command The proof checker can be invoked from the shell, e.g., by a human user. The command reads an input file containing zero or more inference rules (which are added to the rule base for that invocation of the proof checker), zero or more derived inference rules (which are checked and then added to the rule base), and one or more proof schemas (which are checked).

Guard Library The NAL guard library is a wrapper around the proof-checker library routines. The guard library interface allows an α-Nexus `guard` object to be created. Each `guard` object maintains a separate proof checker rule base and is configured with a NAL formula representing the authorization policy to be checked. A program processing a request invokes `guard_check`, which causes the `guard` object to

(i) invoke the proof checker on the provided proof schema,

(ii) validate that what is proved matches the authorization policy, and

(iii) check that each premise of the proof schema is backed by a valid credential.

A `guard` object includes helper routines for validating credentials (e.g., digital signature checks as well as methods to read certain accessible parts of the system state, time of day, etc.); this set of routines can be extended to handle new kinds of credentials.

B.4 External Theories

The axiomatization of NAL presented in Appendix A omits the various other theories that might be required by applications. In the interest of expedience, we implemented only the theories—portions of arithmetic and set theory—needed for reasoning about objects used by our applications and by MSFS.

The proof checker does not directly check proofs of claims about arithmetic or other objects that appear in beliefs. These are handled by invoking a set of pre-installed external proof checker routines.[3] The invocation occurs whenever, in the process of checking a proof schema, the NAL proof checker encounters a line

$$\texttt{extern}\ \mathit{proof\_checker\_name} : \ \mathit{string}$$

where $\mathit{proof\_checker\_name}$ is the name of some pre-installed external proof checker routine and string typically contains, explicitly or implicitly, a formula $\mathcal{F}$ (using a syntax known to that external proof checker routine) to be proved followed by a purported proof of $\mathcal{F}$. $\mathcal{F}$ is then returned by $\mathit{proof\_checker\_name}$ to the NAL proof checker if and only if the proof provided for $\mathcal{F}$ checks.

[3]An alternative is to use `add_inference_rule` and extend the NAL proof checker by adding inference rules for each new theory. We rejected this approach both because it is inconvenient and because it precludes using more efficient decision procedures and model checkers where they exist.

BIBLIOGRAPHY

[1] M. Abadi, "Access control in a core calculus of dependency," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 5–31, Apr. 2007.

[2] ——, "Variations in access control logic," in *Proceedings of the 9th International Conference on Deontic Logic in Computer Science*, ser. Lecture Notes in Computer Science, vol. 5076. Berlin, Germany: Springer, Jul. 2008, pp. 96–109.

[3] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," *ACM Programming Languages and Systems*, vol. 15, no. 4, pp. 706–734, Sep. 1993.

[4] Advanced Micro Devices, Inc., "AMD I/O virtualization technology (IOMMU) specification," http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf, Feb. 2009.

[5] J. P. Anderson, "Computer security technology planning study," ESD/AFSC, Hanscom AFB, Bedford, MA, Tech. Rep. ESD-TR-73-51, Vol. 2, Oct. 1972, NTIS AD758206.

[6] A. W. Appel and E. W. Felten, "Proof-carrying authentication," in *Proceedings of the 6th ACM Conference on Computer and Communications Security*. New York, NY: ACM Press, Nov. 1999, pp. 52–62.

[7] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 1997, pp. 65–71.

[8] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical domain and type enforcement for UNIX," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 1995, pp. 67–77.

[9] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware based database with privacy and data confidentiality," in *Proceedings of the 2011 ACM International Conference on Management of Data*. New York, NY: ACM Press, Jun. 2011, pp. 205–216.

[10] C. Batten, K. Barr, A. Saraf, and S. Trepetin, "pStore: A secure peer-to-peer backup system," Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. LCS 632, 2001.

[11] L. Bauer, "Access control for the Web via proof-carrying authorization," Ph.D. dissertation, Princeton University, Princeton, NJ, 2003.

[12] L. Bauer, L. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea, "A user study of policy creation in a flexible access-control system," in *Proceedings of the 26th ACM Conference on Human Factors in Computing Systems*. New York, NY: ACM Press, Apr. 2008, pp. 543–552.

[13] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar, "Device-enabled authorization in the Grey system," in *Proceedings of the 8th Information Security Conference*, ser. Lecture Notes in Computer Science, vol. 3650. Berlin, Germany: Springer, Jul. 2005, pp. 431–445.

[14] L. Bauer, S. Garriss, and M. K. Reiter, "Distributed proving in access-control systems," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 2005, pp. 81–95.

[15] M. Becker, C. Fournet, and A. Gordon, "Design and semantics of a decentralized authorization language," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*. Los Alamitos, CA: IEEE Computer Society, Jul. 2007, pp. 3–15.

[16] M. Y. Becker and S. Nanz, "A logic for state-modifying authorization policies," in *Proceedings of the 12th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, J. Biskup and J. Lopez, Eds., vol. 4734. Berlin, Germany: Springer, Sep. 2007, pp. 203–218.

[17] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management, applied to electronic health records," in *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. Los Alamitos, CA: IEEE Computer Society, Jun. 2004, pp. 139–154.

[18] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Bedford, MA, Tech. Rep. ESD-TR-73-278, Vol. 1, 1973.

[19] D. E. Bell, "Looking back at the Bell-La Padula model," in *Proceedings of the 21st Annual Computer Security Applications Conference*. Los Alamitos, CA: IEEE Computer Society, Dec. 2005, pp. 337–351.

[20] D. J. Bernstein, "Some thoughts on security after ten years of qmail 1.0," in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*. New York, NY: ACM Press, Nov. 2007, pp. 1–10.

[21] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety, and performance in the SPIN operating system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Dec. 1995, pp. 267–283.

[22] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Bedford, MA, Tech. Rep. MTR-3153, Jun. 1975, NTIS ADA039324.

[23] M. Blaze, "A cryptographic file system for Unix," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*. New York, NY: ACM Press, Nov. 2003, pp. 9–16.

[24] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The role of trust management in distributed systems security," in *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, 1st ed., ser. Lecture Notes in Computer Science *State-of-the-Art*, J. Vitek and C. D. Jensen, Eds. Berlin, Germany: Springer, Jul. 1999, vol. 1603, pp. 185–210.

[25] M. Blaze, J. Feigenbaum, and A. D. Keromytis, "KeyNote: Trust management for public-key infrastructures," in *Proceedings of the 6th Security Protocols Workshop*, ser. Lecture Notes in Computer Science, vol. 1550. Berlin, Germany: Springer, Apr. 1998, pp. 59–63.

[26] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 1996, pp. 164–173.

[27] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance checking in the PolicyMaker trust management system," in *Proceedings of the 2nd International Conference on Financial Cryptography*, ser. Lecture Notes in Computer Science, R. Hirschfeld, Ed., vol. 1465. Berlin, Germany: Springer, Feb. 1998, pp. 254–274.

[28] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter, "Consumable credentials in logic-based access-control systems," in *Proceedings of the 14th Annual Network and Distributed System Security Symposium*. Reston, VA: Internet Society, Feb. 2007, pp. 143–157.

[29] D. F. C. Brewer and M. J. Nash, "The Chinese wall security policy," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 1989, pp. 206–214.

[30] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th USENIX Security Symposium*. Berkeley, CA: USENIX Association, Aug. 2004, pp. 57–72.

[31] K. Buyens, B. de Win, and W. Joosen, "Identifying and resolving least privilege violations in software architectures," in *Proceedings of the 4th International Conference on Availability, Reliability and Security*. Los Alamitos, CA: IEEE Computer Society, Mar. 2009, pp. 232–239.

[32] Canon, Inc., "Image verification: Canon data verification system," http://cpn.canon-europe.com/content/education/infobank/image_verification/canon_data_verification_system.do.

[33] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure Web applications via automatic partitioning," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Oct. 2007, pp. 31–44.

[34] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating system errors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Oct. 2001, pp. 73–88.

[35] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust management for Web applications," *World Wide Web Journal*, vol. 2, no. 3, pp. 127–139, Summer 1997.

[36] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *Proceedings of the 1986 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 1987, pp. 184–193.

[37] M. R. Clarkson, Personal Communication, Oct. 2011.

[38] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010.

[39] Commission of the European Communities (ECSC, EEC, EAEC), "Information technology security evaluation criteria (ITSEC): Provisional harmonised criteria," Jun. 1991, document COM(90) 314, Version 1.2.

[40] "Common criteria for information technology security evaluation (CC)," http://www.commoncriteriaportal.org/, Sep. 2005.

[41] R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," in *Proceedings of the AFIPS Fall Joint Computer Conference, part I*, vol. 27. New York, NY: ACM Press, Nov. 1965, pp. 213–229.

[42] C. V. Damásio and L. M. Pereira, "A survey of paraconsistent semantics for logic programs," in *Handbook of Defeasible Reasoning and Uncertainty Management Systems*. Norwell, MA: Kluwer Academic Publishers, 1998, vol. 2, Reasoning with Actual and Potential Contradictions, pp. 241–320.

[43] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem," *Indagationes Mathematicae*, vol. 34, no. 5, pp. 381–392, 1972.

[44] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[45] J. B. Dennis and E. C. van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[46] Department of Defense, "Department of Defense trusted computer system evaluation criteria (TCSEC)," DoD 5200.28-STD, http://csrc.nist.gov/publications/history/dod85.pdf, Dec. 1985.

[47] J. DeTreville, "Binder, a logic-based security language," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 2002, pp. 105–113.

[48] DocBook, http://www.docbook.org/.

[49] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI certificate theory," Internet Engineering Task Force RFC 2693, 1999.

[50] Ú. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proceedings of the 1999 New Security Paradigms Workshop*. New York, NY: ACM Press, Sep. 1999, pp. 87–95.

[51] H. Farid, "Exposing digital forgeries in scientific images," in *Proceedings of the 8th Workshop on Multimedia and Security*. New York, NY: ACM Press, Sep. 2006, pp. 29–36.

[52] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," in *Proceedings of the 15th NIST-NCSC National Computer Security Conference*. Baltimore, MD: National Institute of Standards and Technology, Oct. 1992, pp. 554–563.

[53] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA: Addison Wesley, 1995.

[54] D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter, "A linear logic of authorization and knowledge," in *Proceedings of the 11th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, Eds., vol. 4189. Berlin, Germany: Springer, Sep. 2006, pp. 297–312.

[55] D. Garg and F. Pfenning, "Non-interference in constructive authorization logic," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. Los Alamitos, CA: IEEE Computer Society, Jul. 2006, pp. 283–296.

[56] ——, "A proof-carrying file system," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 2010, pp. 349–364.

[57] D. Garg, F. Pfenning, D. Serenyi, and B. Witten, "A logical representation of common rules for controlling access to classified information," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-09-139, 2009.

[58] J. Garzik, "libATA developer's guide," http://www.kernel.org/doc/htmldocs/libata.html.

[59] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proceedings of the 10th Annual Network and Distributed System Security Symposium*. Reston, VA: Internet Society, Feb. 2003, pp. 131–145.

[60] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 6th USENIX Security Symposium*. Berkeley, CA: USENIX Association, Jul. 1996.

[61] V. Gough, "EncFS encrypted filesystem," http://arg0.net/encfs/.

[62] C. G. Gray and D. R. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Dec. 1989, pp. 202–210.

[63] Y. Gurevich and I. Neeman, "DKAL: Distributed-knowledge authorization language," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium*. Los Alamitos, CA: IEEE Computer Society, Jun. 2008, pp. 149–162.

[64] M. A. Halcrow, "Demands, solutions, and improvements for Linux filesystem security," in *Proceedings of the 2004 Linux Symposium*, vol. 1, Jul. 2004, pp. 269–286.

[65] ——, "eCryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the 2005 Linux Symposium*, vol. 1, Jul. 2005, pp. 201–218.

[66] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Certified in-lined reference monitoring on .NET," in *Proceedings of the 2006 ACM Workshop on Programming Languages and Analysis for Security*. New York, NY: ACM Press, Jun. 2006, pp. 7–16.

[67] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 public key infrastructure certificate and CRL profile," Internet Engineering Task Force RFC 2459, 1999.

[68] J. Howell, "Naming and sharing resources across administrative boundaries," Ph.D. dissertation, Dartmouth College, Hanover, NH, May 2000.

[69] J. Howell and D. Kotz, "End-to-end authorization," in *Proceedings of the 4th Symposium on Operating System Design & Implementation*. Berkeley, CA: USENIX Association, Oct. 2000, pp. 151–164.

[70] IEEE Computer Society, "1003.1e IEEE draft standard for information technology–portable operating system interface (POSIX)," Los Alamitos, CA, 1997.

[71] ITU Telecommunication Standardization Sector (ITU-T), "Information technology–ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," ITU-T X.690 standard.

[72] S. Jain, F. Shafique, V. Djeric, and A. Goel, "Application-level isolation and recovery with Solitude," in *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*. New York, NY: ACM Press, Apr. 2008, pp. 95–107.

[73] T. Jim, "SD3: A trust management system with certified evaluation," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 2001, pp. 106–115.

[74] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, Mar. 2003, pp. 29–42.

[75] V. Kher and Y. Kim, "Building trust in storage outsourcing: Secure accounting of utility storage," in *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*. Los Alamitos, CA: IEEE Computer Society, Oct. 2007, pp. 55–64.

[76] D. Kilpatrick, "Privman: A library for partitioning applications," in *Proceedings of the 2003 USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jun. 2003, pp. 273–284.

[77] M. N. Krohn, P. Efstathopoulos, C. Frey, M. F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. Vandebogart, and D. Ziegler, "Make least privilege a right (not a privilege)," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. Berkeley, CA: USENIX Association, Jun. 2005.

[78] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, pp. 265–310, Jun. 1992.

[79] B. W. Lampson, "Protection," *ACM Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[80] ——, "Computer security in the real world," *IEEE Computer*, vol. 6, no. 37, pp. 37–46, Jun. 2004.

[81] C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek, "Alpaca: Extensible authorization for distributed services," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY: ACM Press, Oct. 2007, pp. 432–444.

[82] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism separation in Hydra," in *Proceedings of the 5th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Nov. 1975, pp. 132–140.

[83] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984.

[84] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of the 6th Symposium on Operating System Design & Implementation*, vol. 6. Berkeley, CA: USENIX Association, Dec. 2004, pp. 91–106.

[85] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: A logic-based approach to distributed authorization," *ACM Transactions on Information and System Security*, vol. 6, pp. 128–171, Feb. 2003.

[86] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 2002, pp. 114–130.

[87] Z. Liang, V. N. Venkatakrishnan, and R. Sekar, "Isolated program execution: An application transparent approach for executing untrusted programs," in *Proceedings of the 19th Annual Computer Security Applications Conference*. Los Alamitos, CA: IEEE Computer Society, Dec. 2003, pp. 182–191.

[88] Linux Kernel, version 2.6.29.2, http://www.kernel.org/.

[89] D. MacKenzie and G. Pottinger, "Mathematics, technology, and trust: Formal verification, computer security, and the U.S. military," *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 41–59, Jul. 1997.

[90] J. Malcolm and B. Robinson, "Using escrowed public keys to enhance confidence in an image authentication scheme," in *IEE Seminar on Secure Images and Image Authentication*, ser. IEE Seminar Digests. London, UK: IEE, Apr. 2000, p. 11.

[91] J. Marchesini, S. Smith, O. Wild, and R. MacDonald, "Experimenting with TCPA/ TCG hardware, Or: How I learned to stop worrying and love the bear," Department of Computer Science/Dartmouth PKI Lab, Dartmouth College, Hanover, NH, Tech. Rep. TR2003-4761, Dec. 2003.

[92] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*. New York, NY: ACM Press, Apr. 2008, pp. 315–328.

[93] Microsoft TechNet, "BitLocker drive encryption overview," http://technet.microsoft.com/en-us/library/cc732774.aspx.

[94] D. A. Miller, "A compact representation of proofs," *Studia Logica*, vol. 46, no. 4, pp. 347–370, 1987.

[95] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis, "Decentralized access control in distributed file systems," *ACM Computing Surveys*, vol. 40, no. 3, pp. 10:1–10:30, Aug. 2008.

[96] S. Motiee, K. Hawkey, and K. Beznosov, "Do Windows users follow the principle of least privilege? Investigating user account control practices," in *Proceedings of the 6th Symposium on Usable Privacy and Security*. New York, NY: ACM Press, Jul. 2010.

[97] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. New York, NY: ACM Press, Jan. 1997, pp. 106–119.

[98] ——, "A scalable architecture for proof-carrying code," in *Proceedings of the 5th International Symposium on Functional and Logic Programming*, ser. Lecture Notes in Computer Science, H. Kuchen and K. Ueda, Eds., vol. 2024. Berlin, Germany: Springer, Mar. 2001, pp. 21–39.

[99] G. C. Necula and P. Lee, "Efficient representation and validation of proofs," in *Proceedings of the 13th IEEE Symposium on Logic in Computer Science*. Los Alamitos, CA: IEEE Computer Society, Jun. 1998, pp. 93–104.

[100] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization tech-

nology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167–178, Aug. 2006.

[101] R. Nelson, D. Becker, J. Brunell, and J. Heimann, "Mutual suspicion for network security," in *Proceedings of the 13th NIST-NCSC National Computer Security Conference*. Baltimore, MD: National Institute of Standards and Technology, Oct. 1990.

[102] New York Times Company, "Guidelines on integrity," http://www.nytco.com/company/business_units/integrity.html, 1999.

[103] G. H. Nibaldi, "Specification of a trusted computing base (TCB)," MITRE Corp., Bedford, MA, Tech. Rep. M79-228, Nov. 1979, NTIS ADA108831.

[104] M. Nizza and P. J. Lyons, "In an Iranian image, a missile too many," *New York Times—The Lede blog*, July 10, 2008.

[105] OpenOffice, http://www.openoffice.org/.

[106] OpenSSL, http://www.openssl.org/.

[107] Organization for the Advancement of Structured Information Standards (OASIS), "Web services security: SOAP message security 1.1 (WS-Security 2004)," http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, Feb. 2006.

[108] D. P. O'Shanahan, "CryptoFS: Fast cryptographic secure NFS," Master's thesis, University of Dublin, Dublin, Ireland, 2000.

[109] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up? Sentiment classification using machine learning techniques," in *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*. Stroudsburg, PA: Association for Computational Linguistics, Jul. 2002, pp. 79–86.

[110] V. Paxson, "GNU Flex (the fast lexical analyzer)," http://www.gnu.org/software/flex/.

[111] B. Peters, "Security considerations in a multi-programmed computer system," in *Proceedings of the AFIPS Spring Joint Computer Conference*. New York, NY: ACM Press, Apr. 1967, pp. 283–286.

[112] F. Pfenning and C. Schürmann, "System description: Twelf—A meta-logical framework for deductive systems," in *Proceedings of the 16th International Conference on Automated Deduction*. London, UK: Springer, Jul. 1999, pp. 202–206.

[113] A. Pimlott and O. Kselyov, "Soutei, a logic-based trust management system, system description," in *Proceedings of the 8th International Symposium on Functional and Logic Programming*, ser. Lecture Notes in Computer Science, M. Hagiya and P. Wadler, Eds., vol. 3945. Berlin, Germany: Springer, Apr. 2006, pp. 130–145.

[114] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, vol. 12. Berkeley, CA: USENIX Association, Aug. 2003, pp. 231–242.

[115] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *Proceedings of the 2002 USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, Jan. 2002, pp. 15–30.

[116] R. Riglar, "FAT16 / FAT32 file IO library," http://hp.www.robs-projects.com/filelib.html, 2010, version 2.5.0.

[117] R. Rivest and B. Lampson, "SDSI—A simple distributed security infrastructure," http://groups.csail.mit.edu/cis/sdsi.html, 1996.

[118] J. Rushby, "A trusted computing base for embedded systems," in *Proceedings of the 7th DoD/NBS Computer Security Conference*, Sep. 1984, pp. 294–311.

[119] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.

[120] R. S. Sandhu, "Lattice-based access control models," *IEEE Computer*, vol. 26, no. 11, pp. 9–19, Nov. 1993.

[121] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[122] C. Saout, "dm-crypt—a device-mapper crypto target," http://www.saout.de/misc/dm-crypt/, Mar. 2004.

[123] G. E. Schalnat, A. Dilger, J. Bowler, G. Randers-Pehrson *et al.*, "libpng: PNG reference library," http://www.libpng.org/pub/png/libpng.html, 2002, version 1.2.5.

[124] M. Schroeder, "Cooperation of mutually suspicious subsystems in a computer utility," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1972.

[125] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: An authorization architecture for trustworthy com-

puting," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Oct. 2011, pp. 249–264.

[126] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad, "Design and implementation of a distributed virtual machine for networked computers," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Dec. 1999, pp. 202–216.

[127] S. Smith and J. Marchesini, *The Craft of System Security*. Boston, MA: Addison Wesley, 2007.

[128] R. Stallman and N. Friedman, "GNU bison—the GNU parser generator," http://www.gnu.org/software/bison/.

[129] C. A. Stein, J. H. Howard, and M. I. Seltzer, "Unifying file system protection," in *Proceedings of the 2001 USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jun. 2001, pp. 79–90.

[130] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "POTSHARDS—A secure, long-term storage system," *ACM Transactions on Storage*, vol. 5, no. 2, pp. 5:1–5:35, Jun. 2009.

[131] Z. Su and G. Wassermann, "The essence of command injection attacks in Web applications," in *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*. New York, NY: ACM Press, Jan. 2006, pp. 372–382.

[132] P. F. Syverson and S. G. Stubblebine, "Group principals and the formalization of anonymity," in *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems(Volume 1)*, ser. Lecture Notes in Computer Science, vol. 1708. Berlin, Germany: Springer, Sep. 1999, pp. 814–833.

[133] M. Szeredi, "FUSE: Filesystem in user-space," http://fuse.sourceforge.net/.

[134] ——, "SSH filesystem," http://fuse.sourceforge.net/sshfs.html.

[135] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, 5th ed. Bell Telephone Laboratories, Jun. 1974.

[136] A. S. Troelstra and D. van Dalen, *Constructivism in Mathematics*, ser. Studies in Logic and the Foundations of Mathematics. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., 1988, vol. 121.

[137] Trusted Computing Group, "Trusted platform module (TPM) specification, version 1.2," https://www.trustedcomputinggroup.org/specs/TPM/.

[138] D. Tsafrir, D. D. Silva, and D. Wagner, "The murky issue of changing process identity: Revising "setuid demystified"," *;login: The USENIX Magazine*, vol. 33, no. 3, pp. 56–66, Jun. 2008.

[139] P. D. Turney, "Thumbs up or thumbs down? Semantic orientation applied to unsupervised classification of reviews," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Stroudsburg, PA: Association for Computational Linguistics, Jul. 2002, pp. 417–424.

[140] D. van Dalen, *Logic and Structure*, 4th ed. Berlin, Germany: Springer, 2004.

[141] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison Wesley, 2001.

[142] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Dec. 1993, pp. 203–216.

[143] W. H. Ware, "Security and privacy in computer systems," in *Proceedings of the AFIPS Spring Joint Computer Conference*. New York, NY: ACM Press, Apr. 1967, pp. 279–282.

[144] C. Weinhold, "Design and implementation of a trustworthy file system for L4," Ph.D. dissertation, Technische Universität Dresden, Dresden, Germany, 2006.

[145] C. Weinhold and H. Härtig, "VPFS: Building a virtual private file system with a small trusted computing base," in *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*. New York, NY: ACM Press, Apr. 2008, pp. 81–93.

[146] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device driver safety through a reference validation mechanism," in *Proceedings of the 8th Symposium on Operating System Design & Implementation*. Berkeley, CA: USENIX Association, Dec. 2008, pp. 241–254.

[147] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the TAOS operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 3–32, Feb. 1994.

[148] T. Wobber, T. L. Rodeheffer, and D. B. Terry, "Policy-based access control for weakly consistent replication," Microsoft Research, Tech. Rep. MSR-TR-2009-15, Jul. 2009.

[149] T. Y. C. Woo and S. S. Lam, "Authentication for distributed systems," *IEEE Computer*, vol. 25, no. 1, pp. 39–52, Jan. 1992.

[150] World Wide Web Consortium, "Web services policy 1.5–framework (WS-Policy)," http://www.w3.org/TR/ws-policy/, Sep. 2007.

[151] C. P. Wright, J. Dave, and E. Zadok, "Cryptographic file systems performance: What you don't know can hurt you," in *Proceedings of the 2nd IEEE Security in Storage Workshop*. Los Alamitos, CA: IEEE Computer Society, Oct. 2003, pp. 47–61.

[152] C. P. Wright, M. C. Martino, and E. Zadok, "NCryptfs: A secure and convenient cryptographic file system," in *Proceedings of the 2003 USENIX Annual Technical Conference, General Track*. Berkeley, CA: USENIX Association, Jun. 2003, pp. 197–210.

[153] XOM, http://www.xom.nu/.

[154] J. Yao, S. Chen, S. Nepal, D. Levy, and J. Zic, "TrustStore: Making Amazon S3 trustworthy with services composition," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*. Los Alamitos, CA: IEEE Computer Society, May 2010, pp. 600–605.

[155] J. R. Young, "Journals find many images in research are faked," *The Chronicle of Higher Education*, vol. 54, no. 39, p. A1, Jun. 2008.

[156] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A stackable vnode level encryption file system," Computer Science Department, Columbia University, New York, NY, Tech. Rep. CUCS-021-89, 1998.

[157] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. New York, NY: ACM Press, Oct. 2001, pp. 1–14.

[158] L. Zhang, "Why TCP timers don't work well," in *Proceedings of the 1986 ACM Conference on Communications Architectures & Protocols*. New York, NY: ACM Press, Aug. 1986, pp. 397–405.