Naming and sharing resources across administrative boundaries

A Dissertation

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Jonathan R. Howell

DARTMOUTH COLLEGE

Hanover, New Hampshire

21 April 2000

Examining Committee:

David Kotz (chairman)

Robert Gray

Doug McIlroy

Margo Seltzer

Roger Sloboda
Dean of Graduate Studies

for Andy

Abstract

I tackle the problem of naming and sharing resources across administrative boundaries. Conventional systems manifest the hierarchy of typical administrative structure in the structure of their own mechanism. While natural for communication that follows hierarchical patterns, such systems interfere with naming and sharing that crosses administrative boundaries, and therefore cause headaches for both users and administrators. I propose to organize resource naming and security not around administrative domains, but around the sharing patterns of users.

The dissertation is organized into four main parts. First, I discuss the challenges and tradeoffs involved in naming resources, and consider a variety of existing approaches to naming.

Second, I consider the architectural requirements for a user-centric sharing. I evaluate existing systems with respect to these requirements.

Third, to support the sharing architecture, I develop a formal logic of sharing that captures the notion of *restricted delegation*. Restricted delegation ensures that users can use the same mechanisms to share resources consistently, regardless of the origin of the resource, or with whom the user wishes to share the resource next. A formal semantics gives unambiguous meaning to the logic. I apply the formalism to the Simple Public Key Infrastructure, and discuss how the formalism either supports or discourages potential extensions to such a system.

Finally, I use the formalism to drive a user-centric sharing implementation for distributed systems. I show how this implementation enables *end-to-end authorization*, a feature that makes heterogeneous distributed systems more secure and easier to audit. Conventionally, gateway services that bridge administrative domains, add abstraction, or translate protocols typically impede the flow of authorization information from client to server. In contrast, end-to-end authorization enables us to build gateway services that preserve authorization information, hence we reduce the size of the trusted computing base and enable more effective auditing. I demonstrate my implementation, and show how it enables end-to-end authorization across various boundaries. I measure our implementation, and argue that its performance tracks that of similar authorization mechanisms without end-to-end structure.

I conclude that my user-centric philosophy of naming and sharing benefits both users and administrators.

Contents

Abstract **iii**

I Introduction **1**

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Contributions . . . 4
1.3 Overview . . . 4

II Naming **7**

2 Naming **9**
2.1 Qualities of naming systems . . . 9
2.1.1 User cost of establishing a name binding . . . 9
2.1.2 Mnemonic value of a name binding . . . 10
2.1.3 Semantic value of a name binding . . . 10
2.1.4 Performance of name resolution . . . 11
2.1.5 User cost of sharing a name binding . . . 11
2.1.6 User cost of managing storage . . . 11
2.2 Design and qualities of existing systems . . . 11
2.2.1 Global namespaces . . . 12
2.2.2 Plan 9 . . . 12
2.2.3 Sollins' negotiated names . . . 13
2.2.4 Query-based naming . . . 13
2.3 Design of Snowflake naming . . . 13
2.4 Implementation of Snowflake naming . . . 14
2.4.1 Infrastructure . . . 15
2.4.2 Applications . . . 16
2.4.3 System structure . . . 17
2.5 Qualities of Snowflake naming . . . 18
2.5.1 Cost of establishing a name binding . . . 18
2.5.2 Mnemonic value of name bindings . . . 18
2.5.3 Semantic value of name bindings . . . 18

2.5.4 Performance of name resolution 20
2.5.5 User cost of sharing name bindings 21
2.5.6 User cost of managing storage 21

III Sharing Overview **23**

3 Sharing **25**
3.1 Mandatory and discretionary access controls 25
3.2 Qualities of security models 25
3.2.1 Consistent sharing 26
3.2.2 Transitive delegation 26
3.2.3 Restricted delegation 26
3.2.4 Auditable access control 26
3.2.5 User cost of sharing 26
3.2.6 User cost of administration 27
3.2.7 Performance 27
3.2.8 Formal model of sharing 27
3.3 Design and qualities of existing systems 28
3.3.1 Formal models 28
3.3.2 Systems 30
3.4 Design of Snowflake sharing 32
3.4.1 Restricted transitive delegation 32
3.4.2 Arbitrary principals 33
3.4.3 Groups 33
3.4.4 Conjunctions 33
3.4.5 Quoting principals 33
3.5 Preview of the sharing chapters 33

IV Sharing Formalism **35**

4 The logic and semantics of restricted delegation **37**
4.1 Semantics of $\overset{T}{\Rightarrow}$ 39
4.1.1 The $\xrightarrow{T}$ relation 41
4.1.2 The $\overset{T}{\Rrightarrow}$ relation 41
4.1.3 Relationships among the relations 42
4.2 Additional benefits of $\overset{T}{\Rightarrow}$ 42
4.2.1 Supplanting *controls* 42
4.2.2 Supplanting roles 43
4.2.3 Formalizing statement expiration 44

5 The semantics of SPKI names **45**
5.1 The logic of names . . . 45
5.2 The semantics of names . . . 47
5.3 Challenges in the name semantics . . . 48
5.4 Abadi's semantics for linked local namespaces . . . 49

6 The semantics of authorization tag notation **51**
6.1 Overview . . . 51
6.2 Bytestrings and powers . . . 51
6.3 Auths . . . 52
6.4 Closure of auths under intersection . . . 53
6.5 Tags . . . 55
6.5.1 The null tag . . . 56
6.5.2 Lists have an initial bytestring element . . . 56
6.5.3 Special tags cause havoc . . . 56
6.5.4 Semantics of special tags . . . 57
6.6 The meaning of intersection . . . 58
6.7 Order dependence . . . 59
6.7.1 Handling non-bytestring attribute names . . . 60
6.7.2 Interference between ordered and named attributes . . . 60
6.7.3 Intersection of lists containing named attributes . . . 60
6.7.4 Recommendations for the use of ordered and named attributes . . . 61
6.8 Analogy with Dedekind cuts . . . 61

7 Modeling SPKI **63**
7.1 Delegation control . . . 63
7.2 Restriction . . . 64
7.3 Linked local namespaces . . . 64
7.4 Threshold subjects . . . 65
7.5 Auth tags . . . 65
7.6 Tuple reduction . . . 65
7.7 Validity conditions . . . 65
7.8 Safe extensions . . . 66
7.9 Dangerous extensions . . . 67
7.10 Related work . . . 68

V Sharing Implementation **69**

8 End-to-end authorization **71**
8.1 Spanning administrative domains . . . 71
8.2 Spanning network scales . . . 73
8.3 Spanning levels of abstraction . . . 74
8.4 Spanning protocols . . . 75

9 Infrastructure **77**
9.1 Statements . . . 77
9.2 Principals . . . 77
9.3 Proofs . . . 78
9.4 The prover . . . 79

10 Channels **81**
10.1 Secure channels . . . 81
10.1.1 How channels work . . . 82
10.1.2 A channel optimization . . . 83
10.2 Local channels . . . 84
10.3 Signed requests . . . 85
10.3.1 Signed request optimization . . . 85
10.3.2 Server authorization . . . 86
10.3.3 Server implementation . . . 87
10.3.4 Client implementation . . . 88

11 Applications **91**
11.1 Protected web server . . . 91
11.2 Protected database . . . 91
11.3 Quoting protocol gateway . . . 91

12 Measurement **93**
12.1 Experimental methodology . . . 93
12.2 RMI authorization with Snowflake . . . 94
12.3 HTTP authorization with Snowflake . . . 95
12.3.1 Client baseline . . . 95
12.3.2 Server baseline . . . 95
12.3.3 Network baseline . . . 96
12.3.4 Snowflake costs . . . 96
12.3.5 Comparison with SSL costs . . . 97
12.4 Gateway authorization . . . 99
12.5 Observations . . . 99
12.5.1 Comparable operations . . . 99
12.5.2 The performance-security tradeoff . . . 100
12.5.3 Slow libraries . . . 100
12.5.4 Performance lessons . . . 100

13 Qualities of Snowflake sharing and security **101**
13.1 Consistent sharing . . . 101
13.2 Transitive delegation . . . 101
13.3 Restricted delegation . . . 101
13.4 Auditable access control . . . 101
13.5 User cost of sharing . . . 102
13.6 User cost of administration . . . 102
13.7 Performance . . . 102

13.8 Formal model of sharing . . . 103

VI Summary **105**

14 Related work **107**
14.1 Microkernels . . . 107
14.2 Retrofitting existing architectures . . . 108
14.3 Single-system-image clusters . . . 108
14.4 Distributed file systems . . . 109
14.5 Worldwide systems . . . 109

15 Conclusion and Contributions **111**

16 Future work **113**
16.1 End-to-end secrecy . . . 113
16.2 Applications . . . 113
16.3 Performance . . . 114

A Proofs **117**
A.1 Construction of ϕ_T . . . 117
A.2 Equivalence of ϕ_T^R and ϕ_T^+ definitions of $\stackrel{T}{\Rightarrow}$ . . . 117
A.3 An undesirable semantics for $\stackrel{T}{\Rightarrow}$ . . . 118
A.4 Proof of soundness . . . 118
A.5 Relationships among the restricted relations . . . 126

B Review: the logic of belief **129**
B.1 Compound principals . . . 132
B.1.1 The nature of principal relations . . . 133
B.1.2 Trust . . . 134

C Review: the original Calculus for Access Control **135**
C.1 The calculus of principals . . . 136
C.2 The "speaks for" relation . . . 137
C.3 Access Control Lists . . . 138
C.4 Higher-level operators . . . 139
C.5 Roles and the "**as**" operator . . . 139
C.5.1 Semantics for Roles . . . 139
C.6 Delegation and the "**for**" operator . . . 141

D Review: The Simple Public Key Infrastructure **143**
D.1 Certificate types . . . 143
D.2 The SPKI 5-tuple . . . 144
D.3 The SPKI 4-tuple . . . 144
D.4 Tuple reduction . . . 145

E Experimental data **147**

List of Figures

2.1 *The layers of software that comprise the Snowflake naming implementation.* 14
2.2 *The same namespace can capture names at multiple levels of abstraction.* . . 19

5.1 *An example that shows when inherited names can become idempotent.* . . . 46

8.1 *A resource server contains an ACL that refers directly to Alice, a user in the same administrative domain.* 72
8.2 *With restricted delegation, Alice can introduce the remote principal Bob, and describe what authority he has over her resources.* 72
8.3 *When Alice connects to a resource server from the same machine, the server may trust the kernel to correctly identify her* 73
8.4 *Alice's request for a high-level resource involves not only her authority over the final low-level resource, but some interaction with the service providing a level of abstraction.* 74
8.5 *Gateways are necessary to translate between protocols, but they frequently impede the flow of authorization information.* 75

9.1 *A linear sequence (left) designed to advance a stack machine (right) to arrive at an intended conclusion ($H_D \Rightarrow K_C \cdot N$).* 79
9.2 *A structured proof that shows that a name defined by a client ($K_C \cdot N$) is bound to a particular document (H_D).* 80

10.1 *Treating a channel as a principal* 82
10.2 *How my* `ssh` *RMI channel is integrated with Snowflake's authentication service.* 82
10.3 *The HTTP authorization protocol.* 86
10.4 *An HTTP authorization challenge message from a Snowflake server.* 87
10.5 *A response message from a Snowflake proxy.* 88

A.1 *In this example, $T = \{s\}$. Notice that $B \stackrel{T}{\not\rightarrow} A$.* 119
A.2 *A counterexample showing why two delegations for sets S and T do not imply a delegation for set $S \cup T$ (Result E7).* 123
A.3 *A model that demonstrates Result E9.* 123
A.4 *A counterexample that shows $B \stackrel{T}{\Rrightarrow} A$ does not imply $B \stackrel{T}{\Rightarrow} A$.* 126
A.5 *A counterexample that shows $B \stackrel{T}{\Rightarrow} A$ does not imply $B \stackrel{T}{\Rrightarrow} A$.* 127
A.6 *Examples that show why the relation $\stackrel{T}{\rightarrow}$ is weaker than $\stackrel{T}{\Rightarrow}$ and $\stackrel{T}{\Rrightarrow}$.* 128

B.1 *A model of eight worlds (circles), illustrating the relationship between the accessibility relation (arrows) and the the modal operator (A **believes**).* . . 130

C.1 *Roles reduce relations with which they are composed.* 140

List of Tables

4.1 *The symbols used to represent sets in this dissertation.* 38

5.1 *A guide to translating between Abadi's notation and ours* 49

6.1 *Pairwise possibilities for set intersection.* 54
6.2 *Pairwise possibilities for set intersection in the presence of the range and prefix auth constructors.* 57

12.1 *Hardware and software configurations used in my experiments.* 93
12.2 *A* setup and bandwidth *experiment for RMI. Results from experiments where client and server are on separate hosts appear on lines marked* remote. . . 94
12.3 *Disabling caches reveals the cost of proof generation, transmission and verification.* 95
12.4 *Performance baselines for HTTP 1.0.* 95
12.5 *A* setup and per-connection *experiment that reveals the cost of a single HTTP/1.1 request.* 96
12.6 *Snowflake HTTP client authorization performance.* 97
12.7 *Snowflake HTTP server authorization performance.* 97
12.8 *The SSL* setup and bandwidth *experiment.* 98
12.9 *The cost of a single request over SSL.* 98

E.1 *The correspondence between summary tables and the experimental data they summarize* 147

Part I

Introduction

Chapter 1

Introduction

As computing resources become increasingly ubiquitous, users find that they have access to more and more resources, and that these resources are supplied by a variety of different organizations. Conventional systems descend from the time-sharing tradition and organize resources into administrative domains. A user connects to an administrative domain ("logs in"), and can then access the resources from that domain uniformly and easily. To access resources in other domains, however, the user must jump through hoops of varying naming and authorization mechanisms. Access to resources in other administrative domains is less uniform and more difficult than access to resources in the local domain.

In contrast, I propose to organize resources not around administrative domains, but around users. The ultimate goal is for each user to have seamless, simple access to all of the resources he or she is permitted to use. Resources must still change hands from those who supply them to those who use them, but instead of mediating this handoff with an asymmetric relationship between administrators and users, all sharing is from one user to another. An "administrator" is no longer a special entity, but simply a role that one user takes on when sharing resources with another. This approach banishes the concept of "administrative domain" from mechanism to policy. Users access resources provided to them by multiple "suppliers" (other users in the role of administrators); hence resource naming and authorization is uniform and simple. I call the project Snowflake.

1.1 Motivation

Organizing resource administration around user-to-user sharing has benefits for both users and administrators.

A user typically does not care about which administrative domain contains a resource he wants to use; he just wants consistent access to the resource. Communication and resource sharing should correspond to the structure of real relationships, which are not limited by organizational hierarchy. If a user has complete control over the names he uses for resources and the specification of sharing resources with others, then administrative domains will not interfere with his work. Naturally, administrative structure will still affect users, but only in policy, not in mechanism.

Administrators benefit from my approach, to. A central task in system administration

is sharing, just like what users do, but perhaps with more responsibility. Therefore, the mechanisms for administration should be the same; only the policies should change to reflect the administrator's heightened responsibility in the social organization. An added benefit is that a new administrator would have an easier transition from his role as a user by adding responsibility incrementally to his sharing tasks.

1.2 Contributions

The contributions this project makes to distributed computing are:

1. the observation that resource naming and security mechanisms should be organized around user-to-user sharing, not administrative domains, to make resource access uniform and simple in a world where users access resources from a variety of sources,
2. an enumeration of qualities needed for naming and sharing to span administrative domains,
3. a path-based naming mechanism that reflects user-to-user relationships rather than administrative hierarchy,
4. a path-based authorization mechanism, backed by a thorough formalism, that reflects user-to-user relationships, not administrative hierarchy,
5. an *end-to-end* approach to authorization that has benefits even within administrative domains,
6. a proposal for a specific layering of application and system software that helps ensure that naming and authorization remain uniform across applications and systems, and
7. a prototype system with applications that demonstrates the naming and sharing mechanisms at work and illustrates how my model compares with conventionally-organized systems and applications.

1.3 Overview

The dissertation is divided into four parts.

In Part II, I describe how to name resources independently of administrative domains, and compare my approach to related approaches.

Part III describes how to share resources and protect them independently of administrative domain. Its two chapters outline the axes along which a sharing design should be measured, and present an overview of the design for sharing in Snowflake.

Part IV details a formal semantics for sharing that meets the goals presented in the preceding part. Chapter 4 develops the semantics, Chapter 5 adds semantics for strongly-authenticated names, and Chapter 6 adds semantics for delegation restriction sets. In Chapter 7 I connect the semantics to an existing sharing implementation called the Simple Public Key Infrastructure (SPKI), and then extend its applicability to meet the sharing goals of Snowflake.

In Part V, I discuss my implementation of Snowflake sharing. I establish its relevance to general end-to-end authorization problems, including that of authorizing resources across administrative domains, in Chapter 8. Then I describe the infrastructure I built to support sharing in Chapter 9, and the channels and applications that use that infrastructure in Chapters 10 and 11. I measure the performance characteristics of my implementation in Chapter 12.

Part VI summarizes the dissertation. In Chapter 14, I discuss related work with similar goals. I present the conclusions of my work in Chapter 15, and propose future directions in Chapter 16.

In Appendix A I prove statements presented in Part IV. Appendix B reviews modal logic, Appendix C reviews the Calculus for Access Control, and Appendix D reviews the Simple Public Key Infrastructure; since my work rests heavily on each of these topics, I include these appendices to provide the reader a convenient and concise introduction. Appendix E contains plots of the experimental data that are summarized in in Chapter 12.

Part II

Naming

Chapter 2

Naming

In this chapter, I describe and motivate important qualities we desire in a naming system. Then I describe some existing systems, pointing out the relevant design choices they embody, and how those choices affect the qualities of the resulting system. Finally, I describe the naming system I built and tested, and the qualities it exhibits. I make design tradeoffs to better provide resource naming that spans administrative domains.

2.1 Qualities of naming systems

Saltzer and Watson provide thorough surveys of naming issues [Sal78, Wat81]. In designing a naming system to span administrative domains, we consider making different design tradeoffs than have been made in conventional systems. In this section I describe six qualitative measures of a naming system, so that I have a standard by which to evaluate my design.

2.1.1 User cost of establishing a name binding

"User cost" is the per-operation effort a user must expend plus the learning curve the user must have overcome to perform some operation. In this case, the operation is the binding of a name in his namespace to a resource.

Content-based naming systems, for example, have very low user cost for establishing name bindings: the user need only create or modify a resource. The content-based naming system considers the new content of the resource, and any query on the terms or objects appearing in the new content will turn up the modified resource: the "name" (terms or objects) has been automatically bound to the resource by virtue of its appearance in the resource.

A medium cost is found in systems where users rename resources within a single naming system. For example, Unix soft links have medium user cost: the user need only know the name (at the same level of abstraction) of the target resource to grant it a new name.

Naming systems exhibit high user costs for establishing name bindings when they require a user to specify the target resource in some other (generally lower-level) naming system. For example, before one can name a resource in a Plan 9 file system, one must *mount* the file system, an operation outside the consistent naming scheme of Plan 9. To establish a

name binding, the user needs to know about a separate naming scheme, the addresses of file systems. Likewise, to name a resource in the namespace of URLs, one typically needs to know the address (lower-level name) for the resource in terms of the DNS name of the server machine and the file system path on that machine.

2.1.2 Mnemonic value of a name binding

The mnemonic value of a name binding refers to the meaning of the symbolic name to a human user, or to the human reader of code that invokes the name. In systems that embed location information into names, we would expect names to have less mnemonic value. Although perhaps they convey location information consistently, they cannot convey other information as conveniently, nor can resources be grouped by properties other than location. Content-based naming systems encode the content of resources into their names, which one would expect to be quite mnemonic. Sometimes, however, one may want to name a resource based on a property orthogonal to the resource's content, such as where the resource came from, or how it should be processed next.

2.1.3 Semantic value of a name binding

A name should convey some meaning about the resource it names. While meaning may contribute to memorability (mnemonic value), by semantic value I refer specifically to the meaning that a program may automatically infer from a name. Programs often depend on conventionally-structured names to find particular resources.

In a conventional Unix-like environment, names are bound to location. A program may have to look for a resource under three different names in two naming systems: the file `/usr/lib/program.defaults` for system-wide default options, the file `$HOME/.program.rc` for user-specific options, and the environment variable `$PROGRAM_CONFIG` for invocation-specific options. These three names correspond to three separate locations: administrator-controlled persistent storage, user-controlled persistent storage, and user-controlled transient environment storage.

One factor affecting both mnemonic and semantic value of names is *name consistency*; in our case, the consistency of names across even administrative boundaries. By name consistency, I mean the property that a name consistently maps to the resource, and a resource consistently has the same name. Imagine writing a document with a bibliography. If the name of the file containing the bibliography changes depending on whether the typesetting program is running in the same domain as the user or not, then the bibliography may only be used in its "home" environment.

When the naming system does not encourage name consistency, individual applications may support name consistency through application-specific means. For example, IMAP-based mail readers hide from the user the fact that the mail store and its folders may be in a different administrative domain than the mail reader client program. Application-specific naming, of course, only helps with a specific application. I argue that the naming system should promote name consistency transparently to applications, so that all applications automatically reflect the benefits of consistent names.

Content-based and query-based naming systems can have trouble providing programs with semantic value. The naming lookup operation in such systems generally may return

zero, one, or more results, depending on how many resources matched the query. Zero results may mean either that no such resource exists in the scope of the query, or that some matching resource may exist, but that the resource was unavailable or slow when the query was performed. The programmer must be prepared for these ambiguities at each lookup operation, and they often add unfortunate complexity. I describe systems that provide more deterministic semantics as having *robust queries*.

2.1.4 Performance of name resolution

By performance we mean literally the computational cost and latency associated with resolving a name to a resource. We expect systems that embed location information in names to have an inherent performance advantage, since they preclude some levels of indirection.

2.1.5 User cost of sharing a name binding

How difficult is it for a user to share a resource with another? Ideally, one shares a resource the same way one invokes it, using the same name.

2.1.6 User cost of managing storage

Every resource is implemented by some underlying resource. Some resources are implemented in firmware or hardware (disk blocks, RAM pages, CPU cycles, sheets of printed paper). Other resources are abstract, built from a portion of some underlying software-accessible resource. A process or task is made from (multiplexed) CPU cycles and RAM pages. A file is made from disk blocks and a RAM cache. We build increasingly abstract resources from underlying concrete resources; the challenge is to manage the underlying resources to ensure they are available in sufficient supply for the abstract resources we desire.

What does this mean for naming? If a name is bound to an abstract resource, a user may need to manage the concrete resources that support the abstract resource. How does he name the concrete resources? In naming systems that embed location in resource names, the name of the abstract resource can often be reverse-mapped to provide a name in some lower-level naming system for the concrete resource. Such systems may have heterogeneous naming levels, one for each level of resource abstraction. As an example, consider Unix paths. If a file creation fails due to a disk space shortage, one may use the mount table to discover the name of the concrete resource (the disk partition) corresponding to the abstract resource (the directory) that exhibited the resource shortage.

2.2 Design and qualities of existing systems

Having highlighted six important qualities of naming systems, I now explore several existing systems in detail and highlight both their strong and weak qualities. This presentation focuses on qualities that enable or interfere with naming across administrative boundaries.

2.2.1 Global namespaces

A common naming structure assumes that all names belong to a single global namespace. Global namespaces are natural for some scope, but invariably they must be extended to add functionality or integrate multiple "global" namespaces into a larger system.

For example, Unix mount tables integrate multiple file systems into a directory tree. As a result, each resource pathname reflects its storage location in its prefix. Similarly, the Unix r-commands form names as `hostname:path/on/host`. A common contemporary idiom is a set of machines with file systems cross-mounted with NFS; here too resources with common location have a common prefix. The same is true of AFS volumes. Although AFS is a distributed protocol, individual volumes (whose resources have names with common prefixes) reside on a single server host. World-wide web uniform resource locators (URLs) have the same problem; they encode both the network location of the host and the location of the service on the host that provides the service. Managing storage of web resources involves manipulating names in the underlying file system, a level of abstraction below URL names.

Symbolic links can be used to hide the locations of files in global namespaces. For example, another common idiom is to group files by home directories. With some effort, a system administrator can arrange for the path `/home/bob` to always refer Bob's home directory, even when the directory changes locations.

In each of these examples, either the mnemonic and semantic value of names suffers (because names must contain location information of little value to the human or program using the name), or the user must bind a new name to the resource to abstract away the location information. But a typical user is only allowed to bind names in certain parts of the global namespace, such as his home directory. When an administrator binds a short pathname (one that has no embedded location) to a resource, that binding only affects a single host; he must expend effort binding the name on every host for it to appear consistently. Worse yet, short names can only be bound by the administrator, limiting their applicability to within a single administrative domain.

The Sprite cluster operating system provides administrators with a way to bind names consistently across an entire cluster [OCD+88, WO86]. Users are still limited to creating bindings under the prefixes they control, however, and the solution only works within an administrative domain, since its tools are per-cluster, not per-user.

2.2.2 Plan 9

In Plan 9, all name bindings (at the file name level, which is used pervasively in Plan 9) visible to a user may be defined by the user. A tree of named resources is implemented by a *file system*. The Plan 9 notion of file system refers only to the read-write interface to resources, not to their nature as static files; resources in a file system may be arbitrarily abstract. A user has control over the name bindings his programs see by configuring his *mount table*. As in Unix, the mount table describes how the trees of named resources in file systems are assembled into the name tree visible to the calling program. Unlike Unix, in Plan 9 each process can have its own mount table, so that a user may control the name bindings visible in the applications he runs.

Plan 9 names have good mnemonic and semantic value. The difficulties are the user

cost of establishing and sharing name bindings. To establish a name binding within a file system, one simply uses conventional file system operations to create or rename resources. If the file system is not writable by the user, however, he may have to replace the file system in his mount table with another file system that allows him to change the bindings he sees. The replacement file system may point to resources in the original file system. Thus, establishing name bindings is sometimes expensive.

More importantly, the user cost of sharing Plan 9 name bindings is high. One can only share a resource by name if one is sure the name path in the recipient's mount table refers to equivalent resources. Otherwise, one must communicate the appropriate mount table or file system name (in a lower-level "address space") to the recipient before the name becomes meaningful.

2.2.3 Sollins' negotiated names

Karen Sollins' dissertation surveys several approaches to naming in distributed systems. Based on a sociological evaluation of how humans name entities by an iterative mechanism, she proposes that humans using computer applications and programs themselves should similarly negotiate names for resources. She argues that in so negotiating names, groups of clients of the naming system will arrive at names that have high mnemonic value [Sol85].

The cost of establishing a (shared) name binding in Sollins' system, then, is fairly high. Names that are not negotiated in advance to be sharable cannot be shared among users without engaging in the negotiation protocol. Sharing a set of names (a context) requires the recipient to aggregate the received context with his existing context; by Sollins' description, the aggregation algorithm is complex. It is also unclear how useful negotiated names are among a group of entirely programmatic entities, since they are based on a social construct.

2.2.4 Query-based naming

In query-based naming systems, one names a resource according to its attributes or its contents. Naming resources by content provides zero-cost name binding, since the creation or modification of a resource defines its names. Examples of query-based systems include the semantic file system [GJSJ91] and Gopal and Manber's hybrid file system [GM98].

Naming resources by attributes involves some effort in defining appropriate attributes, but produces names with high mnemonic and semantic value. The Placeless system is notable because it allows users to define their own attribute-based names for any resource [dLPT+99]. Other attribute-query naming systems include Active Names [VDAA99], the Intentional Naming System [AWSBL99], and Jini [Wal98].

Both kinds of query-based systems have performance challenges and difficulties managing storage due to the indirect nature of the names. Furthermore, in distributed systems, nondeterministic query results can erode the semantic value of the names.

2.3 Design of Snowflake naming

Snowflake's design for naming aims to enable users to name and share resources with minimal interference from administrative boundaries. With that goal in mind, I consider certain

qualities of the naming system more important than others.

The mnemonic and semantic value of name bindings should be high. Specifically, we wish to avoid tying them to location or administrative structure. The user cost of establishing a name binding should be fairly low, and the user cost of sharing a name binding with another user should be low. For my purposes, the performance of resolving a query and the user costs of managing storage are of secondary importance.

2.4 Implementation of Snowflake naming

The implementation of my naming design involved implementing both the infrastructure to support the design as well as applications to demonstrate the structure of the implementation and its impact on program design. Figure 2.1 shows the organization of the components of the Snowflake naming implementation.

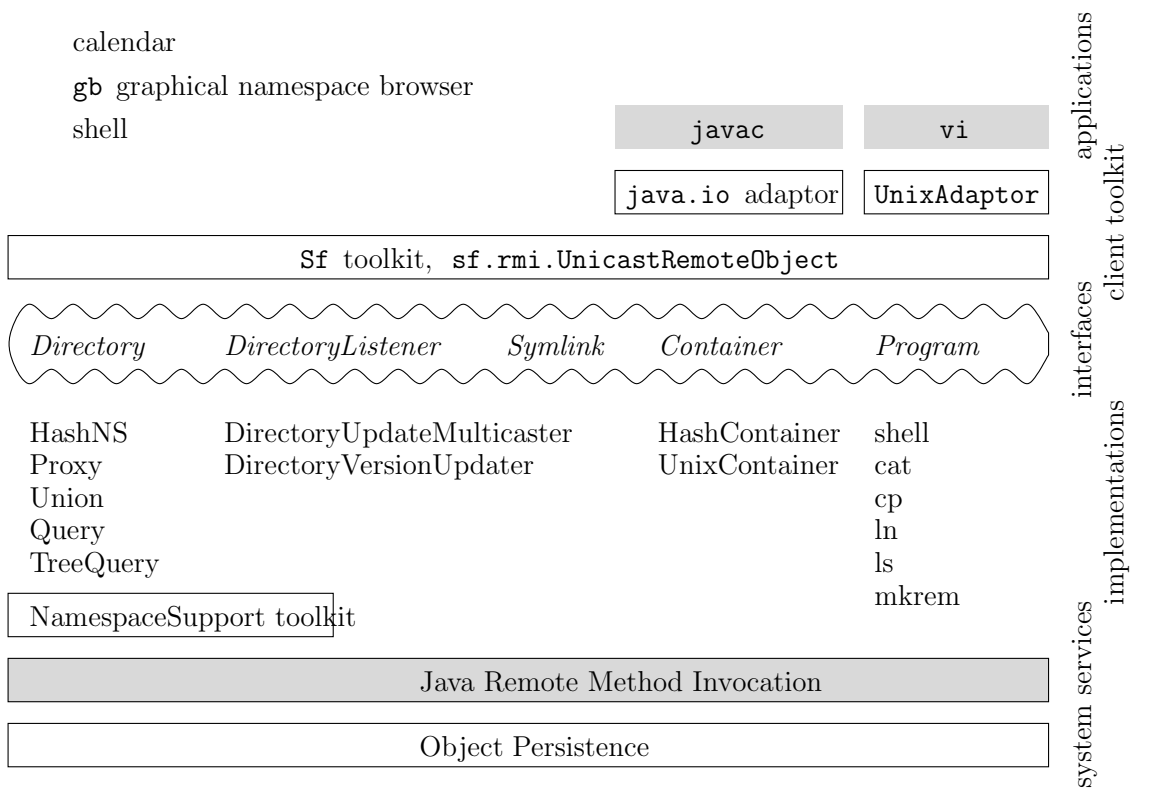


Figure 2.1: *The layers of software that comprise the Snowflake naming implementation. The interfaces (in italics) determine the structure of the system; all name bindings are held in implementations of the* `Directory`*interface. Grey boxes indicate external components used in my environment.*

2.4.1 Infrastructure

The naming infrastructure in Snowflake consists of the interfaces that shape the architecture, implementations of that interface, and client-side support routines that enhance the application programming interface. I chose to implement Snowflake in Java to exploit Remote Method Invocation, a choice that saved me from implementing our own remote-procedure-call mechanism.

The `Directory` interface

I built a naming interface in Java, and several implementations of that interface. I call the interface `Directory`, since *namespace* usually refers to the set of all names that can be specified in a given naming system. The `Directory` interface is `Remote`, meaning that implementations of names (the level below Snowflake naming) may automatically employ Java Remote Method Invocation (RMI). An address (a name at a level below semantic names) in RMI consists of a hostname, port number, and object identifier, so an RMI address has a loose notion of location or storage.

I also built implementations of the name interface; that is, objects that store name-to-object bindings in a variety of forms. The `HashNS` object, for example, implements the minimum functionality, mapping a name to an arbitrary `Remote` object. `HashNS` makes no effort to ensure that the resource is `Remote`, but unless it is, it has no globally accessible address, and hence is not usable by programs residing in other Java Virtual Machines (JVMs). The `Proxy` object transparently redirects requests to another namespace. The redirection is invisible to any client that does not explicitly inquire about the type of the `Proxy` object. Two classes `Query` and `TimeQuery`, part of a Snowflake calendar application, resolve lookup requests by querying an external (web-based) calendar service and packaging the replies as `Remote` objects. A `Union` is a name space that gives a view of several other name spaces, overlaid over one another. It is analogous to union mounts in Plan 9 or 4.4 BSD Unix.

The `Directory` implementations rely on a toolkit called `DirectorySupport` that provides parsing of pathnames into lists of component names, and default implementations of recursive name resolution methods.

The `Container` interface

Another interface called `Container` extends `Directory`; it is the interface to storage management in Snowflake. The `UnixContainer` is an implementation of `Container` that allocates only file-type objects and stores them in the Unix file system; the objects that implement the file resources supply special `Remote` versions of Java streams to make them usable by clients in any location. A more versatile `HashContainer` allocates or copies objects into virtual memory in the same JVM that implements the `HashContainer`. I implemented a checkpointer sophisticated enough to save all of the state of a JVM, so that objects are stored persistently across machine crashes and reboots [How99]. The checkpointer provides the object persistence service in Figure 2.1.

The `DirectoryListener` interface

The `DirectoryListener` interface extends the `Directory` interface to enable applications to discover when name bindings in a `Directory` change. I built a `DirectoryUpdateMulticaster` that distributes update events when they occur, and a `DirectoryVersionUpdater` that polls naïve `Directory`s and generates update events when their bindings change.

The client toolkit

Clients of the `Directory` interface use a naming toolkit called `Sf` that provides client services (but never bindings, to ensure that name bindings are always sharable). The `Sf` lookup tools automatically recognize and resolve special `Symlink` marker objects. `Sf` also maintains a per-thread "current root directory" context to provide programmers with a convenient dynamically-scoped abstraction analogous to the current working directory in Unix. The `Sf` current directory, however, is actually the client's active *root* directory, and thus the entire space is under the control of the process. The notion of a current working directory (the location where a user expects program operations to occur by default) is implemented as a binding to the name `cwd` in the root directory.

When a Snowflake object extends `sf.rmi.UnicastRemoteObject`, and its stub loses its connection to the implementation, the stub will automatically rebind to the implementation *by name*. The stub keeps track of the name lookup that initially resolved to produce itself, as well as the `Directory` that lookup occurred in. What happens if that `Directory` object has also become disconnected? Since `Directory` implementations typically extend `sf.rmi.UnicastRemoteObject`, the process automatically recurses upward to the first name binding that is still intact, then drills back down, re-resolving names, until the stub with the broken reference has acquired a fresh reference.

Hence objects can change location, and the dynamic `sf.rmi.UnicastRemoteObject` stubs will fault and automatically rediscover the new locations of their implementations. Most importantly, the re-resolution uses the name path by which the object was originally referred, so that the semantics of those names are preserved. Rob Pike says that the Plan 9 project happened upon the same solution to a similar problem: determining a meaningful value for `pwd` (print working directory) in Plan 9's per-user namespaces [Pik00]. Note that I make the feature optional because it changes the semantics of RMI: A stub may rebind and find itself attached to a different implementation than before.

This service is especially important for my persistent containers, because whenever they fail and restart, the objects inside acquire new RMI addresses, regardless of whether the container is restarted on the same host.

2.4.2 Applications

To demonstrate and experiment with the naming infrastructure, I wrote several applications. The first is a basic shell that provides the user with a command-line interface to a namespace. The shell prepends to typed commands the string `cmd/` and resolves the resulting name in its own name space. The resulting object implements a simple `Program` interface that specifies a `run` method. The shell makes a shallow copy of its current namespace, binds the name

`argv` to the typed arguments, and the names `stdin` and `stdout` to its I/O streams. Then the shell invokes the `Program`, passing it the freshly-synthesized namespace as an argument.

I built a calendar application that uses the `Query` name space resource to query an external calendar and display the results. The `Union` name space can be used to merge two calendars from separate sources into a single virtual resource.

I modified the `java.io` package to resolve name requests through the `Sf` toolkit. This change causes ordinary Java applications to see Snowflake names rather than the default namespace supplied by the underlying operating system. I built this adaptor specifically to enable the Java compiler `javac` to read Java files from and write class files to Snowflake containers.

I also wrote a Unix application adaptor that uses the `/proc` debugging interface to interpose on a running process' system calls. I based the technique and the interposition code on UFO [AISS98]. The adaptor catches name-related system calls such as `open()` and `stat()`, and routes them through the Snowflake client interface. For `open()` calls, the adaptor returns a synthetic file descriptor, and maintains a table that maps each synthetic descriptor to the corresponding object returned from the Snowflake name resolution. Read and write calls to a synthetic descriptor are converted to method calls on the input and output stream interfaces of the underlying Snowflake object.

The adaptor's namespace is overlaid with a lazily-created tree of the `UnixContainer` objects described above, so that when the application looks for a file resource in the Unix namespace, it finds it through the Unix container. Because every namespace is a Snowflake namespace, the user is free to install arbitrary objects at any pathname seen by the applications. I use the adaptor, for example, to run `vi` on text-file objects in the Snowflake namespace.

2.4.3 System structure

Observe that no application needs to know anything about handling naming across administrative boundaries. Instead, each application works with names in a simple namespace. The user's namespace is a separate layer below all Snowflake applications that arranges name distribution as needed. Therefore, an application cannot distinguish between resources hosted in one administrative domain from those living in another.

Contrast this architecture with conventional application structure. In conventional systems, there is no distinct layer responsible for distributed naming, and the problem is solved either by the operating system or by applications. An application may rely upon the operating system to support distribution, in which case the application is bound into an intra-administrative-domain name space. Alternatively, an application may provide its own notion of name distribution that can span administrative domains. The disadvantage to the conventional approach is that it only applies to some applications, and with each application the user must learn to configure the new application to handle distributed naming to exploit its benefits. My applications emphasize the value of this structure.

2.5 Qualities of Snowflake naming

My experience with the applications in Section 2.4.2 gives me a yardstick by which to evaluate Snowflake's approach to naming. Let me examine how Snowflake naming measures up along the axes of Section 2.1.

2.5.1 Cost of establishing a name binding

The cost of establishing a name binding is medium. Users must do so explicitly, but the bindings may always be established in terms of other Snowflake-level names. There is no "file system mounting." When a resource such as a persistent object store is created, it is given an initial Snowflake name with a low-level semantic meaning like "this resource is a container and it lives in Sudikoff Laboratory and it is backed up daily." As the resource is allocated, abstracted, or aggregated into other resources, the resulting resources are given new names with a higher-level semantic meaning. Notably, the new names are specified in terms of the lower-level *Snowflake* names. Although multiple semantic levels of name meaning still exist because that is how abstract resources are built from concrete resources, they are all Snowflake names, and the same tools are used to establish the bindings from each level to the one below. Figure 2.2 illustrates.

2.5.2 Mnemonic value of name bindings

Because a user may establish any name bindings he wishes, and because a user controls for *all* of the bindings he sees, a user may establish as strong a mnemonic value in his names as desires, modulo the following two limitations.

First, many programs expect to find resources under specific names by convention; to interact with such a program, the user must accept the program's idea of the name of the resource, which may not be so mnemonic for the user. This problem reflects the issue of communication between the user and the author of the program.

Second, bindings map strings to objects. The range of the mapping is generous: a mapped object may implement arbitrary interfaces, and it may implement multiple explicit interfaces so that it takes on an appropriate form for different client programs and users. The domain of the mapping, however, is limited to textual strings. Perhaps for some applications, colors or shapes or arbitrary multidimensional objects would make appropriate names. Furthermore, any aggregation of meaning of names must be done by the single object to which that name maps, such as by the the aggregating `Union Directory`. Query-based systems have a system-wide notion of aggregation, so that an object may "participate" in the implementation of a name simply by conforming to the query.

2.5.3 Semantic value of name bindings

Snowflake names are indeed useful to programs. Snowflake programs communicate arguments and common resources (I/O streams) to one another by storing them at conventionally-determined names in the namespace. Because all names are user-determined, there is no resource a program can depend on that the user (or calling program) does not have the opportunity to redefine. By letting the user define any name, Snowflake's naming

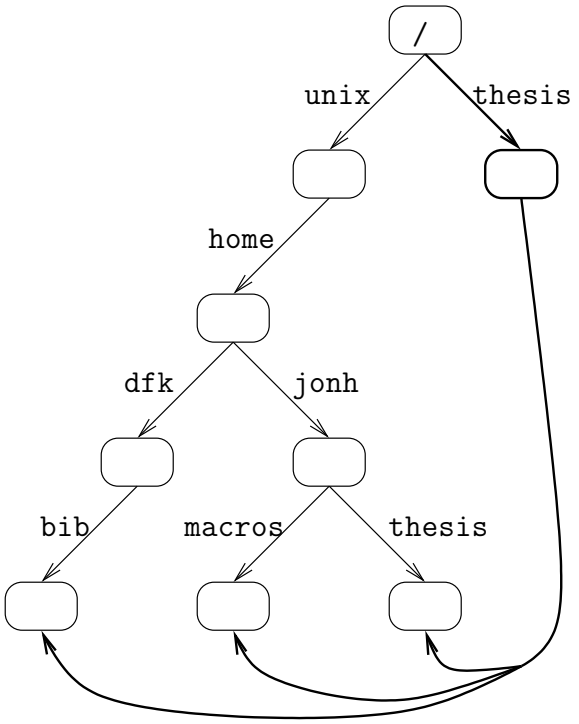


Figure 2.2: *The same namespace can capture names at multiple levels of abstraction. In this Snowflake namespace, lower-level names (thin lines) embed location information, since they reflect location in a Unix file tree. The abstract name "/thesis" is implemented (thick lines) in terms of the lower level names, using a Union directory. Names at both levels of abstraction are Snowflake names.*

model ensures that programs need only look at one name to find a particular resource; contrast this situation with that in Section 2.1.3.

As we discussed in Section 2.1.3, name consistency, the notion that the same name can be used consistently to find a given resource, contributes to both mnemonic and semantic value. Since Snowflake names are implemented by RMI addresses with global scope, since users may define any name binding, and since name bindings may encode high levels of semantic meaning, Snowflake names certainly encourage name consistency across administrative boundaries. The last property is probably the most important for ensuring consistency: once names are decoupled from storage or location, they are easier to use consistently.

Application transparency also encourages name consistency. Since the name binding process is hidden from applications, not implemented by them, the same names in the Snowflake namespace can be used across all applications. The alternative is applications that implement their own cross-domain naming schemes; while these applications may achieve mnemonic and semantic benefits within their own scope, the user does not experience the same benefits consistently across applications. By factoring naming binding out of applications, I have avoided this pitfall.

Are queries robust? Because names are bound directly to an implementing resource, queries are deterministic. One of the possible outcomes of a query is a failure due to damage to the distributed system (e.g., Java RMI's `RemoteException`), but that condition is explicitly reported. Query robustness contributes to semantic value by providing programmatic users of the naming interface with deterministic outcomes that are easier to reason about than the outcome of query-based systems. For most applications, this simplicity is desirable.

2.5.4 Performance of name resolution

Two aspects of the structure of Snowflake names have important performance implications: indirection and cachability. Furthermore, Snowflake's implementation carries with it certain overhead.

First, since Snowflake names do not encode location, there is generally one or more level of indirection between a name presented for resolution and the location of the implementing object. With respect to indirection, "you pay for what you get:" the more abstract the names, the greater the cost involved in resolving the levels of indirection. The benefit is that all of the levels of indirection appear within the same naming scheme, and are all controllable by users.

Indeed, in most systems each level of indirection adds an incremental cost. But in heterogeneous naming systems, the cost can be very high, requiring the deployment of a new mechanism to introduce indirection. Consider the use of round-robin DNS servers and IP rewriting to introduce indirection to location-bound web URLs. In Snowflake's homogeneous naming, there is always an opportunity to replace a name with one with higher-level meaning; there is no need to hack a lower-level addressing scheme to provide indirection.

The second aspect of Snowflake structure that affects performance is cachability. The objects returned by name resolutions can be arbitrary objects, hence they can have arbitrary semantics that may or may not preclude caching of the local implementation of the object. In my prototype, there is no protocol that communicates or negotiates whether a name resolution may be cached. Certainly an object can support its own caching. Since Snowflake allows arbitrary names to be replaced, a user can always ignore the object's cachability semantics and install a cache in front of the object, even when he does not control the underlying object.

In any case, it is fundamentally difficult to provide general cache service for objects with arbitrary semantics. The problems of identifying a "unit" of resource and its identity are very complex in a distributed system with arbitrary object semantics [MKD+00]. I do not argue that caching in the presence of indirection and aliasing is easy, only that indirection happens anyway, so users might as well exploit indirection with per-user names.

The performance (system cost) of resolving a query in the Snowflake implementation is fairly high compared with native operating-system naming, because my implementation depends upon the slow Java substrate. The prototype carries the overhead of Java Serialization (call parameter marshalling) and RMI (IP connection setup and multiplexing overheads).

2.5.5 User cost of sharing name bindings

Because binding implementations always happen on the implementation side of the Snowflake naming interface, and because those implementations are `Remote` objects, they are accessible to any caller. Therefore, users may share bindings directly. Bob may define a Snowflake name `friends/alice` in his namespace, mapped to the top of Alice's naming tree. If Alice wants Bob to read a resource she calls `research/paper`, then Bob's name for that resource is simply `friends/alice/research/paper`.

This cost of sharing names is somewhat more expensive than in systems where all users see the same name tree, since Bob's name for Alice's resource depends on Bob's name for Alice; that is the cost of increasing the mnemonic value of the names. The cost is lower than that in Plan 9, where although names are mnemonic, sharing them requires that Alice communicate to Bob how to mount the same Plan 9 file systems that Alice has mounted. The difference is that file system mounts in Plan 9 are not name bindings (at the same level as most Plan 9 names), thus they cannot be directly shared among users.

2.5.6 User cost of managing storage

Managing storage in Snowflake can be unintuitive. Once names are at a higher semantic level where storage is abstracted away, to manage storage the user must regress to the names that have storage information encoded in them. This user cost is the same one that appears when using symbolic links or location-hiding abstractions in conventional systems. Because Snowflake makes it is so convenient to add abstraction with another level of naming, managing storage often has a higher user cost in Snowflake than in conventional systems.

In my implementation, class evolution presents the primary storage-management difficulty. My persistent Java stores store not only the object data, but the class definitions of objects, so upgrading the classes involves exporting the data out of the objects into a new JVM and discarding the old JVM.

In summary, since Snowflake focuses on accessing resources across administrative domains, it makes different qualitative trade-offs than do conventional systems. Snowflake places a premium on the mnemonic and semantic value of names and the ease with which users can share names. These traits come at a cost in performance and increased difficulty in storage management.

Part III

Sharing Overview

Chapter 3

Sharing

In this chapter, I describe important qualities we might desire in a sharing system. Then I describe some existing systems, pointing out the relevant design choices they embody, and how those choices affect the qualities of the resulting system. Finally, I describe the sharing system I built and tested, and the qualities it exhibits. I make design tradeoffs to better achieve the goal of resource sharing that spans administrative domains.

The dual of the problem is often called "protection" or "security." I use the term "sharing" to highlight the user's goal in using the system: to share resources with others. I assume my work will be used where policy allows users to share resources across administrative domains, and then consider how to provide security given that assumption.

3.1 Mandatory and discretionary access controls

Much of the literature refers to this problem as the "security problem." Security models come in different strengths. Some attempt to prevent any information from flowing where it does not belong; military *multilevel security (MLS)* falls into this category. Loscocco et al. argue in [LSM+98] that mandatory access control is desirable because it limits the burden for security to the system security policy administrator, rather than leaving it up to possibly malicious or careless users.

The very goal of my research, however, is to enable users to access and share resources across administrative boundaries. Therefore, I must consider situations where limiting the security burden to a site administrator is not only impractical, but undesirable. For my purposes, we assume that users can be trusted to exercise discretion in how they use resources. If Alice explicitly shares a resource with Bob, she not only trusts Bob to use the resource responsibly, but she also trusts Bob's discretion in how he further shares the resource. To use it effectively, perhaps Bob must pass on access to the resource to a program or another person.

3.2 Qualities of security models

Having outlined the scope of security problems I want to tackle in my sharing system, I now pinpoint the specific qualities I require.

3.2.1 Consistent sharing

The mechanism of sharing resources is independent of the administrative association of the giver and the recipient. Sharing policy, however, may certainly reflect administrative associations. Because "sharing" a resource in an electronic environment often means authorizing a server to give access to a third party, consistent sharing implies enabling resource servers to reason about previously unknown third parties. This requirement contrasts with many conventional systems, wherein a server need only reason about the set of principals known inside a given administrative domain.

3.2.2 Transitive delegation

A user may share any resource to which he has access with another user, including those resources granted to the user by others. To exploit this quality requires trust in delegates to use a resource responsibly, but ensures that all sharing can be performed with the same mechanism.

3.2.3 Restricted delegation

When sharing a resource with others, a user may always restrict the recipient's access to the resource. That is, when a user receives a resource from another, he should always be able to treat it as a first-class resource, share it again with a third party (transitive sharing), and specify a restriction on how the third party may use the shared resource.

3.2.4 Auditable access control

When resources are shared, the originating resource server can audit how the ultimate resource user came to have access to the resource. In an environment where users must trust delegates to use a resource responsibly, the resource owner should have some auditing mechanism to discover when and how his trust has been abused.

Because every resource, even one received through a restricted delegation, should be "first class," every user sharing a resource should have some opportunity to audit the accesses that involve the virtual resource he shared, but not necessarily the original resource from which it derives.

3.2.5 User cost of sharing

How much trouble is it for one user to share a resource with another? User cost is perhaps more important with sharing than with naming, because not only can a high user cost interfere with the users' computing experience, it can encourage users to circumvent the sharing system, compromising security. The user cost of sharing is evaluated by examing what a user must do to make a resource accessible to another user, and what the second user must do to access the shared resource.

3.2.6 User cost of administration

The administrative cost of a sharing mechanism is largely a function of previous qualities: the user cost of sharing and the auditability of access control. Auditability affects administrators because administrators are responsible to monitor the security of their resources. The user cost of sharing affects administrators in two ways: first, as part of their tasks, administrators must share resources with users; second, administrators must educate users about how to share resources securely.

3.2.7 Performance

Sharing affects performance because it conceptually involves security checks every time a resource is accessed. Different sharing systems invoke different strengths of checks at each access. Performance penalties should be weighted according to the frequency of invocation: common-case accesses to local resources should count more heavily than infrequent accesses to remote resources, for example.

3.2.8 Formal model of sharing

Sharing and security systems frequently have *ad hoc* designs that seem to satisfy common sense; however, common sense can often mislead us. This risk is especially dangerous in security, where we are concerned not with good average-case behavior, but with worst-case behavior in the presence of an adversary.

Formal models for security provide one overriding benefit: unambiguous communication. A formal model clearly communicates between the designer of a system, its implementors, and its users what promises the system makes and what the consequences of certain actions in the system will be.

Formal models are not a panacea for security problems. Any given formal model is a *model*, and hence abstracts away some details of a system that may hide security flaws. Furthermore, a formal model requires interpretation to be applied in a real environment; if the users of the system do not understand the interpretation of the model, they are likely to misapply it and become surprised when the "promises" they mistakenly inferred from the model are not upheld.

With these caveats, however, a formal model can provide many benefits. Because a model rests on a firm mathematical foundation, it is easier for people to evaluate its creator's claims of validity. Because a formal model provides a level of abstraction, it helps the model's users understand details of a complex implementation in terms of a simpler model. Real systems invariably grow. If we wish to extend an *ad hoc* system, we must re-evaluate how the extension interacts with the entire system, to ensure that it does not indirectly introduce security holes or surprises. A formal model can warn us immediately if a proposed extension would nullify prior promises of the model, or it can assure us that the extension fits neatly into the model and introduces no surprising interactions at the level of abstraction of the model.

3.3 Design and qualities of existing systems

In this section, I consider relevant related approaches to resource sharing and security. These fall into two categories: formal models of security, and system architectures that implement security.

3.3.1 Formal models

Formal models for security systems have been designed to prove a wide variety of properties of systems. Some models are designed to prove properties about access to read and write information within a single system. Some models show properties of protocols (in which the parties may be mutually distrusting, or the communications channel may be untrustworthy), and range in detail from simply showing that the communicating parties each know who the opposite party is, to showing that the protocol preserves the secrecy of information.

In this section, I will discuss some relevant security models, and how they relate to our problem of sharing resources across administrative boundaries.

Military security

U.S. military security is concerned with the security of information in a single (albeit large) organization; it applies to all information in the military, whether it is maintained on paper, in computers, or in people's heads. The military security model is concerned not only with accesses to resources, but with the ultimate flow of information. The military uses two axes to specify the clearance of users to see information and the protection category of that information. The vertical axis is called the *sensitivity level* of the information. Military law prohibits the downward flow of sensitive information to users with insufficient clearance; these are called *mandatory access controls*, and military computer systems are required to enforce them. Along the horizontal axis, information belongs to *compartments* that define who has a need to know the information. The military expects individuals to exercise discretion in assigning and enforcing these horizontal restrictions, and military computer systems must assist users in applying *discretionary access control*. Landwehr gives a fine overview of military security [Lan81, pp. 248–250].

The military standard is indeed concerned with allowing appropriate sharing, without unintended transfer, of information resources. This standard, however, is too strong for my purpose. The very notion of military security implies a centralized, top-down administrative structure that defines where information should and should not flow. This dissertation is specifically concerned with the naming and sharing of resources across administrative boundaries, hence concern with administrative enforcement of mandatory controls or restrictions on the flow of information is outside its scope. Note that I may address the restricted problem of access to resources (including information resources) without addressing the larger and more subtle problem of information flow, since information may flow out of one resource into another.

Landwehr discusses several formal models that attempt to capture various aspects of the military security model, including the high-water-mark model, the take-grant model, and information-flow models. The take-grant model is the closest to the one we are interested in, as it directly models delegations of permission from one user to the next.

The access-control matrix

The access-control matrix is a commonly accepted general model intended to capture all possible sharing configurations in a system composed of a collection of subjects (users) and objects (resources). In its generality, it captures all models of sharing, but does not lend much intuition.

> "The access matrix model ... has great flexibility and wide applicability. It is difficult, however, to prove assertions about the protection provided by systems that follow this model without looking in detail at the particular subjects, objects, modes of access, and rules for transforming the access matrix." [Lan81, pp. 256–7]

In summary, just about any access-control problem can be cast into the general access-matrix model, but the resulting model provides little insight into the problem.

Models that ensure secrecy

One class of formal models includes those that can show that a protocol does not leak its secret inputs. Abadi and Gordon's *spi calculus* extends an abstract concurrent language, the *pi calculus*, to include cryptographic primitives. In the spi calculus, one shows that a protocol is secure (has a specific property) by showing that it is isomorphic to a simple protocol that embodies just that property [AG99]. Abadi extends the spi calculus with typing in [Aba97]. Types indicate whether variables hold public or secret information, and rules determine how processes propagate type classifications from inputs to outputs. Gray and Syverson develop a formal system based on modal logic that can prove guarantees about information flow in multilevel-secure systems, including information flowing through covert channels [GS98]. Although useful in evaluating the security of hop-by-hop communications protocols, these models are very detailed, and cannot easily be applied to the transitive sharing I desire to support.

Models that ensure authentication

Members of another class of formal models can show that a communication protocol ensures that its participants arrive at accurate knowledge about those with whom they are communicating; or, more precisely, that they know certain properties about the messages they receive. These protocols may not ensure that specific information remains secret.

Burrows, Abadi, and Needham developed a seminal formal logic for analyzing protocols that has come to be known as "BAN logic" [BAN90]. The logic models principals (active entities engaging in a protocol), encryption keys, and messages traveling over an insecure public channel. Statements in the logic represent the secrecy of keys, the freshness of messages, and principals' beliefs about such statements.

A proof in the logic can show that when honest participants begin a protocol with particular beliefs, they do not arrive at inappropriate beliefs as a result of executing the protocol. An example of an inappropriate belief might be principal A believing that a key is a secret between herself and principal B, when in fact principal C has managed to discover the key. One may uncover a flaw in a protocol by being unable to prove some assertion

about the development of principals' beliefs, or by discovering that such a proof requires strong assumptions that are unlikely to be met in practice.

The BAN paper itself provided a logic (a set of proof rules) but no formal semantics; the work was criticized for the omission. Abadi and Tuttle later remedied the shortcoming by developing a formal semantics [AT91]; in the process, they discovered attractive refinements to the original logic. Bleeker and Meertens define an alternate semantics for BAN logic [BM97].

Syverson argues that for certain security requirements, a logic alone is not likely to produce a positive proof. To verify such properties, he says, we need to use a *model checker* that checks all semantic models and verifies that a given property is valid [Syv93]. Lowe takes just this approach. He uses a model checker to verify that principals in a protocol correctly authenticate one another. The model checker is used to verify a small system, and a hand-developed proof reduces any larger system to the small system [Low96]. Heintze and Tygar describe a model-based approach to verify that protocols are secure [HT96]. Their system can show that the composition of two secure protocols is itself secure.

The Calculus for Access Control is a simpler system than BAN logic, and as such it is not as generally applicable to validating existing protocols [ABLP93]. It has been applied to some systems, however, such as Java stack inspection [WF98]. On the other hand, it offers helpful guidance for generating protocols based on the primitives of the calculus, and can model intricate sharing patterns, including transitivity.

The Calculus for Access Control is primarily concerned with authenticating entities. Lehti and Nikander argue that authorization is the ultimate goal, and that separating authorization into authentication and a second access-control step is both unnecessary and risky [LN98].

Massacci's work on role-based access control defines a semantics for an access-control logic very similar to that of Lampson et al. [Mas97]. It has semantic limitations that allow the use of decisions based on *tableaux methods*. The use of tableaux methods always provides proof of authentication or a counter model that shows that no proof exists. In a distributed system, this latter feature is not as practical as one might hope, since not all relevant statements may be simultaneously available for inspection.

3.3.2 Systems

In this section, I consider architectures that attempt to implement various security models. Some systems implement explicit models such as those in the previous section; other systems approach security in an ad-hoc fashion. Some systems provide a generic substrate that supports a variety of policies. The distributed trusted operating system is a distributed realization of the access-matrix model, designed to enable research into security policies [CL98]. The Flask architecture provides a similarly flexible platform for fine-grained, hierarchical access policies [SSL+99]. The other systems I study here approach security either primarily as an authorization problem or primarily authentication problem.

Authorization

Sollins describes the problem of transitive restricted delegation as "cascaded authentication," and provides a motivating example for its use [Sol88]. She proposes as a solution a

mechanism called *passports*. We will show that passports can be modeled in our system as accreting chains of restricted delegations. Passports do allow a feature we do not support: an integer delegation control field. Because passports authenticate only servers, the implementation requires special-case treatment of channels, which our system models explicitly. Varadharajan et al. propose a more general mechanism that incorporates both symmetric and asymmetric encryption [VAB91].

Neuman's *proxies* are tokens that express delegation; his restricted proxies are analogous to the concept of restricted delegation developed herein [Neu93]. Kerberos version 5 includes support for restricted proxies.

The PolicyMaker system is designed to flexibly describe users' trust requirements, including how they have delegated that trust to others [BFL96]. (The authors use the term "trust" the same way I use "authorization.") Delegations in PolicyMaker may include arbitrary code to check the validity of a delegation; that code has access to the entire access-control decision context in making its decision.

The Simple Public Key Infrastructure is a distributed-system security scheme focused primarily on authorization, and only secondarily on authentication. SPKI certificates implement restricted delegation. In SPKI, every principal is a public key, and is referenced either directly by its key or indirectly by a name defined by another key. SPKI confines itself to reasoning about principals identified by public keys, so it is more applicable in the wide area than on a single machine. Chapter 7 describes how ad-hoc features of SPKI, such as the online revalidation mechanisms, can be cast as applications of the standard components of my system, reducing the complexity of applications that use them.

The CRISIS security architecture is designed to provide a substrate of authentication and authorization services to wide-area applications. It includes features of the Calculus for Access Control such as delegation and roles, as well as a notion of restricted delegation [BVAD98].

These authorization systems are mechanisms with only informally-described semantics. As such, there have no obvious route to generalize their applicability to other situations. For example, they have very specific notions of principal (generally tied to a specific encryption mechanism or authentication server) that do not generalize to support, e.g., channels or compound principals that can describe the authority of cooperating entities.

The Calculus for Access Control mentioned in the previous section was the formal basis for the mechanisms in Taos [LABW92, WABL94]. I borrow the formal foundation of the Calculus for Access Control, and extend it to support my requirement of restricted delegation.

Authentication

The global authentication service of Birrell et al. was a predecessor to that used in Taos; like that system, Birrell's service is hierarchical and hence has a constrained notion of administrative domain [BLNS86].

The goal of a public-key infrastructure is to determine the identity or authority of the holder of a public key. Mione concisely surveys a number of public-key infrastructures [Mio98a, Mio98b].

The `ssh` secure shell utility is designed to provide users with a way to securely authenti-

cate themselves to remote hosts across untrusted networks [Ylo96, YKS+98]. The ssh tools are drop-in replacements for the Unix `rsh` utilities, which were designed to be secure in a trusted network where a system administrator controls what is attached to the wire. The development of ssh was driven by the ubiquitous misuse of `rsh` in untrustworthy networks.

Kerberos is a widely-deployed network authentication system based on encrypted communication with trusted authentication servers. It has a hierarchical notion of administrative domains [NT94]. Lampson et al. model basic Kerberos features in the Calculus for Access Control.

3.4 Design of Snowflake sharing

In this section, I discuss the design of sharing features in the Snowflake prototype. First, I define three terms central to the literature on certificates. A *principal* is an entity that may control a resource and give or receive delegations; common examples are users, machines, and programs. An *issuer* is a principal that initially holds a resource and is delegating the resource to another. A *subject* is the principal that benefits from a delegation. Notice that a single principal can take on either the role of issuer or subject; in fact, because we are concerned with transitive sharing, we expect this case to be common.

To achieve my goal of sharing across administrative boundaries, I need a system focused on the qualities described in Section 3.2. The transitive sharing quality quickly narrows the field; the Calculus for Access Control [LABW92] immediately stands out. I also, however, need to satisfy the restricted delegation quality. The Calculus for Access Control depends on ACLs for restricting delegation, which interferes with the "restriction at every sharing link" requirement.

Therefore, I extend the formal model behind the Calculus for Access Control to support restricted delegation. That extension is the subject of Chapter 4. The new model together with the implementation ideas in [LABW92] combine to suggest a system architecture with the desired qualities.

The authorization systems in Section 3.3.2 have architectures close to that suggested by my formal model. Of those, SPKI's architecture is the closest and best justified by the model. Its goals, however, are different than mine. It is a public-key infrastructure suitable mainly for long-haul networks, but I want a system that is equally applicable locally and remotely to reduce the user cost of sharing. Therefore, Snowflake's architecture incorporates features of the SPKI architecture where convenient, and in fact its implementation derives from a SPKI implementation.

3.4.1 Restricted transitive delegation

Snowflake sharing is centered on the concept of restricted transitive delegation: one user (or other principal) shares a resource with another, possibly restricting what the recipient may do with the shared resource. The resource(s) shared are specified as a subset of all of the resources available to the issuer. So, for example, by giving a subject an unrestricted delegation, the issuer makes the subject as powerful as himself.

3.4.2 Arbitrary principals

SPKI is a public key infrastructure; and in it, every principal is a public key. In Snowflake, principals can be public keys, for example, when they need to be universally recognized, but they may also be all sorts of other entities. A principal may also be an arbitrary symbol whose identity is maintained by an online agent. Such principals are analogous to user IDs in Unix or protected capabilities, and are useful because they admit time- and space-efficient implementations. A principal may also represent a channel as in [LABW92], so that a program can express its confidence in the messages arriving on different channels. More examples of principals appear below.

3.4.3 Groups

One can create a group of principals with a common set of powers by creating a symbolic principal that delegates its authority to each member of a set of other principals. By the transitive property, delegating to the group principal is equivalent to delegating to the members of the group.

3.4.4 Conjunctions

The recipient of a delegation may be the conjunction of several principals, which means that the delegation cannot be used to perform any action unless all of the named recipients agree to the action. My implementation supports SPKI's generalized "threshold subjects," wherein the issuer specifies that k of n subjects must agree to invoke the delegation. In Section 7.4 I show that threshold subjects are formally equivalent to basic conjunction.

3.4.5 Quoting principals

I use quoting principals as described in [LABW92]: a multiplexed gateway, performing trusted actions on behalf of a set of mutually distrusting clients, can intentionally quote the clients to explicitly declare on whose behalf a given action is requested. Each client then delegates authority not to the gateway itself, but to the compound principal "gateway quoting client."

Multiplexed gateways are used for features such as resource aggregation, protocol conversion, proxies, and user interfaces; such a service must pass authority through from the client to the resource implementation, but should not have to make access-control decisions of its own. Quoting allows the server to more easily defer the access-control decision to the ultimate implementation of a resource, which reduces the opportunity for a security flaw in the multiplexed gateway.

3.5 Preview of the sharing chapters

In Part IV I develop the formalism that supports this design. Chapter 4 introduces the formal logic and semantics of restricted delegation. In Chapter 5, I extend the formalism to support the notion of local names originated in SPKI. I develop a separate semantics to clarify the notions of primitive permissions and restriction sets in Chapter 6. Chapter 7

applies these formalisms to model SPKI, and in Chapter 7.9, I propose extensions to SPKI and support or reject them based on the formalisms.

In Part V, I broaden the application of the formalism from SPKI to an implementation of Snowflake's sharing properties. I begin in Chapter 8 by discussing how spanning administrative domains is a special case of a more general problem I call *end-to-end authorization*. Chapters 9 through 11 discuss the infrastructure, channels, and applications I built to implement and demonstrate the Snowflake sharing model. I quantify its performance implications in Chapter 12. Chapter 13 evaluates Snowflake sharing with respect to the qualities outlined in Section 3.2.

Part IV

Sharing Formalism

Chapter 4

The logic and semantics of restricted delegation

In this chapter, I assume that the reader is familiar with the basic operation of modal logic, the Calculus for Access Control, and the semantics that support it, Hughes and Cresswell provide the canonical, concise introduction to modal logic [HC96]. Fagin et al. provide a gentler introduction with motivating examples [FHMV95]. I provide a very brief introduction to modal logic in Appendix B, and an overview of the Calculus for Access Control in Appendix C.

The semantics given in [ABLP93] uses symbols that are conventional in the modal logic community but may surprise those from other backgrounds. The reader should be aware that we use the symbol $\supset$ for logical implication, and the symbol $\Rightarrow$ for the "speaks-for" operator on principals. Table 4.1 summarizes the notation we use for sets in the following sections.

Lampson et al. mention in passing the idea of a qualified speaks-for operator [LABW92, p. 272]. In this section, we introduce our *speaks-for-regarding* operator, which formalizes the notion of the restricted speaks-for operator. It is written $B \overset{T}{\Rightarrow} A$, and read "$B$ speaks for A regarding the set of statements in T." T is any subset of Σ^*. The desired meaning is that when $\sigma \in T$,

$$B \overset{T}{\Rightarrow} A \supset ((B \,\mathbf{says}\, \sigma) \supset (A \,\mathbf{says}\, \sigma))$$

The power of the speaks-for-regarding operator $\overset{T}{\Rightarrow}$ is that A can delegate a subset of its authority *without modifying any ACLs*. Contrast the situation with the use of roles in the Calculus for Access Control (see Appendix C.5), where to delegate authority over a restricted subset of her resources, a user had to define a role and install that role in the ACLs of each resource to be shared.

Restricted speaks-for is transitive:

$$\vdash (\mathcal{C} \overset{T}{\Rightarrow} \mathcal{B}) \wedge (\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A}) \supset (\mathcal{C} \overset{T}{\Rightarrow} \mathcal{A}) \qquad \text{(Axiom E1)}$$

We expect the $\wedge$ operation on principals to be monotonic over $\overset{T}{\Rightarrow}$:

$$\vdash (\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A}) \supset (\mathcal{B} \wedge \mathcal{C}) \overset{T}{\Rightarrow} (\mathcal{A} \wedge \mathcal{C}) \qquad \text{(Axiom E2)}$$

Set	*Example members*	*Description*
Σ	s, t	The set of primitive propositions. They represent resources.
Σ^*	σ, τ $s \wedge t$	The set of well-formed formulas (statements) constructed from Σ, $\wedge$, $\neg$, $\mathcal{A}$ **says**, and $\mathcal{B} \Rightarrow \mathcal{A}$
2^{Σ^*}	S, T, V	The set of sets of statements
P	A, B	The set of primitive principals. They represent agents, including people, machines, programs, and communications channels.
P^*	$\mathcal{A}, \mathcal{B}$ $A \wedge B$	The set of compound principals constructed from P, $\wedge$, $\vert$, and $\cdot N$
$\mathcal{N}$	N	The set of local names

Table 4.1: *The symbols used to represent sets in this dissertation.*

Restricted control over two principals is the same as restricted control over their conjunct:

$$\vdash (\mathcal{C} \stackrel{T}{\Rightarrow} \mathcal{A}) \wedge (\mathcal{C} \stackrel{T}{\Rightarrow} \mathcal{B}) \equiv \mathcal{C} \stackrel{T}{\Rightarrow} (\mathcal{A} \wedge \mathcal{B}) \qquad \text{(Axiom E3)}$$

Let $\mathcal{U}$ be the universe of all well-formed formulas; that is, those formulas over which a model $\mathcal{M}$ defines $\mathcal{E}$. Restricted speaks-for degenerates to the original speaks-for when the restriction set is the set of all statements:

$$\vdash (\mathcal{B} \stackrel{\mathcal{U}}{\Rightarrow} \mathcal{A}) \equiv (\mathcal{B} \Rightarrow \mathcal{A}) \qquad \text{(Axiom E4)}$$

If Bob speaks for Alice regarding a set of statements T, he surely speaks for her regarding a subset $T' \subseteq T$:

$$\forall T' \subseteq T,$$
$$\vdash (\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{A}) \supset (\mathcal{B} \stackrel{T'}{\Rightarrow} \mathcal{A}) \qquad \text{(Axiom E5)}$$

Using Axiom E5, a chain of delegations can be collapsed to a single delegation, connecting the head principal in the chain to the tail, whose restriction set is the intersection of the restriction sets of each of the original delegations.

$$\vdash (\mathcal{C} \stackrel{S}{\Rightarrow} \mathcal{B}) \wedge (\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{A}) \supset (\mathcal{C} \stackrel{S \cap T}{\Rightarrow} \mathcal{A}) \qquad \text{(Theorem E6)}$$

This is not to say that $\mathcal{C}$ may not speak for $\mathcal{A}$ regarding more statements than those in the intersection; we address this topic further in Section 7.9.

If we have two restricted delegations from Alice to Bob, we might expect Alice to speak for Bob with respect to the union of the restriction sets. Because of the semantics we choose for $\stackrel{T}{\Rightarrow}$, however, this intuition does not hold.

$$(B \stackrel{S}{\Rightarrow} A) \wedge (B \stackrel{T}{\Rightarrow} A) \not\equiv B \stackrel{S \cup T}{\Rightarrow} A \qquad \text{(Result E7)}$$

In Section 4.1.1, we describe a relation weaker than $\stackrel{T}{\Rightarrow}$ for which the intuitive statement holds.

The quoting operator on principals ($|$) is monotonic in both arguments over $\Rightarrow$. Quoting is still monotonic over $\stackrel{T}{\Rightarrow}$ in its left argument:

$$\vdash \left(\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{A}\right) \supset \mathcal{C}|\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{C}|\mathcal{A} \qquad \text{(Axiom E8)}$$

Our semantics does not justify monotonicity in the right argument, however:

$$\left(\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{A}\right) \not\supset \mathcal{B}|\mathcal{C} \stackrel{T}{\Rightarrow} \mathcal{A}|\mathcal{C} \qquad \text{(Result E9)}$$

This result appears to limit the usefulness of quoting. The same counterexample that shows Result E9 shows the same property for the weak speaks-for relation defined in Section 4.1.1, so it seems that the notion of quoting simply does not mix easily with restricted delegation.

We can, however, propagate the quoted principal through the restriction set. Let T^* be the closure of T with respect to the propositional operators $\neg$ and $\wedge$: $T \subseteq T^*$, and if $\sigma, \tau \in T^*$, then $\neg\sigma \in T^*$ and $\sigma \wedge \tau \in T^*$. Furthermore let TC be the closure of T with respect to the modal operator $\mathcal{C}$ **says**: $T \subseteq TC$, and if $\sigma \in TC$, then $(\mathcal{C}\,\mathbf{says}\,\sigma) \in TC$. Now $(T^*)C$ is the modal closure applied to the propositional closure of some original set T. With these definitions, we can justify this axiom:

$$\vdash \left(\mathcal{B} \stackrel{(T^*)C}{\Rightarrow} \mathcal{A}\right) \supset \left(\mathcal{B}|\mathcal{C} \stackrel{T}{\Rightarrow} \mathcal{A}|\mathcal{C}\right) \qquad \text{(Axiom E10)}$$

When $T = \mathcal{U}$, this axiom reduces to showing right-monotonicity for the original speaks-for relation. This axiom means that $\mathcal{A}$'s restricted delegation to $\mathcal{B}$ must explicitly include any "quotes" of $\mathcal{C}$ about which it is willing to believe $\mathcal{B}$. It seems awkward, but it is a useful result. Why? Because in any possible-worlds semantics wherein $(\mathcal{B} \stackrel{T}{\Rightarrow} \mathcal{A}) \supset (\mathcal{B}|\mathcal{C} \stackrel{T}{\Rightarrow} \mathcal{A}|\mathcal{C})$ for *all* principals $\mathcal{C}$, the relation representing $\mathcal{A}$ depends on every other principal relation. The introduction of malicious principals with cleverly-chosen relations into such a system can effectively expand T until $T = \mathcal{U}$.

4.1 Semantics of $\stackrel{T}{\Rightarrow}$

We use a semantics based on possible worlds, modeling a system with a *model* $\mathcal{M} = \langle W, w_0, I, J\rangle$ whose components are defined as in [ABLP93]. The semantic definition of $\stackrel{T}{\Rightarrow}$ is based on the notion of *projecting* a model into a space where only the statements in set T are relevant. The idea behind this definition is that if one were to take the "quotient" of a model M with respect to the dual of T, the resulting model $\overline{M}$ would be concerned only with statements in T. $B \Rightarrow A$ in $\overline{M}$ should be equivalent to $B \stackrel{T}{\Rightarrow} A$ in the original model. The model $\overline{M}$ is a projection of M that only preserves information about statements in T.

We begin the construction by defining an equivalence relation $\cong_T$: $W \times W$ that relates two worlds whenever they agree on all statements in T:

$$w \cong_T w' \ \textbf{iff}\ \left(\forall \sigma \in T, w \in \mathcal{E}(\sigma)\ \textbf{iff}\ w' \in \mathcal{E}(\sigma)\right) \qquad \text{(Definition E11)}$$

Then we define the mapping $\phi_T : W \to \overline{W}$ that takes worlds from the original model to equivalence classes under $\cong_T$:

$$\phi_T(w) = \phi_T(w') \textbf{ iff } w \cong_T w' \qquad \text{(Definition E12)}$$

The equivalence classes belong to a set $\overline{W} = 2^T$; notice that worlds (equivalence class representatives) in $\overline{M}$ cannot be confused with those in M. We give a construction of $\phi_T(w)$ in Appendix Section A.1.

Next we extend ϕ_T to the function $\phi_T^w : 2^W \to 2^{\overline{W}}$ that maps a set of worlds $S_w \subseteq W$ to a set of equivalence class representatives in the projected model:

$$\phi_T^w(S_w) = \{\overline{w} \mid \exists w \in S_w,\ \overline{w} = \phi_T(w)\} \qquad \text{(Definition E13)}$$

We use bar notation ($\overline{w}$) to indicate an equivalence class representative (member of a world of a projected model) as opposed to a member of W in the original model.

We can now give our semantic definition of restricted delegation:

$$\mathcal{E}(\mathcal{B} \overset{T}{\Rightarrow} A) = \begin{cases} W & \text{if } \forall w_0 \left(\begin{array}{r} \phi_T^w(\mathcal{R}(\mathcal{A})(w_0)) \subseteq \\ \phi_T^w(\mathcal{R}(\mathcal{B})(w_0)) \end{array} \right) \\ \varnothing & \text{otherwise} \end{cases} \qquad \text{(Definition E14)}$$

For the justifications of several of the axioms it is more convenient to shift the projection (ϕ) operation to one side of the subset relation. To do so, we define

$$\phi_T^+(R) = \left\{ \langle w_0, w_1' \rangle \;\middle|\; \exists w_1 \cong_T w_1',\ \langle w_0, w_1 \rangle \in R \right\} \qquad \text{(Definition E15)}$$

Think of ϕ_T^+ as a function that introduces as many edges as it can to a relation without disturbing its projection under T.

We can use ϕ_T^+ to give an equivalent definition of $\overset{T}{\Rightarrow}$:

$$\mathcal{E}(\mathcal{B} \overset{T}{\Rightarrow} A) = \begin{cases} W & \text{if } \mathcal{R}(\mathcal{A}) \subseteq \phi_T^+(\mathcal{R}(\mathcal{B})) \\ \varnothing & \text{otherwise} \end{cases} \qquad \text{(Definition E16)}$$

The symbolic gymnastics of moving the projection to the right side of the $\subseteq$ relation is equivalent to the definition in terms of ϕ_T^w, but it makes some of the proofs more concise. The equivalence is shown in Appendix A.2.

A casual intuition for this definition is that ϕ_T projects from the full model M down to a model in which worlds are only distinguished if they differ with regard to the truth of statements in T. If we collapse away the accessibility arrows that do not say anything about what is happening in T, and A's relation is a subset of B's relation in the projection, then A knows everything B knows about statements in T. This intuition is exactly what we want for restricted delegation.

What happens if we take an alternative semantic definition for restricted delegation? We explore one seemingly-natural but undesirable alternative in Appendix A.3. Presently, in Sections 4.1.1 and 4.1.2, we explore two intriguing possibilities.

4.1.1 The $\overset{T}{\rightarrow}$ relation

Abadi mentions a weaker version of the speaks-for operator, $B \rightarrow A$,[1] that is true exactly when $B\,\mathbf{says}\,\sigma \supset A\,\mathbf{says}\,\sigma$. Note however that A may have some different reason than B to say σ. Semantically, A may consider a totally different set of worlds possible; it just happens that σ is still true in those worlds. For that reason, $B \Rightarrow A$ is a stronger relation than $B \rightarrow A$. The former requires a special (subset) relationship to appear in the model.

We can define a semantics for these operators.

$$\mathcal{E}(B \rightarrow A) = \left\{ w \;\middle|\; \begin{array}{l} \forall \sigma \in \Sigma^*, \\ \mathcal{R}(B)(w) \subseteq \mathcal{E}(\sigma) \supset \mathcal{R}(A)(w) \subseteq \mathcal{E}(\sigma) \end{array} \right\}$$

$$\mathcal{E}(B \overset{T}{\rightarrow} A) = \left\{ w \;\middle|\; \begin{array}{l} \forall \sigma \in T, \\ \mathcal{R}(B)(w) \subseteq \mathcal{E}(\sigma) \supset \mathcal{R}(A)(w) \subseteq \mathcal{E}(\sigma) \end{array} \right\}$$

These are not particularly exciting definitions; they simply state just what the axiom says:

$$(B \rightarrow A) = (B\,\mathbf{says}\,\sigma \supset A\,\mathbf{says}\,\sigma)$$

It is obvious that weak restricted speaks-for degenerates into Abadi's unrestricted $\rightarrow$ operator, since $\mathcal{U} = \Sigma^*$:

$$B \overset{\mathcal{U}}{\rightarrow} A \equiv B \rightarrow A$$

4.1.2 The $\overset{T}{\Rightarrow}$ relation

Having witnessed a weaker relation $\rightarrow$, one may wonder why Abadi et al. preferred a definition of speaks-for ($\Rightarrow$) that was stronger than it needed to be. The intuition seems to be that the stronger semantics captures the fact that A understands B's *reasons* for believing various statements. Our $\overset{T}{\Rightarrow}$ is "strong" in the same sense, and it degenerates to $\Rightarrow$ when $T = \mathcal{U}$.

Our first attempt to build a strong speaks-for relation, however, ended up too strong. We call the definition below our "*mighty*" speaks-for:

$$\mathcal{E}(B \overset{T}{\Rightarrow} A) = \left\{ w \;\middle|\; \forall \sigma,\ \mathcal{R}(A)(w) - \bigcap_{\sigma \in T} \mathcal{E}(\sigma) \subseteq \mathcal{R}(B)(w) \right\}$$

The idea here is that if a subset relationship restricts A to say everything B says, then permitting A's relation to grow a little permits A to not say some of the statements B says

[1] Read "B weakly speaks for A."

(those not in T). The definition above preserves the desired relationship: if B **says** σ, every world B can see has σ true; because A can only see those worlds and other worlds where σ is true, A **says** σ. So A has no "reasons" (edges to possible worlds) to not say σ that B does not also have.

The $\overset{T}{\Rrightarrow}$ relation seemed promising because it degenerates into $\Rightarrow$ when $T = \mathcal{U}$. It appears too strong, however. For example, if $T = \{s, \neg s\}$, then $\cap_T \mathcal{E}(s) = \varnothing$, so any $B \overset{T}{\Rrightarrow} A$ would imply $B \Rightarrow A$. The semantic requirements are so strong that many interesting choices of T are not meaningful. In fact, we present $\overset{T}{\Rrightarrow}$ here precisely because it emphasizes the importance of having a satisfying semantics to lend intuitive meaning to the logic. We explore this issue further in Section 7.9.

4.1.3 Relationships among the relations

Both $\overset{T}{\Rightarrow}$ and $\overset{T}{\Rrightarrow}$ are strictly stronger than $\overset{T}{\rightarrow}$:

$$B \overset{T}{\Rightarrow} A \supseteq B \overset{T}{\rightarrow} A$$
$$B \overset{T}{\rightarrow} A \not\supseteq B \overset{T}{\Rightarrow} A$$
$$B \overset{T}{\Rrightarrow} A \supseteq B \overset{T}{\rightarrow} A$$
$$B \overset{T}{\rightarrow} A \not\supseteq B \overset{T}{\Rrightarrow} A$$

The $\overset{T}{\Rightarrow}$ relation is certainly not stronger than $\overset{T}{\Rrightarrow}$; it seems that it should be strictly weaker, but a corner case prevents it from being so:

$$B \overset{T}{\Rightarrow} A \not\supseteq B \overset{T}{\Rrightarrow} A$$
$$B \overset{T}{\Rrightarrow} A \not\supseteq B \overset{T}{\Rightarrow} A$$

We establish each of these relationships in Appendix A.5.

We introduced the weak and mighty speaks-for relations to demonstrate the consequences of the choice of formal models. Like Abadi et al., we adopt the $\overset{T}{\Rightarrow}$ version of the relation.

4.2 Additional benefits of $\overset{T}{\Rightarrow}$

Introducing the $\overset{T}{\Rightarrow}$ operator to the logic not only provides the important feature of restricted delegation, but it simplifies the logic by replacing the *controls* operator, replacing roles, and providing a formal mechanism for the treatment of expiration times.

4.2.1 Supplanting *controls*

Now that we have the restricted speaks-for relation, we can dispense with the special *controls* operator for building ACLs.

Consider Abadi et al.'s special identity principal **1** (see Appendix C.5.1). Because it believes only truth, $(\mathbf{1}\,\mathbf{says}\,s) \supset s$ for all statements s. That is, there is an implicit principal that controls all statements. We can replace every statement of the form A *controls* s with an equivalent one: $A \overset{\{s\}}{\Rightarrow}_{w_0} \mathbf{1}$. This statement ensures that if $A\,\mathbf{says}\,s$, then at the actual world w_0 of the model, $\mathbf{1}\,\mathbf{says}\,s$. Since the **1** relation only contains edges from a node to itself, this condition can only be satisfied by selecting an actual world w_0 where s is true.

4.2.2 Supplanting roles

Roles as originally defined are attractive, but they have the significant difficulty that introducing a new restricted role R_2 involves finding all of the objects that role should be allowed to touch, and adding A **as** R_2 to each of those ACLs. When one of those objects does not allow ACL modifications by A, it is impossible for A to express the desired new role. The SPKI document gives a vivid example that shows how ACL management can become unwieldy [EFL+99, p. 17].

With the speaks-for-regarding relation, A can introduce a new role R_2 for itself by allowing $(A\,\mathbf{as}\,R_2) \overset{T_2}{\Rightarrow} A$. In fact, roles are no longer necessary at all, but the **as** and **for** operator, or operators like them, may still be useful for building tractable implementations.

Roles, as semantically defined by Abadi et al., can also have surprising consequences because they belong to a global "namespace." Imagine that both Alice and Bob use the role R_{user} in their ACLs. That means that the same relation $\mathcal{R}(R_{\text{user}})$ encodes both the way that $A\,\mathbf{as}\,R_{\text{user}}$ is weaker than A, and the way that $B\,\mathbf{as}\,R_{\text{user}}$ is weaker than B.

Let us build a model for a concrete example. Our model has two worlds, w_s and $w_{\overline{s}}$, where s is true and s is false, respectively. Assume that neither Alice nor Bob begin by believing s: $\neg A\,\mathbf{says}\,s$ and $\neg B\,\mathbf{says}\,s$. Our model must have the relations:

$$\langle w_0, w_{\overline{s}} \rangle \in \mathcal{R}(A)$$
$$\langle w_0, w_{\overline{s}} \rangle \in \mathcal{R}(B)$$

(w_0 is a placeholder for whichever world is the actual world.) Now assume Alice's doppelganger $(A\,\mathbf{as}\,R_{\text{user}})\,\mathbf{says}\,s$. To model this, we need $\mathcal{R}(A\,\mathbf{as}\,R_{\text{user}}) = \mathcal{R}(A) \circ \mathcal{R}(R_{\text{user}})$ to include only worlds where s is true. We want to preserve $\neg A\,\mathbf{says}\,s$, or else it would be the case that $(A\,\mathbf{as}\,R_{\text{user}}) \Rightarrow A$. That means we cannot change A's relation; so our only recourse is to use $\mathcal{R}(R_{\text{user}})$ to sever the edges leading to $w_{\overline{s}}$:

$$\langle w_{\overline{s}}, w_{\overline{s}} \rangle \notin \mathcal{R}(R_{\text{user}})$$

But because R_{user} is also used by Bob, we arrive at:

$$w_{\overline{s}} \notin (\mathcal{R}(B) \circ \mathcal{R}(R_{\text{user}}))(w_0)$$
$$w_{\overline{s}} \notin \mathcal{R}(B\,\mathbf{as}\,R_{\text{user}})(w_0)$$

Since $B\,\mathbf{as}\,R_{\text{user}}$ has no edges to the world $w_{\overline{s}}$ where s is false, the model supports the statement $(B\mathbf{as}R_{\text{user}})\,\mathbf{says}\,s$. Using a common role has caused unexpected crosstalk between one principal and another.

4.2.3 Formalizing statement expiration

Lampson et al. treat expiration times casually in [LABW92, p. 270]: "Each premise has a *lifetime*, and the lifetime of the conclusion, and therefore of the credentials, is the lifetime of the shortest-lived premise." It is likely that a formal treatment of lifetimes would be time-consuming and unsurprising, but the lifetimes are an unsightly element glued onto an otherwise elegant logical framework. Fortunately, the $\overset{T}{\Rightarrow}$ relation allows us to dispense with lifetimes.

Consider that primitive statements such as s are meant to encode some operation in a real system (see Appendix C.3). Assume that each s describes not only an operation, but the effective time the operation is to take place.[2] Further, assume a restriction set T in a delegation $B \overset{T}{\Rightarrow} A$ includes restrictions on the times of the operations under consideration. After the last time allowed by the set, the delegation remains logically valid, but becomes useless in practice. Furthermore, restrictions on T can be more than expiration times; one can encode arbitrary temporal restrictions, such as only allowing a delegation to be valid on Friday afternoons.

[2]Like Lampson et al., we ignore the issue of securely providing loosely synchronized clocks.

Chapter 5

The semantics of SPKI names

Recall from Section 4.2.2 how roles share a global "namespace," and the danger of crosstalk between applications of the same role. SPKI names are promising, but they have the same property: identical names have different meaning depending on the "scope" in which they appear. To model names, we need to extend our logic and semantics.

We introduce to the logic a new set of primitive *names*, $\mathcal{N}$. We also extend principal expressions to include those of the form $P \cdot N$, where P is an arbitrary principal expression and $N \in \mathcal{N}$. $P \cdot N$ is read "P's N." Because $\cdot$ only accepts a principal as its left argument, there is no ambiguity in the order of operations; $P \cdot N_1 \cdot N_2$ can only be parenthesized $(P \cdot N_1) \cdot N_2$.

5.1 The logic of names

What properties do we want names to have?

Local namespaces. First, a principal should control the meaning of any names defined relative to itself:

$$\forall \text{ principals } \mathcal{A}, \text{ names } N :$$
$$(\mathcal{A} \,\mathbf{says}\, (\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A} \cdot N)) \supset (\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A} \cdot N)$$

We do not take this statement as an axiom for the same reason that Abadi, Lampson et al. do not accept the handoff axiom [LABW92, p. 715], [ABLP93, p. 273]. In particular, our semantics does not support it. Instead, as with the handoff axiom, we allow the implementation to assume appropriate instances of it.

Left-monotonicity. Second, name application should be monotonic over speaks-for. If Alice binds her name "barber" to Bob, and Bob binds his name "butcher" to Charlie, then we want "Alice's barber's butcher" to be bound to Charlie.

$$(\mathcal{B} \Rightarrow \mathcal{A}) \supset (\mathcal{B} \cdot N \Rightarrow \mathcal{A} \cdot N) \qquad \text{(Axiom E17)}$$

Using this rule, we can write the following to capture the desired intuition:

$$(B \Rightarrow A \cdot N_{\text{barber}}) \supset$$
$$B \cdot N_{\text{butcher}} \Rightarrow A \cdot N_{\text{barber}} \cdot N_{\text{butcher}}$$

Distributivity. We combine the following pair of results

$$(\mathcal{A} \wedge \mathcal{B}) \cdot N \Rightarrow (\mathcal{A} \cdot N) \wedge (\mathcal{B} \cdot N) \qquad \text{(Theorem E18)}$$
$$(\mathcal{A} \cdot N) \wedge (\mathcal{B} \cdot N) \Rightarrow (\mathcal{A} \wedge \mathcal{B}) \cdot N \qquad \text{(Axiom E19)}$$

to show that names distribute over principal conjunction:

$$(\mathcal{A} \wedge \mathcal{B}) \cdot N = (\mathcal{A} \cdot N) \wedge (\mathcal{B} \cdot N) \qquad \text{(Theorem E20)}$$

Here is a motivating example: If Alice has two doctors Ed (E) and Fred (F), and Bob visits doctors Fred and George (G), then who is "(Alice and Bob)'s doctor?" Fred is the only person who serves as both people's doctor.

No quoting axiom. The principal $(A|B) \cdot N$ can be written, but we have yet to find a meaningful intuitive interpretation for it. $(A|B) \cdot N$ bears no obvious relation to $(A \cdot N)|(B \cdot N)$, for example. We allow the principal in our logic, but we have no axioms for extracting quoting from inside a name application.

Nonidempotence. Finally, application of names should not be always idempotent. Unless some other speaks-for statement causes it, there is no reason that "Bob's barber's barber" should speak for "Bob's barber." We were initially tempted to model name application $(\cdot)$ with role application, because roles satisfy Axiom E17; however, roles are idempotent.

It may be the case, though, that the application of a name can become idempotent. Take the example in Figure 5.1. In this example, let the symbol N stand for the name "barber."

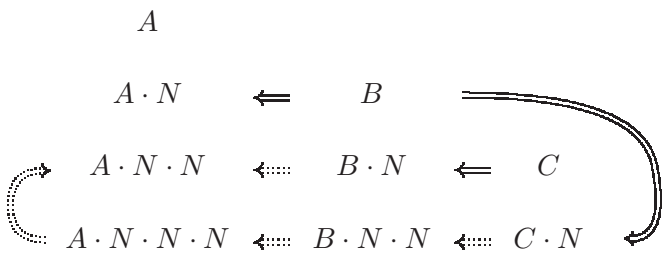


Figure 5.1: *An example that shows when inherited names can become idempotent. Each arrow represents a speaks-for relationship; the text explains each arrow in more detail.*

The upper solid left arrow represents an explicit statement made by Alice: $A\,\mathbf{says}\,B \Rightarrow (A \cdot N)$; that is, Bob may serve as Alice's barber, and do anything "Alice's barber" is allowed to do. Similarly, the other solid left arrow represents Bob delegating Charlie as his barber. It turns out Charlie and Bob work in the same barber shop and cut each other's hair. The swooping solid arrow on the right represents Charlie delegating the responsibility of "Charlie's barber" to Bob. So $A \cdot N$ is "Alice's barber," and is controlled by her barber Bob. $A \cdot N \cdot N$ is "Alice's barber's barber," and is controlled by her barber's barber Charlie. Bob also has some control over "Alice's barber's barber," since he is free to change his barber from Charlie to another.

An interesting thing happens at the next level of name application. The literal name "Alice's barber's barber's barber," who we know as Bob, is actually *equal* to "Alice's barber's barber." It is not that Bob becomes equal to Charlie, but that $\cdot N$ has become idempotent. In our case, both Charlie and Bob have control over $A \cdot N \cdot N$, so any further application of $\cdot N$ introduces no new restrictions on the resulting principal. The derived principal is equal to the parent principal. This conclusion is both intuitive, and valid in the semantics we present next, but cannot be proven using the logic.[1]

5.2 The semantics of names

We mentioned above that names and name application cannot be modeled with the roles and the quoting operator, because quoting a role is always idempotent. Furthermore, using the same role for multiple uses of the same name by different principals introduces crosstalk as described in Section 4.2.2.

Instead, we model names as follows. First, we add a new element K to the tuple that defines a model. A model with naming consists of:

$$\mathcal{M} = \langle W, w_0, I, J, K \rangle$$

The new interpretation function $K : P \times \mathcal{N} \rightarrow 2^{W \times W}$ maps a primitive principal A and a name N to a relation. The idea is that principals only define the first level of names in their namespaces; all other names are consequences of chained first-level name definitions.

Next we extend $\mathcal{R}$ to define the relations for principals formed through name application. We want to define $\mathcal{R}(\mathcal{A} \cdot N)$ as the intersection of several other sets, each requirement ensuring a desired property. Our definition, however, would end up circular (at requirement I, with equal principals) if it were expressed in terms of set intersection. Instead, we define $\mathcal{R}(\mathcal{A} \cdot N)$ as the largest relation (subset of $2^{W \times W}$) satisfying all of the following requirements:

$$\mathcal{R}(\mathcal{A} \cdot N) \subseteq \mathcal{R}(\mathcal{B} \cdot N) \qquad \text{(I)}$$
$$(\forall \mathcal{B} : \mathcal{R}(\mathcal{A}) \subseteq \mathcal{R}(\mathcal{B}))$$
$$\mathcal{R}(\mathcal{A} \cdot N) \subseteq K(\mathcal{A}, N) \qquad \text{(II)}$$
$$(\text{when } \mathcal{A} \in P)$$
$$\mathcal{R}(\mathcal{A} \cdot N) \subseteq \mathcal{R}(\mathcal{B} \cdot N) \cup \mathcal{R}(\mathcal{C} \cdot N) \qquad \text{(III)}$$
$$(\text{when } \mathcal{A} = \mathcal{B} \wedge \mathcal{C})$$
$$\text{(Definition E21)}$$

Requirement I supports Axiom E17. Requirement II applies only to primitive principals, and allows each primitive principal to introduce definitions for first-level names in that principal's namespace. A system implementing instances of the handoff rule does so conceptually by modifying $K(\mathcal{A}, N)$. Requirement III only applies to principal expressions that are conjunctions, and justifies Theorem E20.

[1] We accept incompleteness. Abadi et al. mention that the original logic was incomplete. The $\stackrel{T}{\Rightarrow}$ relation certainly does not help matters, considering that completeness would involve introducing finite-set mathematics to the logic.

There is no question some such largest relation exists. Since each requirement is a subset relation, at least the empty set satisfies all three. There is an upper bound, since every relation is a subset of the finite set $W \times W$. Finally, the largest relation must be unique. For if there were two such relations, then any element in one must belong to the other, since it belongs to every set on the right-hand side of a subset relation in the requirements, and I arrive at a contradiction.

5.3 Challenges in the name semantics

In our semantics, as in Abadi's, left-monotonicity (Axiom E17) turns out to be surprisingly powerful. In this section we consider how to temper it. We also consider a stronger version of left-monotonicity, generalized to the restricted-speaks-for relation, and discuss why it is undesirable.

The semantics we have chosen here have subtle consequences. Axiom E17 is a dangerous axiom, because it lets principals modify other principal relations without limiting their own. For example, if $B = W \times W$, then B speaks for every principal A, since $\mathcal{R}(A) \subseteq \mathcal{R}(B)$. Now if B assigns a name in its namespace ($\Box_B C \Rightarrow B \cdot N$), we must modify our model to ensure that $\mathcal{R}(B \cdot N) \subseteq \mathcal{R}(C)$. Because $B \cdot N$ is a different principal than B, with an independent relation, $\mathcal{R}(B)$ is not affected. But requirement I asserts that all principals that B speaks for inherit B's names; therefore we can conclude that $C \Rightarrow A \cdot N$ for *every principal* A. This result is not satisfying; the semantics has failed to capture the notion that A does not mean to give its power away to any principal B silent enough to speak for A, but only to principals that it somehow delegates to "directly."

One way to repair this problem is to ensure that B's relation is weakened whenever it says something about its local namespace, so that only a principal "intentionally" inheriting B's other beliefs will inherit B's name bindings as well. For example, suppose that when $\Box_B C \Rightarrow B \cdot N$, we also assume that there is some statement s that represents the idea $C \Rightarrow B \cdot N$, and that $\Box_B s$. Now, unless another principal A specifically allows B to speak for it (including about the special statement s), requirement I does not cause A to inherit B's name binding for N.

We are not especially satisfied with the intuitive value of this repair, and we continue to search for a more satisfying semantics for names that avoids the pitfalls we have identified in our own semantics and in Abadi's.

One may also wonder why we restrict the condition of requirement I to the complete speaks-for relation, and not include the restricted speaks-for relation:

$$\mathcal{R}(\mathcal{A} \cdot N) \subseteq \phi_T^+(\mathcal{B} \cdot N) \qquad (\forall \mathcal{B} : \mathcal{R}(\mathcal{A}) \subseteq \phi_T^+(\mathcal{R}(\mathcal{B})))$$

This proposal would mean that if B has limited power over A, then $B \cdot N$ should have the same power over $A \cdot N$. There are two reasons we avoid this definition.

First, the result is not intuitive. If Bob speaks for Alice with regard to stock trading (Bob is Alice's broker), should Bob's barber speak for Alice's barber with regard to stock trading? It may be a moot question, since no one is likely to delegate permission over stock trading to the symbolic name "Alice's barber," but in any case the proposal seems unnatural.

Abadi's notation	*Our notation*
S	Σ
$\mu : S \times \mathcal{W} \to \{true, false\}$	$I : \Sigma \to 2^W$
$\rho : N \times \mathcal{W} \to 2^{\mathcal{W}}$	$K : P \times \mathcal{N} \to 2^{W \times W}$
$a \in \mathcal{W}$	$w \in W$
principals p, q	$\mathcal{A}, \mathcal{B} \in P^*$
$n \in N$	$N \in \mathcal{N}$
$[\![n]\!]_a = \rho(n, a)$	$\mathcal{R}(\mathcal{A} \cdot N)(w) = K(\mathcal{A}, N)(w)$
$[\![p'\mathtt{s}\ n]\!]_a$	$\mathcal{R}(\mathcal{A} \cdot N)(w)$

Table 5.1: *A guide to translating between Abadi's notation and ours*

Second, the semantics suggest that the proposal is troublesome. If we replace requirement I with the proposal, suddenly almost every principal, by its choice of local name bindings, has some degree of control over every other principal's names. One can define a function that, given two relations $\mathcal{R}(\mathcal{A})$ and $\mathcal{R}(\mathcal{B})$, returns a subset of $\mathbf{1}$ which when composed with $\mathcal{R}(\mathcal{B} \cdot N)$, produces an the upper bound for $\mathcal{R}(\mathcal{A} \cdot N)$ by considering *every* choice of T. It turns out that this upper-bound usually ends up surprisingly small, giving barely-related principals a high degree of control over one another's names. Furthermore, the repair we suggest above (introducing a special statement s to capture intentional delegations) does not help, because the upper-bound function considers every T, including those that disregard s.

I initially thought it arbitrary that SPKI allowed restricted authorizations but only unrestricted name bindings. In the light of my current semantics, however, I cannot justify extending name bindings to allow restriction.

5.4 Abadi's semantics for linked local namespaces

Abadi gives an alternate logic and semantics for SPKI-style linked local namespaces [Aba98]. (He refers to SDSI, from which SPKI 2.0 derives.) Abadi's notation diverges from that used in [ABLP93], but the semantics are the same. Table 5.1 helps translate the notation. Our semantics differs in three interesting ways.

First, SPKI has special global names, so that if N_G is a global name, $\mathcal{A} \cdot N_G = N_G$. The result is that the same syntactic construct can be used to bind a local name to another local name or to a globally-specified name. All names in linking statements are implicitly prefixed by the name of the speaking principal; but if the explicitly mentioned name is global, the prefix has no consequence. We consider this syntactic sugar, and leave it to an implementation to determine from explicit cues (such as a key specification or a SDSI name that ends in !!) whether a mentioned principal should be interpreted as local to the speaker.

Second, Abadi's logic adopts the handoff rule for names, which he calls the "Linking" axiom Here it is, translated to our terminology:

$$\mathcal{A}\,\mathbf{says}\,(\mathcal{B} \Rightarrow (\mathcal{A} \cdot N)) \supseteq (\mathcal{B} \Rightarrow (\mathcal{A} \cdot N))$$

He validates the axiom by the use of composition to model name application, with which we disagree.

The third and most important way our semantics differs from Abadi's is that Abadi's semantics models name application as quoting (composition). Each unqualified (local) name is mapped to a single relation. This property can introduce crosstalk between otherwise unconnected principals; recall the example from Section 4.2.2. Even when a name relation is not constrained to be a role, the same problem arises. For example, let N represent the name "doctor." Imagine that Bob assigns Charlie to be his doctor: $C \Rightarrow B|N$. This is fine; Charlie should be able to do some things on Bob's behalf, but not everything: If $B|N \stackrel{T}{\Rightarrow} B$, then Charlie can do the things in T.

Enter Alice, who is not only omniscient ($A = \mathbf{1}$), but serves as her own doctor ($A \Rightarrow A|N$). Abadi's semantics requires that $\mathcal{R}(\mathbf{1}) \circ \mathcal{R}(N) \subseteq \mathcal{R}(\mathbf{1})$. At worst, $\mathcal{R}(N) = \mathcal{R}(\mathbf{1})$, causing $B|N = B$, enabling Charlie's doctor to make investment decisions on Charlie's behalf. At best, $\mathcal{R}(N) \subset \mathcal{R}(\mathbf{1})$, and $B|N$ begins spouting off random statements, some of which may be in T, making Bob believe random statements.[2]

Our semantics escapes this fate by assigning to each use of a name its own relation, then ensuring the correct subset relationships remain among those relations. We must admit that our semantics for names is at best opaque. Although using an existential definition like "largest set satisfying the requirements" is not illuminating, we feel it is better than the alternative.

[2]Specifically, those statements in T where $\phi_T(\mathcal{R}(B|N)) = \varnothing$.

Chapter 6

The semantics of authorization tag notation

An important part of SPKI is a user-defined `<tag>` object that describes either a specific request to be verified, or a set of permissions granted in a delegation. SPKI defines how tags are to be intersected, which gives the user some idea about the meaning of tags. In this chapter, I derive a formal language for tags and show that tags are (almost) closed under intersection. I also show that tags have a desirable property that I depend upon for our formalization of SPKI's properties in Section 7.9.

6.1 Overview

One confusing aspect of tags is that they serve to represent both single requests and delegation restrictions (sets of requests to be permitted). How do they do this? The short answer is containment. Most tags represent an infinite set of finite strings. Let us call those strings *powers*. A request to be verified is a set of powers; the intuition is that executing the request requires that the requester have at least a certain set of privileges. A permission describes another set of powers. If the permission's set contains the request's set, then the permission grants at least each of the powers required for the request.

Why use infinite sets of powers? For extensibility. In general, a permission (tag) is represented by an infinite set of powers. Therefore, it can always be further subdivided into more-specific permissions, each still represented by an infinite subset of the original permission's powers.

Given this motivation for the structure of tags, I construct tags from the ground up using grammars. I begin by defining bytestrings and the atomic finite strings I have called powers.

6.2 Bytestrings and powers

Let Σ be the natural alphabet of our system; in our case, let it be the set of 256 octets. Let $\mathbf{B}$ be the set of all finite-length strings of octets, Σ^*:

$$\begin{aligned} \mathbf{B} &:= \sigma\mathbf{B} && (\forall \sigma \in \Sigma) \\ &\mid \varepsilon \end{aligned}$$

This is the set SPKI calls "bytestrings." Next we extend Σ with three metasymbols to unambiguously demarcate the list structure of a power: $\Sigma' = \Sigma \cup \{$ (, ␣ ,) $\}$. Now we define the mutually-recursive sets of expressions ($\mathbf{E}$), lists of expressions ($\mathbf{L}$), and non-empty lists of expressions ($\mathbf{N}$):

$$\begin{aligned} \mathbf{E} &:= \mathbf{B} \\ &\mid (\,\mathbf{L}\,) \\ \mathbf{L} &:= \mathbf{N} \\ &\mid \varepsilon \\ \mathbf{N} &:= \mathbf{E}\text{␣}\,\mathbf{N} \\ &\mid \mathbf{E} \end{aligned}$$

The gymnastics with the non-empty lists serve to prevent lists from ending with a ␣ just before the); this feels right but it is not important for the development of the semantics.

We call the set of expressions $\mathbf{E}$ the set of powers. Notice that every power is a finite object with an unambiguous tree structure defined by the special delimiters $\{$ (, ␣ ,) $\}$. Every internal node is a list, and every leaf node is a bytestring.

6.3 Auths

Recall that a tag, which specifies either a request or a set of delegated permissions, represents a (usually infinite) set of powers. Our next task is to define the set of *auths*, each a subset of $\mathbf{E}$. In the next section, I show that the set of SPKI tags is, with some caveats, isomorphic to the set of auths I define here.

I first define the base-case auths:

$$A_{null} = \varnothing \qquad \text{(Definition T1)}$$

$$A_{*} = \mathbf{E} \qquad \text{(Definition T2)}$$

$$A_{bs}(b) = \{b\} \qquad (\forall b \in \mathbf{B})$$

(Definition T3)

$$A_0 = A_{null} \bigcup A_{*} \bigcup_{b \in \mathbf{B}} A_{bs}(b) \qquad \text{(Definition T4)}$$

Then I extend it recursively:

$$
\begin{aligned}
& A_{set}(a_1 ... a_k) \\
&= \bigcup_{1 \le i \le k} a_i && \text{(Definition T5)} \\
& A_{list}(a_1 ... a_k) \\
&= \left\{ \begin{array}{l} (x_1 \sqcup x_2 \sqcup \ldots \sqcup x_k \sqcup \cdots \sqcup x_n) \\ \text{such that} \\ \quad x_i \in a_i \quad \forall i, 1 \le i \le k \\ \quad x_i \in \mathbf{E} \quad \forall i, k < i \end{array} \right\} && \text{(Definition T6)} \\
A_j = & \quad A_{j-1} && (\forall j > 0) \\
& \bigcup \; A_{set}(a_1 \ldots a_k) \quad \forall a_i \in A_{j-1} \\
& \bigcup \; A_{list}(a_1 \ldots a_k) \quad \forall a_i \in A_{j-1} \\
& && \text{(Definition T7)}
\end{aligned}
$$

Finally, the set A of auths is the union of all of the A_j.

Observe that every member $a \in A$ is indeed a member of $2^{\mathbf{E}}$. I show this inductively. Clearly A_{null} and A_* are members of the power set of $\mathbf{E}$. By rule $\mathbf{E} := \mathbf{B}$, every singleton set A_{bs} belongs to the power set of $\mathbf{E}$. This shows the base case, $A_0 \in 2^{\mathbf{E}}$.

An auth introduced at any set A_j is either an A_{set} or an A_{list}. If it is an A_{set}, it is formed of the union of other a_i from A_{j-1}. By the induction hypothesis, each a_i is a subset of $\mathbf{E}$, and hence their union is as well. If the new auth is instead defined by A_{list}, it is composed of strings $(x_1 \sqcup \ldots \sqcup x_n)$. Each x_i belongs to $\mathbf{E}$, either by its requirement to belong to a subset a_i of $\mathbf{E}$, or by its requirement to belong to $\mathbf{E}$ itself. By rule $\mathbf{E} := (\mathbf{L})$, we know that $\mathbf{E}$ includes any list formed of other members of $\mathbf{E}$, which ensures that every such string $(x_1 \sqcup \ldots \sqcup x_n)$ is indeed in $\mathbf{E}$.

6.4 Closure of auths under intersection

The proof that A is closed under intersection is constructive, and in fact leads directly to a concrete implementation of tag intersection. Put another way, this proof provides direct intuition why the SPKI tag intersection procedure is meaningful.

Given any two members $a_x, a_y \in A$, we wish to exhibit $a_z = a_x \cap a_y$, with $a_z \in A$. We know that there exist i_x and i_y such that $a_x \in A_{i_x}$ and $a_y \in A_{i_y}$; let us assume we have the smallest such i_x and i_y. We will show by induction over $\max(i_x, i_y)$ that $a_x \cap a_y \in A$; that is, there exists some positive integer j such that $a_x \cap a_y \in A_j$.

It is actually possible to show that $a_x \cap a_y \in A_{\max(i_x,i_y)+1}$, but it's waaaaay more work, and surely unnecessary. See notes 1/31/2000-2/1/2000

The base case of the induction has $\max(i_x, i_y) = 0$; that is, $a_x \in A_0$ and $a_y \in A_0$. We show in cases I, II, and III that $a_x \cap a_y \in A_0$; all possibilities for the base case appear in the italicized upper-left-hand corner of Table 6.1.

The induction hypothesis assumes for all $i_x, i_y < n$, there exists a j' such that $a_x \cap a_y \in A_{j'}$. Our task, given $a_x \in A_{i_x}$ and $a_y \in A_{i_y}$ with $i_x, i_y \le n$, is to exhibit j such that $a_x \cap a_y \in A_j$. We do so by constructing a set equal to the intersection using one of the

	a_x				
a_y	A_{null}	A_*	$A_{bs}(b)$	$A_{list}(\dots)$	$A_{set}(\dots)$
A_{null}	***I***	***I***	***I***	I	I
A_*	***I***	***II***	***II***	II	II
$A_{bs}(b)$	***I***	***II***	***III***	IV	VI
$A_{list}(\dots)$	I	II	IV	V	VI
$A_{set}(\dots)$	I	II	VI	VI	VI

Table 6.1: *Pairwise possibilities for set intersection. Roman numerals indicate the proof section that handles the given case. The emphasized entries in the upper-left corner are the cases handled in the base case of the inductive proof.*

five auth formulas A_{null}, A_*, $A_{bs}(b)$, A_{set}, or A_{list}. Then we demonstrate that the set given by the formula is in some A_j. The choice of formula depends on how a_x and a_y came to belong to A_{i_x} and A_{i_y}. For example, if i_x (the smallest index for which $a_x \in A_{i_x}$) is zero, then we know either $a_x = A_{null}$, $a_x = A_*$, or $a_x = A_{bs}(b)$ for some bytestring b. Otherwise, if $i_x > 0$, then $a_x = A_{list}(\dots)$ or $a_x = A_{set}(\dots)$. The same options are possible for a_y. To construct the intersection, we must consider all of the pairwise possibilities. Table 6.1 maps each possibility to a proof case below. Notice we reuse base cases I and II in the inductive step.

Case I. Either $a_x = A_{null} = \varnothing$ or $a_y = A_{null} = \varnothing$, so their intersection is empty, and can be represented by A_{null}. $A_{null} = \varnothing$ belongs to A_0.

Case II. Assume without loss of generality (WOLOG) that $a_x = A_* = \mathbf{E}$ (if instead it is $a_y = A_*$, the proof works symmetrically). Then $a_x \cap a_y = a_y$. Since $a_y \in A_i$, we have $a_x \cap a_y \in A_i$.

Case III. Both a_x and a_y are singleton bytestrings. If they contain the same bytestring b, then their intersection is clearly $a_x \cap a_y = A_{bs}(b) = a_x$, which we know to be in A_i. Otherwise, the bytestrings are different, the singleton sets intersect to $\varnothing$, and we have $a_x \cap a_y = \varnothing = A_{null} \in A_0$.

Case IV. Assume WOLOG that $a_x = A_{list}(\dots)$ and $a_y = A_{bs}(b)$. Then every member of a_x is a string beginning with the special list delimiter ⦇, but the single member of a_y does not. Therefore their intersection is null, a member of A_0.

Case V. Let $a_x = A_{list}(c_1 \dots c_j)$, and $a_y = A_{list}(d_1 \dots d_k)$. Assume WOLOG $j \leq k$. By Definition T6, the intersection $a_x \cap a_y =$

$$\left\{ \begin{array}{l} ⦇ x_1 \sqcup x_2 \sqcup \dots \sqcup x_k \sqcup \dots \sqcup x_n ⦈ \\ \text{such that} \\ \quad x_i \in c_i \quad \forall i, 1 \leq i \leq j \\ \quad x_i \in \mathbf{E} \quad \forall i, j < i \\ \quad x_i \in d_i \quad \forall i, 1 \leq i \leq k \\ \quad x_i \in \mathbf{E} \quad \forall i, k < i \end{array} \right\}$$

We can rewrite the conditions as:

$$\left\{ \begin{array}{ll} \{\!|x_1 \sqcup x_2 \sqcup \ldots \sqcup x_k \sqcup \cdots \sqcup x_n|\!\} & \\ \quad \text{such that} & \\ x_i \in c_i \cap d_i & \forall i, 1 \leq i \leq j \\ x_i \in \mathbf{E} \cap d_i & \forall i, j < i \leq k \\ x_i \in \mathbf{E} \cap \mathbf{E} & \forall i, k < i \end{array} \right\}$$

Since $a_x, a_y \in A_i$, we know by Definition T7 that $c_1 \ldots c_j \in A_{i-1}$ and $d_1 \ldots d_k \in A_{i-1}$. Let

$$e_i = \left\{ \begin{array}{ll} c_i \cap d_i & \forall i \leq j \\ d_i & \forall j < i \leq k \end{array} \right.$$

The induction hypothesis gives us j' such that $e_i \in A_{j'}$. I can now write the intersection as

$$\begin{aligned} a_x \cap a_y &= A_{list}(e_1 \ldots e_k) \\ &= \left\{ \begin{array}{ll} \{\!|x_1 \sqcup x_2 \sqcup \ldots \sqcup x_k \sqcup \cdots \sqcup x_n|\!\} & \\ \quad \text{such that} & \\ \quad\quad x_i \in e_i & \forall i, 1 \leq i \leq k \\ \quad\quad x_i \in \mathbf{E} & \forall i, k < i \end{array} \right\} \end{aligned}$$

Because $e_i \in A_{j'}$, we conclude that $a_x \cap a_y \in A_{j'+1}$.

Case VI. Assume WOLOG $a_x = A_{set}(a_1 \ldots a_k)$. Let $a_z = A_{set}(a_1 \cap a_y, a_2 \cap a_y, \ldots a_k \cap a_y)$. To see that $a_z = a_x \cap a_y$, observe that for any auth a,

$$\begin{aligned} & a \in a_x \cap a_y \\ \equiv\ & a \in a_x \wedge a \in a_y \\ \equiv\ & (\exists i \leq k, a \in a_i) \wedge a \in a_y \\ \equiv\ & \exists i \leq k, (a \in a_i \wedge a \in a_y) \\ \equiv\ & \exists i \leq k, (a \in a_i \cap a_y) \\ \equiv\ & a \in a_x \end{aligned}$$

We know $a_i \in A_{x_i - 1}$ for $i \leq k$, since $a_x \in A_{x_i}$. By the induction hypothesis we know that there exist $j_1 \ldots j_k$ such that $a_m \cap a_y \in A_{j_m}$ for $m \leq k$. Let $j = \max_m(j_m) + 1$. Because A_{j-1} contains every A_i for $i \leq j - 1$, we have $a_m \cap a_y \in A_{j-1}$ for all $m \leq k$. By our construction of a_z, $a_z \in A_j$.

Having covered every possible combination of a_x and a_y, we have shown the induction, and hence that A is closed under intersection. Furthermore, the cases above direct our implementation of tag intersection: given any two tags, we know which constructor (such as A_{list}) was used to create it, since tags are represented as such constructions. We can immediately apply the techniques in the preceding cases to discover a tag construct that represents the intersection of the input tags.

6.5 Tags

Tags in SPKI are approximately isomorphic with auths. There are three caveats, related to null tags, a special requirement on lists, and the special range and prefix tags.

6.5.1 The null tag

The first caveat is that SPKI has no representation for the null tag (A_{null}). The result is that the SPKI documentation must tread clumsily around the issue by saying that two authorizations "fail to intersect," rather than intersecting to a null set. By including the null set, we promote "failure" to a first-class object representable in the system.

6.5.2 Lists have an initial bytestring element

The second caveat is that lists in SPKI tags must always have at least one element, and the first element can only be a non-empty bytestring. One can readily redefine **B**, **E** and A_{list} to satisfy this constraint. The basic structure of A does not change; I depend only on each A_j's membership in $2^{\mathbf{E}}$.

6.5.3 Special tags cause havoc

The third caveat is that SPKI has special tags `range` and `prefix` that define infinite subsets of the set of bytestrings. Our auth structure A can be readily extended by `prefix`, but with `range` present, A is no longer closed under intersection. Consider for example the SPKI tags:

```
(tag (* range numeric ge 0.5 le 0.5))
(tag (* prefix 000))
```

Their intersection is $A_{range}(0.5, \leq, \leq, 0.5) \cap A_{prefix}(000)$, which we know belongs to **E**. We can see, however, that it does not belong to A. The set contains an infinite number of bytestrings. We cannot construct it with an A_{prefix}, or we would end up with numeric values other than 0.5; we cannot construct it with an A_{range} or we would have prefixes other than 000. The only other way to introduce bytestrings is A_{bs}, which introduces only one at a time. We may union together any finite number of bytestrings with each application of A_{set}, but by no A_j will we have constructed the infinite set of bytestrings necessary to describe the intersection of the tags in the example.

Indeed, the trouble is concentrated in the `range` form. The intersection of two ranges with different ordering specifications can be an infinite set of bytestrings not representable with the `range` or `prefix` form.

How can we escape the dilemma? We can omit the `range` special form, but that would not provide a satisfying model of SPKI. We could introduce an intersection operator analogous to the union operator A_{set}, but that would be a hack. Since A is otherwise closed under intersection, an intersection operator should never be used except when intersecting these curious bytestring expressions, for it would only lead to needlessly larger representations for auths. Finally, we may accept the incompleteness of tags. Assume $t_3 = t_1 \cap t_2$, that is, the tag-intersection procedure run on tags t_1 and t_2 produces tag t_3. Let $A(t)$ be a function mapping a tag to the auth it represents, a typically-infinite subset of **E**. If tags are complete, then $A(t_3) = A(t_1) \cap A(t_2)$. If we must sacrifice the completeness of tags, we still know that $A(t_3) \subseteq A(t_1) \cap A(t_2)$. This provides assurance that the authorization procedure is at least still sound: we will not conclude a chained delegation confers powers that are not conferred by both members of the chain.

a_y \ a_x	A_{null}	A_*	$A_{bs}(b)$	A_{prefix}	A_{range}	$A_{list}(\dots)$	$A_{set}(\dots)$
A_{null}	I	I	I	***I***	***I***	I	I
A_*	I	II	II	***II***	***II***	II	II
$A_{bs}(b)$	I	II	III	***VII***	***VII***	IV	VI
A_{prefix}	***I***	***II***	***VII***	***VIII***	***IX***	***X***	***VI***
A_{range}	***I***	***II***	***VII***	***IX***	***IX***	***X***	***VI***
$A_{list}(\dots)$	I	II	IV	***X***	***X***	V	VI
$A_{set}(\dots)$	I	II	VI	***VI***	***VI***	VI	VI

Table 6.2: *Pairwise possibilities for set intersection in the presence of the range and prefix auth constructors. The emphasized entries are additions beyond Table 6.1.*

Treating the intersection of a `range` with a `range` or `prefix` as null should not be terribly limiting in practice. When either form is used, it is specifying a value for some field with a particular interpretation; it is likely that in a real system any given field would only have one meaningful mode of comparison.

6.5.4 Semantics of special tags

SPKI contains several special forms for tags: `(*)` represents the auth A_*. `(* set ...)` represents the auth $A_{set}(...)$. `(* prefix ...)` and `(* range ...)` represent (possibly infinite) sets of bytestrings. To model these tags, we need to extend the base definition of A:

$$A_{prefix}(p) = \{b \mid b = ps, s \in \Sigma\} \quad \text{(Definition T8)}$$

$$A_{range}(f) = \{b \mid f(b) = true\} \quad \text{(Definition T9)}$$

$$A_0 = A_{null} \bigcup A_* \bigcup_{b \in \mathbf{B}} A_{bs}(b) \bigcup_{p \in \mathbf{B}} A_{prefix}(p) \bigcup_{b \in f} A_{bs}(b) \quad \text{(Definition T10)}$$

The function $f : \Sigma^* \to \{true, false\}$ selects a range of bytestrings, and is used here as an abbreviation to hide that complexity. The function depends on the specified ordering (alpha, numeric, time, binary, or date), the (optional) low and high bounds, and bits specifying whether each bound is exclusive or inclusive.

The matrix of intersection cases must now be extended to support the new possibilities (see Table 6.2). This extension of course will no longer show the completeness of A under intersection (unless A_{range} is removed). But it is still useful as a thorough guide to intersecting tags.

Case VII. In this case, assume $a_x = A_{bs}(b) = \{b\}$ and $a_y = A_{prefix}(p)$ or $a_y = A_{range}(f)$. This case is very similar to case III: if $b \in a_y$, then the intersection is $a_z = A_{bs}(b) = a_x$; otherwise it is $\varnothing = A_{null}$.

Case VIII. We have $a_x = A_{prefix}(p_x)$ and $a_y = A_{prefix}(p_y)$. Assume WOLOG $|p_x| > |p_y|$ (p_x is a longer string). If p_y is a prefix of p_x (that is, $p_x = p_y s$), then $a_x \cap a_y = a_x$: if $b \in a_x$, $b = p_x s' = p_y s s'$, so $b \in a_y$. Otherwise, when p_y is not a prefix of p_x, the intersection is empty: $b \in a_y$ implies $b = p_y s$, and we know p_y disagrees at some symbol position with p_x, so $b \notin a_x$.

Case IX. Set a_x is a range and set a_y is a range or a prefix. Often we will treat the intersection as null (to preserve the soundness of auths). In a specific circumstance, when both a_x and a_y are ranges specified with the same ordering function, we can readily construct a range equal to the intersection by taking the more restrictive of the bounds from each input range.

Case X. Assume WOLOG a_x is a list and a_y is a range or prefix. In this case, every string in a_x begins with the list delimiter `(`, and every string in a_y begins with a symbol in the octet alphabet (Σ), so the intersection is null.

Notice that only case IX spoils the completeness property; striking A_{range} from our definition removes its row and column from the matrix, eliminating any reference to case IX.

6.6 The meaning of intersection

The SPKI documentation describes the intersection of two authorization tags as having two possible outcomes: a new tag or a failure to intersect. These results are meant to be interpreted differently depending on whether the intersection operation was between two delegations, or between a delegation and a specific request.

In the former case, the desire is that the tag that is the product of the intersection represents no more power than either argument tag delegated by itself, and that if the intersection fails, then the combination of the delegations is worthless.

In the latter case, the desired interpretation is that should the intersection succeed at all, the request must be authorized by the delegation tag. This interpretation makes sense if every request is more specific than any delegated permission; the only intersection possible is to return the request tag.

My semantics lends a very concrete interpretation in both cases. When intersecting delegated permissions, it returns exactly the subset of powers granted by both input tags (modulo the incompleteness introduced by the `range` form, in which case it returns a subset of the intersection of the powers). If the tags have a null intersection, we treat that object just like any other; however, because it is an empty set of powers, it is worthless in that no request will be authorized by it.

When authorizing requests, we intersect the delegation tag with the request tag, *and test whether the result equals the request tag.* If so, we conclude that all of the (typically infinite) set of powers demanded by the request are granted by the delegation. If not, we conclude that some smaller (possibly empty) set of powers were granted; in any case, they are not sufficient to justify granting the request. With my semantics, it is not necessary for all requests to be finer grained than all delegated permissions. This property ensures that user-defined tag structures can be readily extended without confusing the meaning of the request authorization test.

Set containment provides a concrete, mathematically sound interpretation for specifying authorizations in an infinitely-extensible fashion.

6.7 Order dependence

The semantics of SPKI tags specifically depend on the order of elements in a list; intersection of two lists involves pairwise intersection of each list's elements. Because lists are implicitly followed by an arbitrarily-long supply of A_*s, lists are extensible in that a new property can be defined for the list and assigned to the next unused position in the list.

For example, imagine that one is defining a tag format for delegating access to a database of employee records. A first-cut tag format might look like:

```
(employee
  (id (*))
  (salary (*))
)
```

Such a definition gives users of the system the ability to delegate to others rights such as the right to inspect only employee records with a specific ID number `(employee (id 01247) (*))`, or those of employees earning more than $50,000 `(employee (*) (salary (range gt 50000)))`. Because tags are extensible, one may later decide that the ability to select employees based on anniversary year is useful, so the definition is extended to:

```
(employee
  (id (*))
  (salary (*))
  (anniversary (*))
)
```

Because list tags are followed by implicit `(*)` members, all existing delegation tags continue to be meaningful even when the new format is deployed.

What happens, however, if two independent organizations want to extend the format independently? In our example, perhaps one department of the corporation wishes to add the anniversary extension (and does so for their internal applications), and another department adds an extension representing hair color to the tag format used in their applications. The resulting extensions do not necessarily compromise security (since each sublist is annotated with the name of the attribute it refers to), but the extensions may never be used together: that third spot in the `employee` list can only contain one sublist.

There is an attractive solution. The semantic definition of auths does not preclude assigning elements to locations in lists with arbitrarily high indices. Therefore, when assigning a new extension such as `anniversary`, we could simply assign it to the index of the `employee` list given by the ordinal value of the bytestring "`anniversary`." This approach, however, is not desirable with the current definition of tag representations. A tag using the `anniversary` extension, for example, would contain some 1.2×10^{26} instances of `(*)` to place the `(anniversary ...)` sublist in the correct location.

To make the solution work, I propose a simple extension to SPKI's special-form tags, the named-attribute (`named`) form:

`(* named (`*`attribute-name ...`*`))`

An named-attribute tag expression always has a single list argument. The first element of the list is a bytestring (a requirement in SPKI), which I call the *attribute-name*, and the remainder of the list is the associated *value*. Let ord(b) be the ordinal value of a bytestring b. A list containing a `(* named (`*`attribute ...`*`))` special form would represent the list in which the argument of the special form appears at position ord(`attribute`) in the list.

6.7.1 Handling non-bytestring attribute names

Our general semantics does not require lists to begin with a bytestring. We can easily define an ordering over Σ' rather than just Σ, and compute the location of the attribute in the parent list based on the ordinal value of the attribute name, even when that "name" is itself a list. This fix does not handle lists containing sets. The use of sets as attribute names would require some canonical ordering of the members in the set. Their use would also require a unique representation for any auth containing a set. It turns out that such a representation is indeed possible, formed by bubbling every set operator out of the inside of lists and joining them. This canonical form makes small tags into large ones; for example, `(a (* set b c d) e)` becomes `(* set (a b e) (a c e) (a d e))`. Fortunately, the *ord*() function is only a theoretical construct used in the semantics; it would never be needed in any implementation of auth intersection.

6.7.2 Interference between ordered and named attributes

With the definition given above, one might cause an unexpected interaction between named attributes and attributes specified by their order when their positions in the list coincide. For example, if ord(A) = 65, one might construct a list with sixty-five attributes specified in order, as well as a named attribute `(* named (A cat))`. The named attribute and the sixty-fifth ordered attribute would coincide, and so far I have given no specification for how to map a list tag to an A_{list} when the tag specifies multiple auths for the same position in the list.

The semantic solution is simple: let the n^{th} ordered attribute appear at location $2n-1$ in the list, and let each named attributed appear at location $2 \cdot \text{ord}(\textit{attribute-name})$ in the list. There are an infinite supply of odd and even list locations, and they do not interfere with one another.

What should an implementation do with a tag that specifies the same named attribute twice? It seems natural that the list location should contain the intersection of the associated values.

6.7.3 Intersection of lists containing named attributes

In any real implementation, of course, we cannot expand a list containing named attributes into its semantic form, since the length of the list grows exponentially with the length of the names of the attributes. We expect the lists to be sparse, so a sparse representation of

the lists should work well. Store explicit position indices (*attribute-names* where defined, and the list index otherwise) alongside the corresponding values, with the entire collection sorted by position. The intersection routine walks the lists simultaneously and invokes itself recursively whenever it encounters two values with the same *attribute-name* or position index. As in the basic list intersection routine, if only one list specifies a value for a given position, the other list's value is assumed to be A_*, and the intersection is the explicitly specified value.

6.7.4 Recommendations for the use of ordered and named attributes

In the SPKI RFC, the authors suggest that while lists can be used to name attributes, one can omit the names for compactness. In their example,

```
(ftp (host ftp.clark.net) (dir /pub/cme))
```

becomes:

```
(ftp ftp.clark.net /pub/cme)
```

The rationale is that attributes are position-dependent, so there is no ambiguity when the attribute names (`host` and `dir`) are dropped.

Indeed, any correct mechanical implementations can infer the meaning of the values by their position in the list. It is likely, however, that a human implementor may incorrectly infer the intent of the values, perhaps because he only has access to example attribute values but not the names.

I recommend that attributes be supplied with names whenever possible. Whether attribute positions are specified by order or by a named-attribute special form is immaterial; that decision is one of expediency, and can be made based on the likelihood that a given attribute will be omitted from a tag specification. Providing names that document the meanings of values, however, helps avoid ambiguity, especially in a structure that is intended to be extensible in the future and by unknown parties. My recommendation is an example of principle 1 from [AN96]: "Every message should say what it means."

6.8 Analogy with Dedekind cuts

This perspective on SPKI auth tags has a pleasant analogy to Dedekind's construction of the real numbers [BM91, pp. 15–17]. Each real number α is defined by an infinite set of rational numbers less than α; the result is continuity. The rationals are totally ordered. Every Dedekind cut respects that ordering by containing every rational less than any rational that appears in the cut.

In the construction of auths as sets of powers presented here, powers can be partially ordered, and auths respect that ordering by containing every power less than any power that appears in the auth. The result is a kind of density corresponding to the continuity of the reals. Any two unequal reals have another real between them; by analogy, any list auth can be arbitrarily subdivided into smaller auths along an arbitrary number of dimensions. This limitless extensibility makes SPKI auth tags adaptable to changing environments.

Chapter 7

Modeling SPKI

The original Calculus for Access Control is useful because its principals are general enough to model several parts of a computing system, from users to trusted servers to communications channels. By adding the ability to restrict authority with any delegation, we make the calculus powerful enough to be useful across administrative domains. Perhaps the most convincing evidence of this power is how well our extended calculus can model SPKI, an access-control system designed to span administrative domains.

For the reader unfamiliar with SPKI, I have supplied a concise overview in Appendix D. To formally model SPKI with our extended calculus, we first give a construction that models the delegation-control bit.

7.1 Delegation control

The SPKI document gives the motivation for including a delegation-control bit in SPKI certificates. We disagree with the argument and fall in favor of no delegation control, and for the same reasons as described in the document: delegation control is futile, and its use tempts users to divulge their keys or install signing oracles to subvert the restriction. Such subversion not only nullifies delegation control, but forfeits the benefits of auditability provided by requiring proofs of authorization. In spite of our opinion, we present a construction that models delegation control.

To model the delegation-control feature we wish to split the **says** modality into two separate modalities: "utterance," which represents a principal actually making a statement, and is never automatically inherited by other principals, and "belief," which is inherited transitively just as **says** is. Not only is introducing a new logical modality clumsy, but it would require us to support a dubious axiom, undermining the simplicity of the semantics.

Instead, we resort to an equivalent construct: we split each "real" principal A we wish to model into subprincipals A_u and A_b. A_u shall say only the things that A utters (statements that are actually signed by A's key; recall that all certificate-issuing principals in SPKI are keys), and A_b shall say all of the things that A believes. A may inherit her beliefs from other principals (because she has delegated to other subjects the authority to speak on her behalf), and furthermore A should believe anything she utters. This last condition replaces the clumsy axiom we wished to avoid; instead we enforce it by explicitly assuming

the following statement for all principals A and statements s:

$$\vdash A_u \,\mathbf{says}\, s \supset A_b \,\mathbf{says}\, s \qquad \text{(Assumption E22)}$$

Certificates issued by A are statements uttered by A asserting things that A believes, so we model them as statements about A_b said by A_u. The desirable outcome is that no principal can delegate authority to make herself utter something (make A_u say something); she may only utter the statement directly (by signing it with her key).

7.2 Restriction

Recall that a SPKI 5-tuple includes five fields: issuer, subject, delegation-control bit, authorization, and validity dates. Let I and S represent the issuer and subject principals. Let T_A represent the set of primitive permissions represented by the authorization S-expression, and T_V the set of primitive permissions limited by the validity dates (assuming the effective-time encoding of Section 4.2.3). The 5-tuple can be represented this way if its delegation-control bit is set:

$$I_u \,\mathbf{says}\, S_b \stackrel{T_A \cap T_V}{\Rightarrow} I_b$$

or this way if not:

$$I_u \,\mathbf{says}\, S_u \stackrel{T_A \cap T_V}{\Rightarrow} I_b$$

A 4-tuple has a name field (N) and no authorization field or delegation-control bit. It would be encoded:

$$I_u \,\mathbf{says}\, S_b \stackrel{T_V}{\Rightarrow} I_b \cdot N$$

It seems natural that a delegation bit is meaningless for a name binding, for in SPKI, a name principal can never utter a statement directly, only a key principal can. It surprised us, however, that SPKI name-binding certificates omit the authorization field. Why not allow a principal to say the following?

$$I_u \,\mathbf{says}\, (S_{2b} \stackrel{\{shampoo\}}{\Rightarrow} I_b \cdot N_{\text{barber}})$$

As it turns out, our semantics does not support such restricted name bindings (see Section 5.3).

7.3 Linked local namespaces

The subject principals in the keys above may be either keys (each directly represented by a primitive principal) or a string of names grounded in a key. Hence namespaces are "local" in that names are meaningless except relative to a globally unambiguous key; namespaces are "linked" in that the naming operation may be repeated: If $K_1 \cdot N_1$ resolves to K_2, then $K_1 \cdot N_1 \cdot N_2$ is the same as $K_2 \cdot N_2$, perhaps defined as some K_3.

We gave a logic and semantics for linked local namespaces in Section 5. We model the SPKI name subject "george: (name fred sam)" with the principal expression $K_{\text{george}} \cdot N_{\text{“fred”}} \cdot N_{\text{“sam”}}$. Substituting the principal expression for S_b, a 4-tuple takes on the general appearance:

$$I_u \textbf{ says } ((K_S \cdot N_1 \cdots N_k) \stackrel{T_V}{\Rightarrow} I_b \cdot N_0)$$

7.4 Threshold subjects

A threshold subject is a group of n principals who are authorized by a certificate only when k of the principals agree to the requested action. Such certificates are really just an abbreviation for a combinatorially-long $\binom{n}{k}$ list of conjunction statements. For example, a certificate with a 2-of-3 threshold subject naming principals P_1, P_2, and P_3 and an issuer A can be represented as:

$$P_1 \wedge P_2 \Rightarrow A$$
$$P_1 \wedge P_3 \Rightarrow A$$
$$P_2 \wedge P_3 \Rightarrow A$$

Hence the logic easily captures threshold subjects, although any tractable implementation would obviously want to work with them in their unexpanded form.

7.5 Auth tags

The "auth tags" used in authorization fields in SPKI represent sets of primitive statements. Therefore, we simply model them using mathematical sets.

7.6 Tuple reduction

The SPKI access-control decision procedure is called "tuple reduction." A request is granted if it can be shown that a collection of certificates reduce to authorize the request. The reduced tuple's subject must be the key that signed the request; the tuple's issuer must represent the server providing the requested service; and the specific request must belong to the authorization tag of the reduced tuple.

It is clear that tuple reduction is sound with respect to our extended logic. When 5- and 4-tuples are encoded in the logic as shown in Sections 5 and 7.2, tuple-reduction simply constructs a proof from several applications of Theorem E6 and Axiom E17.

7.7 Validity conditions

An optional validity condition, such as a certificate revocation list, a timed revalidation list, or a one-time validation, can be encoded in the logic using a conjunction. For example, a certificate requiring a timed revalidation would be interpreted

$$A \textbf{ says } (B \wedge (R|H_1)) \Rightarrow A$$

to mean that principal R must verify that this certificate (with hash H_1) is valid. Principal R signs a revalidation instrument I with a short validity interval T_V

$$R \,\mathbf{says}\, I \stackrel{T_V}{\Rightarrow} R$$

and a given revalidation instrument would agree with all valid outstanding certificates:

$$I \,\mathbf{says}\, \mathbf{0} \Rightarrow I|H_1$$
$$I \,\mathbf{says}\, \mathbf{0} \Rightarrow I|H_2$$
$$\vdots$$

The principal $\mathbf{0}$ has relation $\mathcal{R}(\mathbf{0}) = \varnothing$, so that every principal speaks for $\mathbf{0}$. Using the logic, we can reason that

$$\mathbf{0} \Rightarrow I|H_1 \stackrel{T_V}{\Rightarrow} R|H_1$$

and since $B = B \wedge \mathbf{0}$, $B \stackrel{T_V}{\Rightarrow} A$. Notice the treatment of a certificate's hash as a principal. In the logic, principals are general entities and can be used to represent many objects and actors.

Negative certificate revocation lists can be handled similarly; an implementation examining a revocation list would conclude $I \,\mathbf{says}\, \mathbf{0} \Rightarrow I|H_1$ for any H_1 *not* present in the list.

One-time revalidations are meant to be interpreted as having a zero validity interval. A system verifying a request s creates a nonce E, understanding $E \,\mathbf{says}\, s$, and sends it to the revalidator R. R replies with a statement meant to be interpreted

$$R \,\mathbf{says}\, E \stackrel{\{s\}}{\Rightarrow} R|H_1$$

Now both B_1 and $E \stackrel{\{s\}}{\Rightarrow} R|H_1$ say s, so $A \,\mathbf{says}\, s$. Any future request of the same sort will require another revalidation, for its s will have a different effective time.

7.8 Safe extensions

Our semantics suggests that SPKI may be safely extended to support a variety of principals other than public keys. Channels protected by secret keys or a trusted computing base, for example, are easily modeled as principals in the logic. Conjunct principals ($A \wedge B$) are not first-class entities in SPKI, although they can appear as threshold subjects; an extended SPKI might exploit Theorem E20.

Quoting principals are also missing from SPKI; Lampson et al. give nice examples showing how quoting can help a multiplexed server or communications channel differentiate when it is working on behalf of one client versus another [LABW92, Sections 4.3, 6.1, 6.2, and 7.1]. Without quoting, such a server has permission to make statements for either client, so it must perform an access-control check in advance of relaying a client's statement. Quoting lets the multiplexed server defer the complete access-control decision to the final resource server that verifies the proof. The result is a smaller trusted computing base and improved auditability.

7.9 Dangerous extensions

In this section, we argue that SPKI auth tags should not be extended to represent logical negations. If $\mathcal{B}$ speaks for $\mathcal{A}$ regarding multiple restriction sets, the semantics suggest that $\mathcal{B}$ actually has some authority not explicitly mentioned in either set. For example,

$$(\mathcal{B} \overset{\{\sigma,\tau\}}{\rightarrow} \mathcal{A}) \supset (\mathcal{B} \overset{\{\sigma \wedge \tau\}}{\rightarrow} \mathcal{A}) \qquad \text{(Axiom E23)}$$

means that a principal believed on a set of statements is also believed on their conjuncts. This conclusion seems fairly natural, but it is interesting to note that a restriction set actually permits more statements than it represents explicitly.

With our projected version of Abadi's speaks-for semantics, not only does

$$(\mathcal{B} \overset{\{\sigma,\tau\}}{\Rightarrow} \mathcal{A}) \supset (\mathcal{B} \overset{\{\sigma \wedge \tau\}}{\Rightarrow} \mathcal{A}) \qquad \text{(Axiom E24)}$$

hold, but also:

$$(\mathcal{B} \overset{\{\sigma\}}{\Rightarrow} \mathcal{A}) \supset (\mathcal{B} \overset{\{\neg\sigma\}}{\Rightarrow} \mathcal{A}) \qquad \text{(Axiom E25)}$$

This result implies that given authority on a set of primitive statements, a principal also has authority on any propositional formula constructed from those statements. It is surprising, for even if only $B \overset{\{s\}}{\Rightarrow} A$ is explicitly granted, B can also cause A to say the negation of s.

Perhaps scarier still is that

$$\begin{aligned} B \overset{\{\sigma\}}{\Rightarrow} A &\supset B \overset{\{\sigma,\neg\sigma\}}{\Rightarrow} A \\ &\supset (B \,\mathbf{says}\, \text{false}) \supset (A \,\mathbf{says}\, \text{false}) \end{aligned}$$

The conclusion is the definition of Abadi's $\mapsto$ relation:

> "Intuitively, $A \mapsto B$ means that there is something that A can do (say *false*) that yields an arbitrarily strong statement by B (in fact, *false*). Thus, $A \mapsto B$ means that A is at least as powerful as B in practice." [ABLP93, p. 713]

With these semantics, one might fear that no restriction is actually meaningful. How might we escape it? One option is to abandon the **K** axiom ($A \,\mathbf{believes}\, s \wedge A \,\mathbf{believes}\, (s \supset t) \supset A \,\mathbf{believes}\, t$), so that principals no longer believe every consequence of their beliefs. This option is undesirable because it cripples the logic to only operate outside the scope of belief operators.

A second option is to both disallow negative statements in restriction sets and to use the weaker $B \overset{T}{\rightarrow} A$ relation instead of $B \overset{T}{\Rightarrow} A$ to model delegation.

A third option is to prevent principals from making contradictory statements. This is difficult in general in a distributed system. One approach is to prevent principals from making negative statements at all. SPKI takes this approach. Its tags, which represent both restriction sets and individual statements, cannot represent both a statement and its logical negation. We provide a formal treatment of tags in Section 6.

Another extension might be to allow SPKI name bindings (4-tuples) to include authorization restrictions. As mentioned in Section 5.3, the semantics does not support this seemingly-natural extension.

We conclude that in certain dimensions, SPKI is as strong as it can be. Changing SPKI by allowing principals to make negative statements or by allowing negative statements in restriction sets would push SPKI "over the edge," making its restrictions meaningless. Those proposing to augment SPKI or other systems based on a logic such as that presented here must be wary of this hazard.

7.10 Related work

Abadi provides a semantics for SPKI names in [Aba98], but its definition shares a flaw with that used for roles in [ABLP93]. We discuss Abadi's name semantics in Section 5.4.

Halpern and van der Meyden supply an alternate semantics for SPKI names in [HvdM99], but it only encompasses the containment relation among names, and does not treat names as principals. As a result, it cannot relate names to compound principals nor relate names to other principals that are only connected by a restricted delegation.

Aura supplies a semantics for SPKI restricted delegation in [Aur98], but it is unsatisfying in that it essentially says what the reduction procedure says: a delegation is in place if there is a chain of delegation certificates and principals. It does not lend intuition about what the delegations mean. In contrast, our semantics connects restricted delegation to the logic of belief, a formal model that describes what a principal means when it delegates authority.

Part V

Sharing Implementation

Chapter 8

End-to-end authorization

Part V is concerned with my implementation of sharing across administrative domains. In this chapter, I describe how the theory in Part IV can be applied not only to span administrative boundaries, but to span other boundaries that appear in complex systems. The resulting implementation allows one to reason about sharing and protection *end-to-end*, from the provider of a resource to its consumer.

Saltzer et al. describe a general principal for computer engineering: implement semantics end-to-end to achieve correctness, and only implement semantics hop-by-hop as an optimization [SRC84]. Voydock and Kent argue for end-to-end security measures when the hops are between network routers [VK83]. The same principle holds for authorization semantics when the hops are between gateways and other high-level boundaries. End-to-end authorization makes systems more secure by reducing the number of programs that make access-control decisions, by giving those programs that do control access more thorough information, and by providing more useful audit trails.

In this chapter, I illustrate four kinds of boundaries in distributed systems that can impede the flow of authorization information from one end of a system to another. I discuss how my unified system can support end-to-end authorization across these boundaries.

In Chapter 9, I discuss my implementation of an infrastructure to support end-to-end sharing based on the semantics introduced in Part IV. Applications using that infrastructure require communications channels; Chapter 10 describes several channels that I implemented that plug into the infrastructure. In Chapter 11 I describe three applications built using the pieces from the preceding chapters. My measurements in Chapter 12 quantify the potential impact of my architecture.

8.1 Spanning administrative domains

Administrative boundaries frequently interfere with end-to-end authorization. The conventional approach to authorization involves authenticating the client to a local, administratively-defined user identity, then authorizing that user according to an access-control list (ACL) for the resource. When resources are to be shared across administrative boundaries, this scheme fails because the server has no local knowledge of the recipient's identity. Figure 8.1 illustrates this problem.

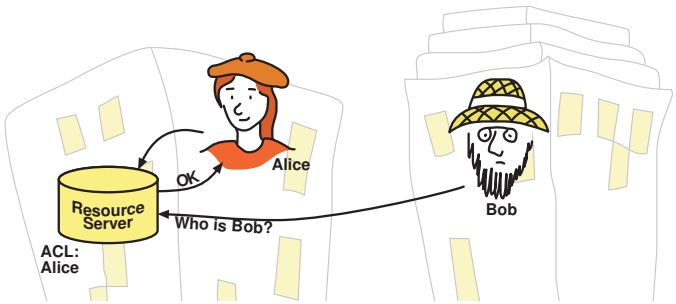


Figure 8.1: *A resource server contains an ACL that refers directly to Alice, a user in the same administrative domain. Alice cannot add Bob to the ACL, however, since he is in another administrative domain, and unknown to the resource server.*

Typical solutions to this problem involve authenticating the remote user in the local domain, either by having the local administrator create a new account, or by the resource owner sharing her password. Another approach is to install a gateway that accesses the resource with the local user's privilege but on behalf of the remote user. With the gateway the owner achieves her goal of sharing, but obscures the identity and authority of the actual client from the service that supplies the underlying resource.

Another way a user might share resources across administrative boundaries is by *delegating* her authority with *restriction*. In our example, Alice may authorize Bob to perform some restricted set of actions on certain resources (see Figure 8.2). Authority information flows across the administrative boundary: the delegation provides the resource server with sufficient information to reason about the client regardless of his membership in the local administrative domain. Indeed, the authorization mechanism has *no* inherent notion of administrative domain.

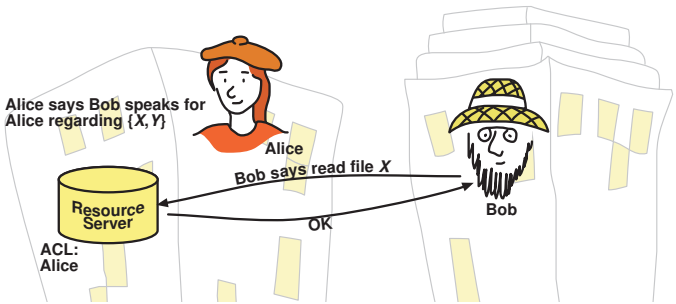


Figure 8.2: *With restricted delegation, Alice can introduce the remote principal Bob, and describe what authority he has over her resources. The resource server can reason about Bob's authority in terms of Alice's wishes, even though Bob is not a member of the local administrative domain.*

8.2 Spanning network scales

A second boundary that interferes with end-to-end authorization is network scale. Network scale affects an application's choice of hop-by-hop authorization protocol. For example, a strong encryption protocol is appropriate when crossing a wide-area network. Inside a firewall where routers are locally administered, some installations may base authority decisions on IP source addresses. On a local machine, we can often trust the OS kernel to correctly identify the participants in an interprocess communication.

If a service exploits my unified authorization model, the server uses a single API to reason about authority. One can plug in protocols (hop-by-hop authorization mechanisms), and the server can reason about the authorizations supported by each protocol. If the server does not support a given mechanism natively, it can still reason about authorization information from a protocol-translating gateway that passes through end-to-end authorization information using my system.

Figure 8.3 illustrates how I reason about authorization at different network scales. When Alice is at the same machine as the resource server (a), the server trusts the kernel to reliably report her identity, from which it derives her authority. Objects in my model encode the statements "the kernel says this interprocess communication (IPC) channel speaks for Alice" and "the server says the kernel speaks the truth." The server uses the authorization library to conclude that requests arriving on the IPC channel represent Alice's wishes.

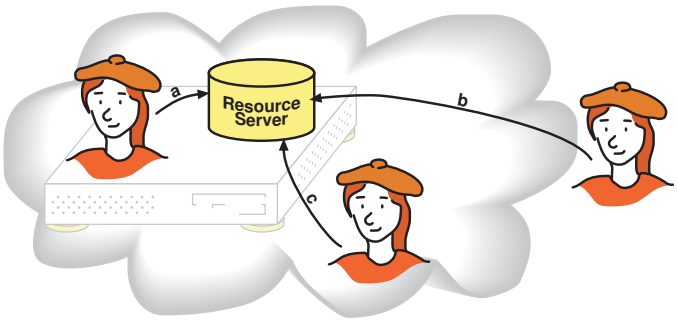


Figure 8.3: *When Alice connects to a resource server from the same machine, the server may trust the kernel to correctly identify her (a). When Alice connects from elsewhere on the Internet, the server may require strong encryption to verify her authority (b). When Alice connects from inside a department firewall, the server may accept IP source addresses as sufficient proof of her authority (c).*

In the wide-area network case (b), a standard encryption-based authentication protocol is used to bind an integral channel to a public key. The protocol is taken to ensure that "this channel speaks for Alice's public key," and if the server is convinced that "Alice's public key speaks for Alice," then it arrives at a conclusion like that in case (a): the requests arriving on the authenticated channel represent Alice's wishes.

Finally, perhaps a site wishes to use IP source addresses inside its firewall for a quick and dirty form of authentication (c). By writing statements like "the sysadmin says the router speaks for Alice's workstation" and "Alice's workstation quoting Alice speaks for

Alice," I encode this trust. The resource server concludes that the requests arriving on a TCP stream from a given source address represent Alice's wishes. Although some may consider this misplaced trust, the point is that my unified authorization system not only supports this trust model, but ensures that each final authorization decision represents the assumptions of trust that went into that decision.

Our unified approach separates policy from mechanism, creating two benefits. First, applications reason about policy using a toolkit with a narrow interface. The toolkit can transparently support multiple access mechanisms, and simply enable those that policy allows. Second, when an application does not support a desired mechanism, we can build a gateway that forwards requests from another mechanism while still passing end-to-end authorization information in a form the server can audit. Ultimately, the high-level security analysis of a program is independent of mechanism, and reflects end-to-end trust relationships.

8.3 Spanning levels of abstraction

Another use for gateway programs is to introduce another level of abstraction over that provided by a lower-level resource server. A file system takes disk blocks and makes files; a calendar takes relational database records and makes events; a source-code repository takes files and makes configuration branches. Figure 8.4 gives another illustration. Typically, an abstracting gateway controls the lower-level resource completely and exclusively, so that the gateway makes all access-control decisions. With end-to-end authorization, one can instead allow multiple mutually untrusting gateways to share a single lower-level resource.

Figure 8.4: *Alice's request for a high-level resource involves not only her authority over the final low-level resource, but some interaction with the service providing a level of abstraction. For example, Alice's authority may be expressed in terms of sandwiches, not beef.*

For example, a system administrator might speak for the disk-block allocator. To grant Alice access to a specific file X, the sysadmin may allow Alice to speak for the file system regarding X, and allow the conjunction of Alice and the file system quoting Alice to speak for the disk blocks. In this configuration, the file system cannot access the lower-level disk block resource without Alice's agreement (due to the conjunction), and Alice cannot meddle with arbitrary disk blocks without the file system agreeing that the requests are appropriate. The system helps us adhere to the principal of least privilege by encoding partial trust in the user and in the file system program. Furthermore, auditing any request for disk blocks provides end-to-end information indicating the involvement of both Alice

and the file system program.

8.4 Spanning protocols

Commonly a gateway is installed between two systems simply to translate requests from one wire protocol to another, as illustrated in Figure 8.5. Like any gateway, these gateways often impede the flow of authorization information from client to server.

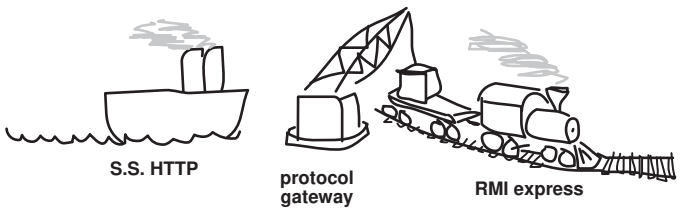


Figure 8.5: *Gateways are necessary to translate between protocols, but they frequently impede the flow of authorization information.*

In Snowflake, authorization information is encoded in a data structure that has both robust and efficient wire transfer encodings [Riv97]. Thus the unified system is easily adapted for transfer over a variety of existing protocols. In this dissertation, I describe its implementation over HTTP and over Java Remote Method Invocation (RMI). Adapting more protocols, such as NFS and SMTP, to support unified authorization will result in wider applicability of end-to-end authorization.

The four boundaries described above turn up in real systems that accrete from smaller subsystems. Gateway software installed at each boundary maps requests from clients on one side of the boundary to requests for services on the other side. The system described herein allows us, at each boundary, to preserve the flow of authorization information alongside the flow of requests. By allowing gateways to defer authorization decisions to the final resource server when appropriate, and ensuring that resource servers have a full explanation for the authority of the requests they service, I provide applications with end-to-end authorization.

Chapter 9

Infrastructure

In this chapter, we describe the basic infrastructure that implements the formalism of Part IV.

The basic elements of the system are statements and principals. A statement is any assertion, such as "it would be good to read file X," or "Bob speaks for Alice," or "Charlie says Alice speaks for Charlie." A principal is any entity that can make a statement. Examples include the binary representation of a statement itself (that says only what it says), a cryptographic key (that says any message signed by the key), a secure channel (that says any message emanating from the channel), a program (that says its output), and a terminal (that says whatever the user types on it).

A proof of authority, like a proof of a mathematical theorem, is simply a series of statements that incrementally convince the reader of the veracity of the conclusion statement. Of course, in an authorization system, a proof is read by a program, not by a mathematician.

9.1 Statements

Snowflake's implementation of sharing begins with the Java implementation of SPKI by Morcos [Mor98]. It is a useful starting point because not only do I wish to preserve features of SPKI, but SPKI includes a precise and easily extensible specification of the representation of various abstractions. Furthermore, starting with a SPKI implementation offers an easier path to SPKI interoperability.

The restriction imposed on a delegation is specified using *authorization tags* from SPKI. Authorization tags concisely represent infinitely refinable sets, which makes them an attractive format for user-definable restrictions. I replaced Morcos' minimal implementation of authorization tags with a complete one that performs arbitrary intersection operations as described in Chapter 6.

9.2 Principals

SPKI makes a distinction between principals and "subjects," entities that can speak for others but can utter no statements directly, such as threshold (conjunct) principals. My formalism does not make that distinction. Mine also supports new compound principals,

such as the quoting principal of Lampson et al. Therefore, I extended Morcos' Principal class to support SPKI threshold (conjunction) principals and Lampson's quoting principals. When a service reads a request from a communications channel, it associates the request with an appropriate principal object that represents the channel; this principal is the one that "says" the request. The channel may claim to quote some other principal; that assertion is noted by associating the channel with a `Quoting` principal object. The object's `quoter` field is the channel itself, and its `quotee` field is the (possibly compound) principal the channel claims to quote.

9.3 Proofs

I implemented a `Proof` class that represents a structured proof consisting of axioms and theorems of the logic and basic facts (delegations by principals). An instance of `Proof` describes the statement that it proves and can verify itself upon request. While `Proof` objects may be received from untrusted parties, their methods are loaded from a local code base, so that the results of verification are trustworthy. Servers receive from clients instances of the `Proof` class that show the client's authority to request service. Conversely, a server may send a `Proof` to a client to establish its authenticity, that is, to prove its authority to identify itself by some name or to provide some service the client expects.

Proofs can be transmitted as SPKI-style S-expressions or directly transferred between JVMs using Java serialization. No precision is lost in the latter case, since the basic internal structure of every proof component is the Java object corresponding to an S-expression.

SPKI's *sequence* objects also represent a proof of authority. SPKI sequences are barely defined, but they are linear programs apparently intended to run on a simple verifier implemented as a stack machine. When certificates and opcodes are presented to the machine in the correct order, the machine arrives at the desired conclusion [EFL+98]. Figure 9.1 shows an example sequence that proves that the hash of a document H_D represents (speaks for) the name N defined by the client; the client principal here is a public key K_C. In the example, opcodes and operands appearing in the `sequence` on the left mutate the operand stack on the right. I invented three opcodes for this example. The `pop` opcode discards an unused conclusion from the stack, and the `name-monotonicity` and `transitive` opcodes apply theorems of the formal logic to produce new conclusions.

Transmitting proofs rather than SPKI sequences is attractive for three reasons. First, the proofs clearly exhibit their own meaning; to quote Abadi and Needham, "every message should say what it means" [AN96]. Second, the structured proof components map one-to-one to implementation objects that verify each component. The SPKI sequence verifier, in contrast, requires an external mapping to show that the state machine corresponds to correct application of the logic of Chapter 4. Third, it is simple to extract lemmas (subproofs) from structured proofs, allowing the prover to digest proofs into reusable components (Section 9.4).

Figure 9.2 shows the schematic of a structured proof that turns up in my implementation. Compare this example proof with the linear sequence in Figure 9.1. The graph retains the structure of the proof: each statement is directly connected to the statements that support it. Hence, subproofs can be easily extracted from the graph. For example, while the top statement about H_D only applies to a single document, the subproof that $K_S \Rightarrow K_C \cdot N$ is

opcodes	operand stack
`(sequence`	
`(public-key` K_C`)`	K_C
`(do hash sha1)`	$K_C \Rightarrow H_{K_C}$, $H_{K_C} \Rightarrow K_C$
`(do pop)`	$H_{K_C} \Rightarrow K_C$
`(do name-monotonicity` N`)`	$H_{K_C} \cdot N \Rightarrow K_C \cdot N$

`(cert`
`(issuer` $H_{K_C} \cdot N$`)`
`(subject` K_S`))`
	$K_S \Rightarrow H_{K_C} \cdot N$, $H_{K_C} \cdot N \Rightarrow K_C \cdot N$

`(do transitive 0 1)`	$K_S \Rightarrow K_C \cdot N$

`(cert`
`(issuer` K_S`)`
`(subject` H_D`))`
	$H_D \Rightarrow K_S$, $K_S \Rightarrow K_C \cdot N$

`(do transitive 0 1)`	$H_D \Rightarrow K_C \cdot N$
`)`	

Figure 9.1: *A linear sequence (left) designed to advance a stack machine (right) to arrive at an intended conclusion ($H_D \Rightarrow K_C \cdot N$).*

reusable and easily extracted from the proof.

9.4 The prover

A `Prover` object helps Snowflake applications collect and create proofs. It has three tasks: it collects delegations, caches proofs, and constructs new delegations.

A user's application collects delegations from other users. Gateways collect delegations directly from client applications. Both sorts of applications use a `Prover` to maintain their collected delegations in a graph where nodes are principals and edges are delegations from one principal to the next. When a collected delegation is actually a proof involving several component delegations, the `Prover` "digests" the proof into its component parts for storage in the graph. The `Prover` traverses the graph breadth first to find proofs of delegation the application requires.

Whenever it receives or computes a derived delegation proof from components, the `Prover` stores a shortcut edge in the graph holding the proof. These shortcuts form a cache that eliminates most deep traversals of the graph.

When an application controls one or more principals (e.g., by holding the corresponding private key or capability), its `Prover` can store a closure in its graph for the controlled principal. When desired, the `Prover` can not only find existing proofs, but complete new proofs by finding an existing chain of delegations from the controlled principal to the required issuer, then using the closure to delegate to the required subject restricted authority over the controlled principal.

My simple Prover is incomplete, but it is suitable for most authorization tasks applications face. Abadi et al. note that the general access-control problem in the presence of

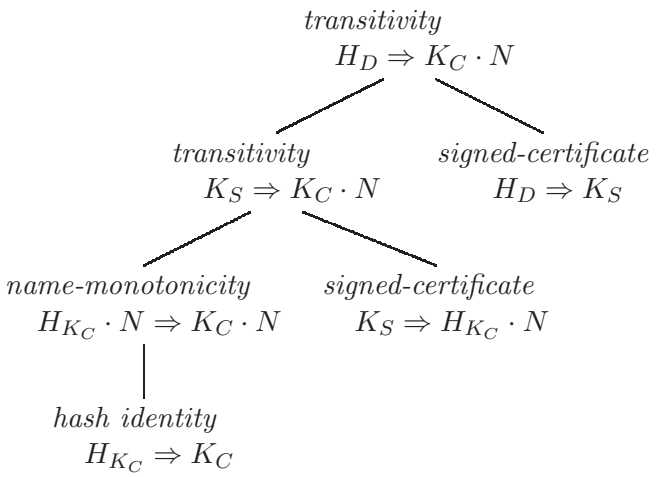


Figure 9.2: *A structured proof that shows that a name defined by a client ($K_C \cdot N$) is bound to a particular document (H_D).*

both conjunction and quoting requires exponential time [ABLP93, p.726]. Elien gives a polynomial-time algorithm for discovering proofs in a graph with only SPKI certificates (no quoting principals) [Eli98]. In the common case, I expect applications to collect authorization information in the course of resolving names, so that proofs are built incrementally with graph traversals of constant depth.

Chapter 10

Channels

With the infrastructure above in place, applications and services have the tools they need to generate, propagate, and analyze authority from the source of a request to its final resource server. The authorization information must be propagated from one program to the next through channels.

When a client makes a request of a server, the server needs some mechanism to ensure that the client really uttered the request. I implemented three such mechanisms: a secure network channel, a local channel vouched for by a trusted authority in the same (virtual) machine, and a signed request. I describe each and discuss how they are represented as principals in my unified system.

10.1 Secure channels

To implement a secure channel, I built a Java implementation of the `ssh` protocol that can interoperate with the Unix `sshd` service [Ylo96]. Then I built Java `ServerSocket` and `Socket` classes based on `ssh` that provide a secure connection. Either end of the connection can query its socket to discover the public key associated with the opposite end.[1]

I plugged my `ssh` sockets into RMI using socket factories. Ssh ensures that the channel is secure between some pair of public keys. To make that guarantee useful, I embody the channel as a principal. Consider the channel in Figure 10.1. To establish the channel, the server (principal P_S) uses public key K_1 and the client (P_C) key K_2 in the key exchange, and together they establish secret key K_{CH} as the symmetric session key.

Suppose a message M emerges from the channel at the server. In the language of the formalism, the `ssh` implementation promises that $M \Rightarrow K_{CH}$. The initial key exchange convinced the server that $K_{CH} \Rightarrow K_2$, and the client may explicitly establish that $K_2 \Rightarrow P_C$. Because $M \Rightarrow K_{CH} \Rightarrow K_2 \Rightarrow P_C$, the server concludes that $M \Rightarrow P_C$, that is, the message says what the client is thinking. My implementation never actually represents the channel key K_{CH} explicitly, since a channel key can only be spoken for by a message emerging

[1] Why did I build an `ssh` implementation? Some have suggested that I use SSL over RMI, which is apparently now fairly practical. When I began this work, however, RMI did not have easily pluggable socket factories, and even once it did, the only open-source SSL implementation I could find did not operate well under RMI.

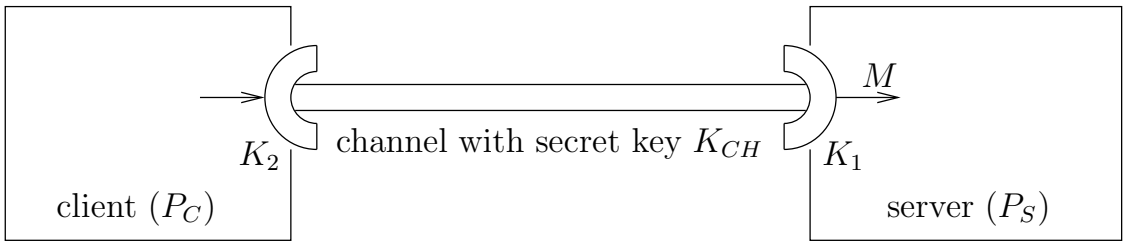


Figure 10.1: *Treating a channel as a principal*

from the channel. Because the same code that establishes $M \Rightarrow K_{CH}$ also establishes $K_{CH} \Rightarrow K_2$, it simply emits the conclusion $M \Rightarrow K_2$.

10.1.1 How channels work

Figure 10.2 illustrates my RMI/`ssh` channel in action. Initially, the server creates an instance of an RMI remote object (a), defines the key K_S that controls it, and binds it to an `SSHContext` (b). The `SSHContext` is associated with the RMI listener socket (c) that will receive incoming requests for the object, and defines the public key (K_1) that will participate in `ssh` session establishment.

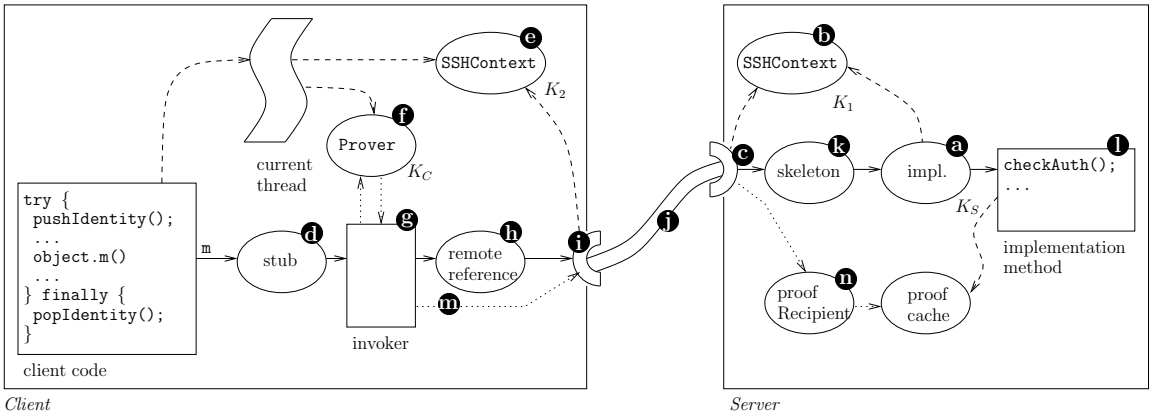


Figure 10.2: *How my `ssh` RMI channel is integrated with Snowflake's authentication service. Dashed arrows ----> represent object references. Solid arrows ——> represent the critical remote call path, and dotted arrows ·····> represent the longer path taken when the server requires fresh proof of the client's authority.*

The client retrieves a stub (d) for the remote object from a name service. To exercise its authority on the object, the client first establishes its identity in thread scope. In a `try ... finally` block, it establishes its own `SSHContext` (e) and a `Prover` (f) that holds its private key K_C. Any method called in the runtime scope of the `try` block will inherit the established authority.

Then the client invokes a method `m` on the remote stub. The remote stub has been mechanically rewritten to wrap its remote invocations with calls to the `invoker` helper method ⓖ. The invoker method makes the usual RMI remote call through the remote reference ⓗ, and the reference creates an `ssh` socket ⓘ using the `SSHSocketFactory` specified in the stub. The `ssh` channel is established ⓙ, and each context learns the public key associated with the opposite end (K_1, K_2). The method call passes through the channel to the skeleton object on the server ⓚ, which forwards the call to the implementation object.

The programmer has prepended to each remote method implementation a call to `checkAuth()` ⓛ. This routine retrieves from the local `SSHContext` the key K_2 associated with the channel that the request arrived on. At this point, the server knows that K_2 **says** `m`. It does not, however, know who K_2 speaks for, so `checkAuth()` throws an `SfNeedAuthorizationException`.

RMI passes the exception back through the channel, where the client's invoker method catches it. The invoker inspects the exception to discover the issuer K_S it must speak for and the minimum restriction set T regarding which it must speak for that issuer. The invoker queries the Prover ⓕ for a proof of the required authority; since the prover controls the client's private key K_C, it can construct a statement to delegate authority from K_C to K_2. The exception carries a special remote `proofRecipient` object; the invoker calls a method on it to pass ⓜ the proof to the server. The `proofRecipient` object ⓝ stores the proof at the server, and returns to the client.

The invoker again sends the original invocation `m` through the remote reference, and the request travels the same path to `checkAuth` on the server. This time, the proof that $K_2 \stackrel{T}{\Rightarrow} K_S$ (via K_C) is available, `checkAuth()` returns without exception, and the remote object's implementation method runs to completion. Future calls encounter no exception as long as the proof at the server remains valid, and are only slowed by the layer of encryption protecting the integrity of the `ssh` channel.

The client programmer need only establish the client's authority at the top of a code block; inside that scope, the `Prover` and the invoker together handle the nitty-gritty of proof generation and authorization. The server programmer must ensure that every Remote implementation method calls `checkAuth()` and defines an appropriate restriction tag; I envision a mechanical tool that automatically injects `checkAuth()` calls and generates reasonable default tags based on the signature of the remote method.

10.1.2 A channel optimization

RMI is not as careful as it should be about reusing existing channels between endpoints. As a result, an application that makes repeated calls to the same server may end up establishing several redundant connections to that server. When each connection spends hundreds of milliseconds performing encryption operations, the waste is unbearable. The preferred solution, ensuring that RMI is optimal in its reuse of connections, requires substantial replumbing of RMI.

Instead, I implemented a session-reuse protocol similar to session caching in SSL. I call my solution "session key borrowing." After each new connection's key exchange, the SSH socket class caches a tuple containing the session secret key and the public key of the opposite endpoint, indexed both by the listening host's address and port and by a secure

hash of the secret key itself. If the RMI subsystem on the connecting host receives a new request to connect to the same listening host and port, instead of engaging in the usual key exchange, the connecting host sends the secure hash of the session key to the listening host. The listening host looks up the session key by its hash, and if it is not found, the host replies by initiating the key exchange protocol. Otherwise, the listening host replies with a message indicating that the borrowed key is acceptable, both ends adopt the borrowed session key and the identity of the opposite endpoint, and the `ssh` protocol is resumed with the client sending a "success" message encrypted with the session key to show that the key works.

I recognize that introducing a new cryptographic protocol is always a risky endeavor. I suggest that this protocol be analyzed carefully before being adopted as a standard or in a production implementation. In any case it would be preferable to repair the higher layer (RMI) to avoid the need for the protocol at all. I now justify my belief that this protocol provides the same invariants as the original application of `ssh`, up to cryptographic weakness.

In a single `ssh` channel implemented over an insecure network, the channel's secrecy is maintained because only the two endpoints know the secret key, and the channel's integrity is maintained because the message CRCs signal modifications by entities that do not know the secret key. The key-borrowing protocol begins by having the client transmit only a secure hash of the channel secret key from another channel. By the cryptographic properties of secure hashes, I am confident that no attacker without knowledge of the secret key could infer the secret key. After this simple transmission and its acknowledgement, the new channel immediately begins operation under the cached secret key. Thus the new channel is essentially operating identically to the previous channel; the fact that its messages are sent over a different (insecure) TCP channel is immaterial.

My protocol is weaker than caching in SSL because I use the same session key in multiple parallel sessions, sacrificing the cryptographic advantages of block chaining or stream ciphers. I propose three solutions to this weakness. Repairing RMI to reuse channels optimally is the most desirable option. Alternatively, a fancier protocol could establish multiple independent session keys during the initial key exchange, amortizing the cost of the public key operation over more secret keying material.

A third alternative would be to identify the first channel established to a given endpoint, and reroute traffic destined for duplicate channels over that original channel. A disadvantage of this approach is dealing with asynchrony in the processes sending and receiving on the original and new (virtual) channels. The `ssh` 2 protocol specifies a protocol for multiplexing virtual channels over a single channel, and that protocol involves data windows to accommodate the asynchrony of the different processes using the virtual channels [YKS+98]. The tactic treats an `ssh` link as a virtual private datagram network, and reimplements TCP-style windowing inside that network.

10.2 Local channels

Setting up a secure network channel is an expensive operation because it involves public-key operations to exchange keys. If a server trusts its host machine enough to run its software, it may as well trust the host to identify parties connected to local IPC channels. Within

my Java environment, I treat the JVM and a few system classes as the trusted host, and bypass encryption when connecting to a server in the same JVM.

In the local case, the `ssh` channel is replaced with a Java "IPC" pipe implemented without any operating system IPC services, and the public keys corresponding to the channel endpoints (K_1 and K_2) are swapped directly. Because it was involved in constructing the key pairs and the keys are stored in immutable objects, the trusted system class knows whether a client holds the private key corresponding to a given public key. Hence when a client is colocated in the same JVM with the server, there is no encryption or system-call overhead associated with the channel, only RMI serialization costs.

10.3 Signed requests

Not all applications can assume that my `ssh`-enhanced version of RMI is available as an RPC mechanism. Indeed, the most visible RPC mechanism on the Internet is HTTP. To facilitate applications that use HTTP, I created a Snowflake version of the HTTP authorization protocol.

HTTP defines a simple, extensible challenge-response authorization mechanism [FHBH+99] diagrammed in Figure 10.3.[2] The client sends an HTTP request to the server. The server replies with a "401 Unauthorized" response, including a `WWW-Authenticate` header describing the method and other parameters of the required authorization. The client resends its request, this time including an `Authorization` header. If the `Authorization` satisfies the server's challenge, the server honors the request and replies with the return value of the operation. Otherwise, the server returns a "403 Forbidden" response to indicate the authorization failure.

HTTP defines two standard authorization methods. In Basic Authentication, the client's `Authorization` header includes a password in cleartext. In Digest Authentication, the server's `WWW-Authenticate` challenge includes a nonce, and the client's `Authorization` header consists of a secure hash of the nonce and the user's password. Both methods *authenticate* the client as the holder of a secret password, and leave *authorization* to an ACL at the server.

In my new method, called Snowflake Authorization, the parameters embedded in the server's `WWW-Authenticate` challenge are the issuer that the client needs to speak for and the minimum restriction set that the delegation must allow. The `Authorization` header in the client's second request simply includes a Snowflake proof that the request speaks for the required issuer regarding the specified restriction set. The subject of the proof is a hash of the request, less the `Authorization` header. Figures 10.4 and 10.5 show examples of these messages.

10.3.1 Signed request optimization

As described, the signed request protocol requires an expensive public-key signature on every request. Since it is common that a client and server interact in more than one

[2]The protocol specification itself uses the terms "authentication" and "authorization" interchangeably; when not quoting the specification, I use the term authorization since it best describes my goal. We reduce the authentication problem to authorization in Section 10.3.2.

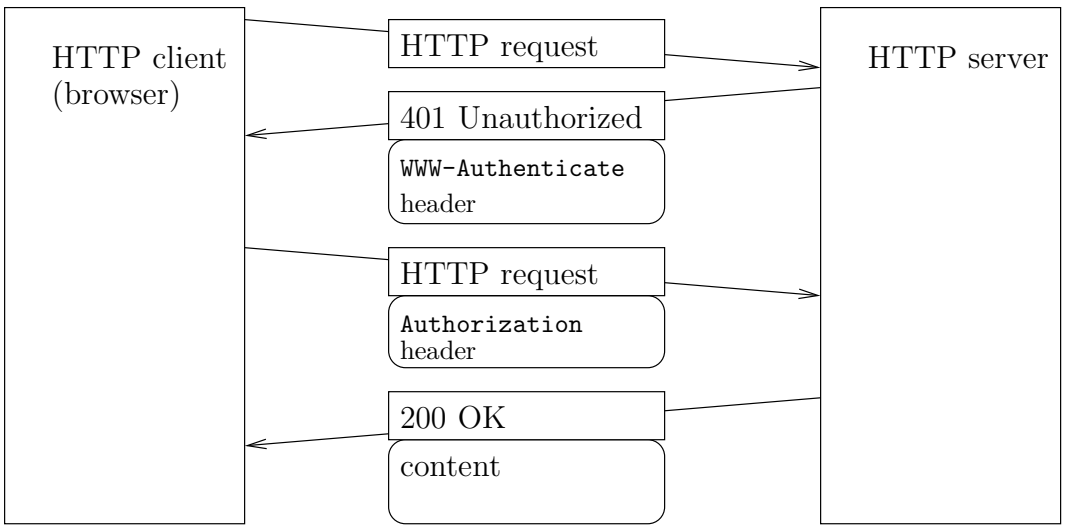


Figure 10.3: *The HTTP authorization protocol.*

request, I wish to amortize the signature operation over many requests. One approach is to send multiple requests over a stream that guarantees integrity; among other things, this is a property SSL can provide.

We chose to implement an alternative protocol with a different set of performance and security characteristics. When sending in a signed request, a client may specify a public key under which it wishes to receive an encrypted secret message authentication code (MAC). If the server agrees, it generates a random authenticator, encrypts it with the specified public key, and returns it to the client. Then the client signs a certificate $H_{MAC} \Rightarrow K_C$ delegating its authority to the secret MAC, named by its hash so that the certificate need not remain secret. For future requests, instead of signing the specific request, the client need only hash the request together with the MAC. This hash shows that the text of the request speaks for the MAC: $H_{request} \Rightarrow H_{MAC}$.

The protocol is fast because secure hashing has replaced expensive public-key signatures. It requires that the client authenticate the server, or else a man-in-the-middle can supply the client with a bogus MAC. The client will also likely want to make the statement $H_{MAC} \stackrel{V}{\Rightarrow} K_C$ with a short validity interval V to limit the trust extended to the MAC. This protocol highlights one of the attractive properties of our unified authorization protocol: it can make any authorization trust relationship explicit, auditable, and visible end-to-end. The final server receiving a request can see that my home brew MAC protocol was involved in the transaction.

10.3.2 Server authorization

Often a client also wants to verify that it is communicating with the "right" server. The notion of "right" can be as simple as the server speaking for the client's idea of a well-known name like `www.dartmouth.edu`, but in general the real question is still one of authorization:

```
HTTP/1.0 401 UNAUTHORIZED
Content-Type:  text/html
MIME-Version:  1.0
Server:  MortBay-Jetty-2.3.3
Date:  Sat, 08 Apr 2000 15:18:47 GMT
WWW-Authenticate:  SnowflakeProof Authorize-Client
Sf-ServiceIssuer:  (hash md5 |ehtQYd4EpQXOa/ON6Smesg==|)
Sf-MinimumTag:  (tag
  (web (method GET)
    (service |Sm9uJ3MgUHJvdGVjdGVkIFNlcnZpY2U=|)
    (resourcePath "")))
Connection:  close
```

Figure 10.4: *An HTTP authorization challenge message from a Snowflake server. It indicates the method, the required resource issuer, and the minimum restriction of a delegation that must be proven.*

Does this server have the right to claim authority about Dartmouth's course list? Does that server have authority to receive my e-mail?

I defined a simple protocol by which an HTTP server can communicate its authority to serve a document to a client. The server P_S simply constructs a proof that the document speaks for the server's name for the document $P_S \cdot N_{document}$. The server returns the proof in the `Sf-DocSpeaksForServerName` header that precedes the document. The client queries its `Prover` to see who the server speaks for. Perhaps the document was returned as a result of a user clicking a link with embedded authorization information, in which case the client expects to find a proof that the server speaks with that authority; or perhaps the client just wishes to find any locally-trusted name for the document to report to the user[3]. Notice that although the client and server are in opposite roles in the trust relationship, the burden of proof is still at the client: the protocol design assumes that clients have more resources, and that the client's history of requests may provide hints for efficient proof.

My server authorization protocol leaves a lot to be desired; for example, because it does not attest to the authority of the complete transaction including headers, it cannot help with the man-in-the-middle attack described in Section 10.3.1. An alternate tactic would have the server deliver the document over a stream with integrity (one of the features of SSL), and delegate its authority to the stream. End-to-end authorization still occurs: the client arrives at a proof showing the participation of either the document hash (in my protocol) or the stream.

10.3.3 Server implementation

I implement the server side of the protocol as an abstract Java Servlet `ProtectedServlet` [Mos98]. Concrete implementations extend `ProtectedServlet` with a method that maps a request to an issuer and minimum restriction set, as well as the service implemen-

[3]My current implementation only supports the latter naming operation.

```
GET /files/ HTTP/1.0
User-Agent:  Mozilla/4.7 [en] (X11; U; SunOS 5.7 sun4u)
Accept:  image/gif, image/x-xbitmap, image/jpeg, */*
Authorization:  SnowflakeProof (proof
  two-step
  (proof
    signed-certificate
    (signed-certificate
      (cert
        (issuer (hash md5 |ehtQYd4EpQXOa/ON6Smesg==|))
        ...
        ClkEKqKOL+AYBYSsz7456NkeLwyv4CdxrlnJKueASaxEmN
        HJstLleUaTS8OSKQ7xZ45zNF5mFDz|))))
```

Figure 10.5: *A response message from a Snowflake proxy. It contains a proof of the required delegation. Most of the proof has been omitted to save space.*

tation method that maps a request to a response. When each request arrives, the `ProtectedServlet` ensures that appropriate authorization has been supplied, and if not, constructs and returns the "401 Unauthorized" response to the client. I plug my servlets into the Jetty open-source web server,[4] but they should operate correctly in any web server that supports servlets.

10.3.4 Client implementation

In as much as HTTP is a browser protocol, the client side of Snowflake Authorization belongs inside the client's web browser. For practical reasons, I instead implemented the protocol in an HTTP proxy that intercepts all of the browser's outbound requests. Using this technique, I must take care to ensure that the proxy is accessed by only a single user. The first step is to install the proxy on the same machine as the browser, and arrange for it to only accept requests from the local host. The second step, that I have not yet implemented, is to ensure that every request comes from the same user's browser, perhaps using `identd` [Joh93].

When the proxy receives a request from the browser, it forwards the request on to the destination server. The proxy inspects the response headers before returning them to the browser. If the response has a "401 Unauthorized" result code and demands Snowflake authorization, the proxy uses a `Prover` to construct the required `Proof`. The proof shows a delegation from the required issuer to the hash of the outgoing request message; it is inserted into the outbound message in an `Authorization` header, and the request is resubmitted to the server. Generally, the server's second reply is the result of the desired operation. If the proxy cannot generate a proof to meet the server's challenge, or if the second request is again rejected, the proxy converts the response to a "403 Forbidden" message to prevent broken browsers from initiating the Basic Authentication protocol.

[4] `http://www.mortbay.com/software/Jetty.html`

The user needs a user interface to the proxy to manage its collection of keys and delegations. The proxy sees all outbound requests using the HTTP protocol, so it creates a virtual server at the URL `http://security.localhost/`. Any request beginning with that URL is mapped to a handler inside the proxy that provides an HTML interface to the proxy's keys and delegations. Through this interface, the user can create a new private key pair, import principal identities and delegations, and delegate his authority to others.

The proxy keeps a history of recently visited pages, and offers the user commands to delegate authority over any Snowflake-protected pages to other users. Hence to share a Snowflake-protected web page, all one needs to do is visit the web page, then access the history list at `security.localhost` and click "delegate authorization for this page." The user selects from the proxy's list of known principals the recipient of the delegation, and the proxy outputs a snippet of HTML that the user can deliver (for example, by email) to the recipient. The HTML snippet is actually a single link to a special URL at the virtual `security.localhost`.

When the recipient clicks the link, his proxy intercepts the URL embedded in the HTML snippet, and extracts from the URL both the proof of authorization needed to access the resource and the URL of the resource itself. The proxy digests the proof, redirects the recipient's browser to the resource URL, and the authentication protocol proceeds from the recipient's proxy just as it did for the original user. Notice that the client's request pattern reveals the required proof of the user's authority just as it can reveal the required proof of the server's authority (see Section 10.3.2).

Chapter 11

Applications

I built three applications to demonstrate the Snowflake architecture for sharing.

11.1 Protected web server

The first application I built that uses Snowflake's sharing architecture is simply a protected web file server. One user initially controls the file server (by specifying the hash of his public key when starting up the server); he may delegate permission to read subtrees or individual files from the server to others using the mechanisms described above.

11.2 Protected database

The second application attaches Snowflake security to a relational email database. The original database server accepts insert, update, and select requests as RMI invocations on a Remote `Database` object, and returns the results of the query as serialized objects from the database. Adapting the application to Snowflake required only minimal changes. I modified the database instance constructor to use a `SshSocketFactory` so that all connections to the object use my `ssh` secure channels. Then, I prepended each implementation of a method in the remote interface with a call to the `checkAuth()` method. The database clients required only a modification to their initialization code to install an `SSHContext` and a `Prover`.

11.3 Quoting protocol gateway

The third application is a protocol gateway that provides an HTML over HTTP front-end to the email database. I built this application to demonstrate the power of the quoting principal. The protocol gateway has its own identity, and quotes each user accessing it to ensure that the user only sees those parts of the database he is authorized to see.

A transaction begins when the client (C) sends an unauthorized request to the gateway (G). The gateway attempts to access the database (D), but the RMI authorization fails because the gateway has no authority. The gateway sees an exception that indicates the required issuer D and restriction set (T). The gateway generates a "401 Unauthorized" Snowflake Authorization HTTP response, and in that response indicates it needs a proof

that $G|P \stackrel{T}{\Rightarrow} D$. By $G|P$ the gateway means it needs a proof of authority that the gateway quoting the client speaks for the database. The gateway uses P as a "pseudo-principal" to stand for the client's identity, since it has no information about the client yet.

The client proxy now knows it needs to delegate its authority to the principal "gateway quoting client," $G|C$. The client proxy generates the proof and resubmits its original request, along with the proof, to the gateway. The gateway digests the new proof and forwards the request to the database server. This time, the automatic RMI authorization protocol of Section 10.1.1 finds the proof in the gateway's `Prover`, and the database fulfills the request. The gateway builds an HTML interface from the database results for presentation to the user. Subsequent requests are accepted without so much fanfare, since the database server holds the appropriate proof of delegation.

The quoting gateway is a motivating application because it spans each of the four boundaries discussed in Section 8. My gateway operates identically whether the client and the server are in the same administrative domain or different ones. It can be colocated with the server, in which case its RMI transactions automatically avoid encryption overhead by using the local channels of Section 10.2. The gateway constructs a view of an e-mail message from several rows and tables of a relational database, and so introduces a level of abstraction above the server resource. Finally, the gateway spans protocols by connecting an HTTP-speaking web browser with an RMI-speaking database server. Despite each of these boundaries, the gateway preserves the entire chain of authority that connects the client to the final server, enabling the server to make a fully-informed access-control decision.

The gateway services requests from multiple mutually-untrusting clients, but it makes no access-control decisions, and hence minimizes the likelihood that one client can exploit the gateway to use the authority of another. The gateway avoids making access-control decisions simply by carefully quoting the appropriate client in requests forwarded to the server. Therefore, casual inspection can convince us that uncompromised gateway code protects the authority of each of its clients. As always, a client can restrict its delegation to the gateway to limit its exposure in the event that the gateway's code is compromised.

Chapter 12

Measurement

To better understand the costs of the Snowflake authorization model, and how they compare to costs of related systems, I timed the performance of my Snowflake-enhanced HTTP implementation. I also timed standard HTTP servers, with and without SSL support.

12.1 Experimental methodology

The performance numbers presented in this chapter were acquired in the environment described in Table 12.1. My experiments take two forms: *setup and bandwidth* and *setup and per-connection*. In the first form, I vary the length of a file transferred and extract from a linear regression the cost of initiating the request (the y-intercept) and the cost per megabyte to transfer the request data (the slope). In the second form, I vary the number of requests made after establishing a single connection. From a linear regression I extract the cost of establishing the connection (y-intercept) and the cost per request to send an individual request over the connection (slope).

In each experiment, I measure the wall-clock time required to complete 10 to 1000 repetitions of the operation, enough to make the experiment run on the order of seconds. I repeat each run (a fixed combination of parameters) ten times, and discarded the first time to discount startup costs such as loading class files. Where they are within three orders of

processor	Sun Ultra 5, 270 MHz, 128 MB RAM
network	shared 10 Mbps Ethernet
operating system	Solaris 2.7
Apache web server	version 1.3.12
OpenSSL	version 0.9.5
Java virtual machine	JDK 1.2.2, locally compiled, `sunwjit`, green threads
PureTLS	0.9b1
Cryptix	3.1.1

Table 12.1: *Hardware and software configurations used in my experiments.*

magnitude of the y-intercept quantity, I report the standard deviation of errors (σ), from which one can infer the amount of noise over the linear process. When the quality of fit $R^2 < 0.99$, we report it as well; experiments with small R^2 have shallow slopes, and hence the linear model does not contribute much information beyond the constant line at the mean of the y observations. I computed 95% confidence intervals on the linear regression parameters and found them always to be at least three orders of magnitude smaller than the quantity they bound. I round all values to two significant figures. I visually inspected each scatter plot to ensure that the linear models are reasonable; the plots appear in Appendix E.

Sometimes the ten trials of an experiment with a single collection of values exhibited a significant coefficient of variation (C.V.). I checked the C.V.s for each individual experiment, and re-ran those with C.V.s greater than 0.1. My understanding of the system and my inspection of the data give me confidence that by doing so, I have eliminated noise due to asynchronous events such as `cron` jobs or external network traffic, rather than rubbing out observations of rare events in my experimental system.

12.2 RMI authorization with Snowflake

In this section, I quantify my implementation of Snowflake authorization over Java remote method invocation as described in Section 10.1. The test operation is a Remote object that returns the contents of a file. According to Table 12.2, the setup time for this operation in basic RMI is 4.8 ms, and the compute-bound data-copy overhead is 990 ms/MB. Performing the same operation over an open `ssh` channel increases the fixed cost to 13 ms, and the data copy cost to 6100 ms/MB. Adding the Snowflake protocol brings the total to 18 ms: the extra work is the server's `checkAuth()` call, which retrieves the caller's public key, finds a cached proof for that subject, and sees that the proof has already been verified.

protocol	request ms	copy ms/MB	σ ms	R^2
RMI	**4.8**	**990**	**5.7**	**0.96**
remote	6.9	1100	1.7	
ssh alone	**13**	**6100**	**4.4**	
remote	13	3900	6.6	
Snowflake	**18**	**6100**	**6.7**	
remote	16	3900	5.3	

Table 12.2: *A* setup and bandwidth *experiment for RMI. Results from experiments where client and server are on separate hosts appear on lines marked* remote*.*

The above results are from a warm connection. Table 12.3 presents results from cold connection caches. It costs 470 ms to establish a new connection, reflecting the public-key operation the client performs to delegate its authority to the channel. When the client caches the delegation but the server forgets its copy after each use, we learn that the server spends about 190 ms parsing and verifying the proof from the client.

proofs cached	request ms	copy ms/MB	σ ms
none	**470**	**6200**	**14**
remote	410	3800	5.1
client	**190**	**6400**	**19**
remote	140	3800	5.5
server	**20**	**6100**	**7.1**
remote	17	3800	4.4

Table 12.3: *Disabling caches reveals the cost of proof generation, transmission and verification.*

client	server	request ms	copy ms/MB	σ ms	R^2
C	**Apache**	**4.6**	**61**	**0.27**	**0.98**
	remote	4.8	960	0.56	
Java	**Apache**	**11**	**93**	**0.92**	**0.89**
	remote	10	950	0.51	
	Java	**17**	**180**	**1.0**	**0.96**
	remote	17	980	0.70	
	Jetty	**25**	**240**	**2.3**	**0.90**
	remote	24	1000	6.1	0.96

Table 12.4: *Performance baselines for HTTP 1.0.*

12.3 HTTP authorization with Snowflake

In this section, I quantify my implementation of Snowflake authorization over the HTTP protocol as described in Section 10.3.

12.3.1 Client baseline

First, I measured the time it takes to access an Apache web server with a C client and with my Java client using an HTTP stream-processing package (part of the Jetty web server). The results are presented in Table 12.4. With the C client, a connection costs about 4.6 ms, each megabyte of data transferred locally costs about 61 ms/MB more, and as one might expect, each megabyte transferred over a shared 10 Mbps Ethernet segment costs about 960 ms/MB. From my Java client, the connection costs 11 ms plus 93 ms/MB locally and still around 950 ms/MB remote. This experiment establishes the baseline cost of my Java timing client that I use in all of the further tests.

12.3.2 Server baseline

Next, I compared the Apache server with both a trivial server written in Java and the Jetty web server, a full-fledged Java web server with rich support for the features of HTTP

server	connect ms	request ms/req	σ ms
Apache	**4.3**	**4.6**	**1.6**
remote	3.6	4.5	0.80
Jetty	**10**	**22**	**2.0**
remote	16	28	9.9

Table 12.5: *A* setup and per-connection *experiment that reveals the cost of a single HTTP/1.1 request. By varying the number of requests per connection, a linear regression provides the setup cost of a connection and the cost of each request thereafter.*

streams.

Taking the jump to Java adds about 6 ms to the cost of a connection, and doubles the per-MB cost. These costs likely reflect the fact that while Apache uses C system call interfaces directly, Java's I/O interfaces are overhead-incurring wrappers linked to the same C glue routines.

My implementation is based on the convenient services and clean factorization provided by the Jetty web server. Using Jetty adds another 8 ms, and increases the data transfer cost another 33%. These costs reflect Jetty's fairly complete request parser and Jetty's stream filters needed to support the various HTTP models for content length and chunking. The Jetty implementation could certainly stand some optimization, but I present the Jetty costs as a baseline since my implementation is based on Jetty.

12.3.3 Network baseline

I ran the same tests on a shared segment of 10Mbps Ethernet, with no intervening router. Not surprisingly, the per-connection costs in the remote case are very similar to those in the local case, since a connection setup only incurs a few sub-microsecond latencies. Per-megabyte costs snap to about one second per megabyte, reflecting the 10Mbps bandwidth of the cable.

12.3.4 Snowflake costs

Now that I have established a baseline, what are the costs of our implementation of Snowflake services over HTTP? As shown in Table 12.6, a fresh connection costs around 380 ms to establish, including making a public-key signature of the outbound request. Replaying the same request costs only about 81 ms, since the `Prover` has the required proof in cache. Using the MAC optimization from Section 10.3.1 brings the cost for authorizing a new request to 110 ms. The extra 28 ms over an identical request reflects the fact that we do all the same work (hashing the request), plus an additional hash to include the MAC, plus parsing and verifying the proof at the server.

Table 12.7 presents our measurements of the server document authentication protocol described in Section 10.3.2. In this experiment, the client presented identical requests. Signing a document $\langle sign,ignore \rangle$ is dominated by a public-key operation, hence the 430 ms connection cost. It also incurs a hashing operation that applies to the entire document,

requests	request ms	copy ms/MB	σ ms	R^2
signed	**380**	**190**	**10**	**0.21**
remote	380	980	8.8	0.91
identical	**81**	**300**	**1.9**	**0.95**
remote	85	920	13	0.81
MAC opt	**110**	**300**	**1.6**	**0.97**
remote	110	990	1.7	

Table 12.6: *Snowflake HTTP client authorization performance. The three major cases are new requests that require a public-key signature, a repeated request with a cached proof, and new requests that exploit the MAC protocol.*

server behavior	client behavior	request ms	copy ms/MB	σ ms	R^2
none	**ignore**	**86**	**290**	**2.2**	**0.96**
	remote	84	960	1.9	
sign	**ignore**	**430**	**760**	**7.0**	**0.93**
	remote	430	1400	9.0	0.97
cache	**ignore**	**99**	**300**	**1.9**	**0.97**
	remote	100	910	5.5	0.97
sign	**verify**	**490**	**1200**	**9.6**	**0.95**
	remote	480	1800	11	0.97
cache	**verify**	**160**	**560**	**14**	**0.68**
	remote	150	1300	5.2	

Table 12.7: *Snowflake HTTP server authorization performance. The experiments vary according to whether the server signed the returned document, used a cached proof, or sent no proof at all, and whether the client verified the document's authenticity.*

increasing the copy cost. Caching document signatures ⟨*cache*,*ignore*⟩ eliminates both expenses; the increase from 86 ms to 99 ms is due to formatting the cached proof for inclusion in the reply header.

The final two cases correspond to the previous two, but now the client actually parses and verifies the signature on the document and finds a proof of the server's authority in its `Prover`. The per-document cost increases by about 60 ms for parsing the server's proof and querying the `Prover` to complete the proof and yield a name binding. The data copy cost increases since the client must also hash the document. The differences in copy cost in the remote cases appear because hashing operations and network delivery overlap.

12.3.5 Comparison with SSL costs

How do my channels compare in cost with alternative approaches to authorization? The obvious alternative is the Secure Sockets Layer (SSL) or its offspring, the Transport Layer

server	cache	request ms	copy ms/MB	σ ms
Apache	**none**	**670**	**14000**	**15**
	remote	710	11000	43
	context	**650**	**15000**	**36**
	remote	680	11000	8.5
	session	**190**	**14000**	**39**
	remote	190	14000	39
Jetty	**none**	**420**	**24000**	**31**
	remote	440	12000	23
	context	**390**	**25000**	**23**
	remote	420	12000	37
	session	**290**	**24000**	**45**
	remote	290	24000	45

Table 12.8: *The SSL* setup and bandwidth *experiment. In the* context *case, we discard cached sessions; in the* session *case, we allow SSL to cache and reuse sessions.*

server	connect ms	request ms/req	σ ms	R^2
Apache	**180**	**13**	**18**	**0.97**
remote	210	16	38	0.90
Jetty	**260**	**47**	**45**	**0.98**
remote	240	46	8.5	

Table 12.9: *The cost of a single request over SSL. The first column shows cost of establishing a connection with session caching, and the second column is the cost to send a 100-byte request over the connection.*

Security (TLS) protocol. I measured the costs associated with SSL in two server configurations: a C implementation, Apache plus OpenSSL; and a pure Java implementation, Jetty plus PureTLS (SSL protocol) plus Cryptix (cryptographic primitive implementation). The client was always our Java client with PureTLS and Cryptix. For posterity, I also measured my simple Java web server with PureTLS and Cryptix, and found that in fact the 15 ms connection setup difference between Jetty and our simple server was lost in the noise of the long SSL connection establishment. We used 1024-bit RSA keys on both the server and client in the SSL experiments so they would compare with the Snowflake measurements.

With the Apache server, an SSL connection takes 650 ms to establish a session to a new server once the client's local context has been initialized, and 190 ms to rebuild using a cached session. A cached session reuses secret-key encryption parameters, saving two expensive public-key operations. Surprisingly, The Jetty server takes less time (390 ms) to set up a new connection. Session reuse with Jetty takes 290 ms.

12.4 Gateway authorization

My informal test of the gateway application show that it has a latency of about 770 ms. This exorbitant cost reflects the fact that the gateway transaction always involves two public-key signatures. I have not optimized the protocol to eliminate public-key encryptions in the common case. The optimizations used in the other protocols do not apply directly because of the use of quoting in the gateway.

12.5 Observations

I hypothesize that the Snowflake authorization model is not prohibitively expensive. In fact, because it can subsume many hop-by-hop authorization models, it allows applications and users to make security-performance tradeoffs freely by selecting alternate hop-by-hop authorization protocols and plugging them into the same authorization framework.

Do my measurements support my hypothesis? Unfortunately, since my implementation is unoptimized and built on top of slow libraries, the numbers do not support my hypothesis unequivocally. By comparing them with baseline experiments, however, I believe I can make a strong case for the hypothesis. In the next two sections, I examine the two parts of my hypothesis. In Section 12.5.3, I argue that an optimized Snowflake promises to be competitive with existing hop-by-hop protocols. I summarize our observations with some lessons learned in the process of measuring our implementation.

12.5.1 Comparable operations

Snowflake-enhanced protocols are not inherently more expensive than other protocols with similar guarantees. The measurements indicate that Snowflake protocols perform similar encryption steps. SSL spends about 400 ms starting up, as does Snowflake. SSL can complete a request over an established channel in about 50 ms. With my MAC optimization, a Snowflake request takes about 110 ms. As I discuss in Section 12.5.3, we have observed that most of the difference is attributable to slow libraries.

Both SSL and Snowflake engage in similar operations. SSL verifies message authenticity with symmetric-key decryption and a CRC; Snowflake does the same with an MD5 hash. Regardless of protocol, the server parses and processes the request and returns the reply. SSL checksums and encrypts the reply; Snowflake securely hashes the reply document. In both cases, the client uses a corresponding operation to verify the reply. Because the expensive cryptographic operations are comparable, one expects optimized implementations to perform comparably.

The likely sources of additional overhead in Snowflake are time spent walking the proof graph and memory consumed maintaining cached proofs. My experiments do not explore that space in depth, but as I describe in Section 10.3.4, proofs are usually constructed incrementally while walking the name graph, an operation driven by the client user or application.

12.5.2 The performance-security tradeoff

By comparing our authorized-request protocol to SSL I somewhat compare apples and oranges, for the protocols make different performance-security tradeoffs. That, however, is part of my point. With Snowflake, one is free to choose an established hop-by-hop protocol or to develop a new one. By stating in my logic the authorization promises the protocol makes, one can integrate the protocol into Snowflake's end-to-end authorization model. Conceivably, new protocols can be dynamically integrated into existing Snowflake-aware applications; in other cases, a protocol-translating gateway can introduce the new protocol to the distributed system without hiding authorization information from the underlying application.

12.5.3 Slow libraries

Our formal measurements and informal tests indicate that a large fraction of Snowflake's cost is needless overhead. As Table 12.4 shows, my baseline HTTP measurements indicate that using Java and the convenient Jetty web server incurs substantial overhead (250%). Furthermore, Table 12.8 shows that the Java encryption library Cryptix imposes a substantial bandwidth overhead.

What surprised me most was the overhead of the SPKI implementation on which we built Snowflake's objects. In informal tests, parsing a 2 KB S-expression from a string takes around 20 ms, and converting the resulting tree into typed Java objects takes another 20 ms. Indeed, even basic operations like `hashCode()` took over 5 ms until I made simple fixes that improved the performance an order of magnitude. There is no reason a well-implemented library should spend milliseconds parsing short strings in a simple language; and 40+ ms delays such as these explain much of the difference between Snowflake's warm-connection performance and that of simple HTTP transactions. I am considering a full reimplementation of the SPKI libraries and adopting native crypto providers to eliminate a large fraction of the inefficiencies in our implementation and make it achieve production-system performance.

12.5.4 Performance lessons

While developing these experiments, I learned two lessons about performance and measurement. First, do not forget to close sockets when they are no longer in use, for extra file descriptors slow things down [BM98]. Second, when latency matters, be careful to disable Nagle's algorithm, which delays delivery of packets in hopes that more data will be arrived that can be aggregated. Oddly, Nagle's algorithm is even used on connections to the local host. I am also surprised that the interface to Nagle's algorithm is a single bit on the file descriptor. The algorithm is analogous to turning off buffering in the C `stdio` library; why not supply a `flush` call instead to give the application more control?

Chapter 13

Qualities of Snowflake sharing and security

I argue here that the architecture and implementation described in the preceding chapters satisfy the important qualities outlined in Section 3.2.

13.1 Consistent sharing

Users share resources with one another the same way, regardless of whether the parties involved are in the same administrative domain or different ones. In any case, the issuer ascertains the subject's identity as a principal, asks the sharing tool to construct a delegation to the subject based on his authority, and communicates the name of the resource together with the constructed delegation to the subject.

13.2 Transitive delegation

A user may share any resource he can access with another user simply by constructing a delegation of his authority to that user.

13.3 Restricted delegation

Users share resources the same way regardless of whether the resource is from "first principles" or is a resource received by restricted delegation from another user. If a resource has an abstract notion of divisibility, my architecture provides a way for the user to express restriction of that resource when sharing it with another.

13.4 Auditable access control

Whenever a server answers a request, it does so only after verifying a proof that the request, however indirectly, speaks for the owner of the resource with respect to some restriction.

Therefore, the owner always has access to the chain of parties that authorized the request, and may verify that those delegates are behaving responsibly.

Suppose a user downstream from a resource server wishes to audit requests made using a delegation he has issued. With conjunction, he can express a delegation that requires the agreement of an online "auditing oracle," ensuring that the oracle has an opportunity to inspect any transactions that rely on the delegation.

13.5 User cost of sharing

When a user Alice wants to share a resource, she thinks "I want to share part of resource X with user Bob." Unix groups make it difficult to express sharing with a simple entity such as "user Bob." Capabilities express the sharing, but do not encode the intended recipient of the delegation. Access-control lists make it easy to express "user Bob," but only if Alice controls the ACL for X. The speaks-for relation in the Calculus for Access Control expresses transitive sharing, but since it relies on ACLs, has the same limitation. Unlike the alternatives, Snowflake's restricted delegation captures the user's concept of sharing precisely.

13.6 User cost of administration

One attractive facet of the user learning curve in Snowflake is that mechanism is decoupled from administrative boundaries. Hence a system administrator uses the same tools as an ordinary user. An administrator performs the same tasks that regular users do, resource naming and resource sharing; the only difference is where resources come from. In the common case, an administrator is delegated resources directly by the hardware itself; users are delegated resources by the administrator. The consequence is that a user can smoothly graduate to the role of administrator.

13.7 Performance

Recall from Section 12.5 that one way to consider the Snowflake sharing architecture is as a unification of several more specific mechanisms. To that extent, its performance tracks that of particular mechanisms and depends upon the environment in which it is used. For example, if one is using the Snowflake model on a local machine with a security kernel trusted to provide valid authorization information, there is no need for expensive encryption, and Snowflake exploits that scenario.

Recall Chapter 12, where I measure the impact of Snowflake in a few concrete applications, and argue that its performance indeed tracks that of more specific authorization protocols. The advantage of the Snowflake model over a given hop-by-hop authorization protocol is that it can be used consistently regardless of whether resources are separated by administrative boundaries, and in every situation it provides end-to-end authorization information.

13.8 Formal model of sharing

One of the greatest strengths of the Snowflake architecture is its foundation in an unambiguous formal model. In Sections 7.8 and 7.9 I show how my model can support or warn against proposed extensions. An unambiguous model gives us confidence that we understand the ramifications of the logic, so that the worst-case behavior of the system in the presence of an adversary is less likely to surprise us.

Part VI

Summary

Chapter 14

Related work

In this chapter, I discuss work related to my goals, either because it shares the same goals or because it provides a path to reaching those goals. I examine in turn microkernel-based operating systems, extensible operating systems, single-system-image clusters, middleware infrastructure, distributed file systems, and worldwide systems.

14.1 Microkernels

Microkernels move services from an administrator-controlled kernel into user processes, where they potentially may be provided by not just a distinguished administrator, but by multiple users. The change in mechanism supports a more egalitarian resource distribution model from which Snowflake derives inspiration.

The Mach project pioneered microkernel architecture. Mach services are provided by active objects (Mach tasks) communicating via Mach's IPC primitives. IPC is rather expensive in Mach, so Mach active objects are typically large-grained. Indeed, most Mach-based operating system "personalities" implement most of the OS functionality in a single server task. Even the Mach kernel itself is still fairly large compared with other microkernels: it provides virtual memory, a rich IPC primitive, device drivers, and a file system [BGJ+92, Loe92, BRS+85]. Tanenbaum's survey compares Mach with the Chorus and Amoeba microkernels [Tan95]. The GNU Hurd is based on a Mach microkernel, but does a better job of factoring services into multiple object servers [Fou96].

The Flex project advocates a model called "Recursive Virtual Machines" [FS96, FHL+96]. This model is facilitated by a virtual machine interface in which each resource (memory, CPU scheduling) can be administered hierarchically. While a user may only be given some fraction of the resources owned by an administrator, the user still retains the same level of expressive power to redistribute those resources. Hence, Flex' mechanisms may support the user-centric naming and sharing I pursue.

Grasshopper is a general-purpose microkernel with a hierarchical memory-mapping model. Because any page in a process can be both supplied by a parent process and supplied to a child process, Grasshopper supports user-centric resource naming through its memory model. Its memory-mapping facility is used to provide pervasive services such as persistence [DdBF+94, LRD95].

14.2 Retrofitting existing architectures

Microkernels rewire the system to make all resource providers user processes, and hence take a first step toward user-centric resource naming and sharing. In contrast, other systems hold to a conventional administrator-centric architecture, but provide users with an escape route for providing user-specific functionality. The assumption in these approaches is that administrator-centric behavior is appropriate most of the time, so that the hooks need be used only occasionally. Unfortunately, that philosophy means that these systems retain the notion of an administrative domain.

VINO and SPIN are operating systems based on a conventional monolithic kernel plus an extension mechanism that allows user processes to install kernel extensions that modify the system interface [SESS96, BCE+95, BSP+95]. While this approach gives user processes the ability to reshape the system according to their own needs, it does not enable users to share resources with the same mechanisms by which the administrator initially doles resources out to users.

Other systems retrofit a system of conventional heritage with extensions to allow user-specific naming. Portals in BSD 4.4 reflect some naming operations out to user processes, but they still respect the concept of a central system administrator [SP95]. Bershad and Pinkerton's Watchdogs and Neuman's Prospero insert hooks into the operating system kernel to allow user-specific extensions to the system naming interface [BP88, Neu92]. Alexandrov's UFO and Jones' Interposition Agents use existing hooks to implement user-level API extensions [AISS98, Jon93].

Although they preserve a distinguished notion of administrator, these systems do allow users to redefine how resources are supplied to their processes, enabling users to run existing applications under a new resource access model. Indeed, I use an emulation layer based on the UFO approach to adapt existing applications to use Snowflake services.

14.3 Single-system-image clusters

Single-system-image clusters are clusters of machines running kernels that communicate to provide users with the illusion of a single machine. Any resource anywhere in the cluster is accessible uniformly and easily. The cluster is still an administrative unit, however; accessing resources beyond the cluster is nonuniform and hence more difficult. I mention single-system-image clusters because they have similar goals for users (uniform and easy access to "all" resources), but they contrast in an important way: they define the scope of "all" resources to be those in an administrative domain, not all resources potentially accessible to a user.

Examples of single-system-image clusters include Spring (based on intracluster remote method invocation), Sprite and Amoeba (based on intracluster remote procedure call), and Plan 9 (based on intracluster file system operations) [MGH+94, OCD+88, TvRvS+90, PPD+95]. We discuss Plan 9 in more detail in Section 2.2.2. GLUnix uses middleware to create a cluster-wide global namespace on conventional Unix hosts [GPR+98]. Single-address-space operating systems implement a single-system-image cluster by accessing services via distributed shared memory. Examples include the Apollo DOMAIN system, Opal, Angel, Mungi, and Hurricane [RLML86, CLFL94, MWSK94, HEV+98, UKS95]. The dis-

tributed shared memory approach faces even greater restriction in scale and distribution than that based on message passing.

The Grapevine system was an early example of an enterprise-wide single-system image. It provided coherent naming and simple authentication for distributed resources such as e-mail and printing. Grapevine's hierarchical structure treated the entire enterprise as a single administrative domain [BLNS82, SBN84]. Xerox' production Clearinghouse distributed name service inherited much of Grapevine's structure [OD83]. Lampson's global name service extends these notions to deeper hierarchy and explicit nondeterministic semantics to tolerate unreliability [Lam86].

14.4 Distributed file systems

The problem of distributing access to files is more restricted than distributing access to arbitrary resources, and yet it is by no means a trivial problem. Distributed file systems provide insight into the challenges of naming in a global environment and transparent distribution across slow and unreliable networks. Several useful surveys of distributed file system technologies introduce the topic [LS90, Wel94, DMST95, AEK96].

The Andrew File System is the first distributed file system to successfully span administrative domains. It employs aggressive caching to contend with slow networks. Andrew File System configuration depends upon administrators with special privileges, however, and its naming structure reflects the hierarchy of participating administrative domains [MSC+86].

14.5 Worldwide systems

Some ongoing projects aim to produce large-scale virtual computer resources by aggregating resources across a wide-area network. These projects share some of my goals in that they must deal with crossing administrative boundaries, but they have more specific purposes. For example, they are typically not concerned with sharing resources outside the virtual computer defined by the system; in essence, the virtual system becomes its own administrative domain.

The Legion project aims to make a worldwide virtual computer by combining unused workstation resources and exporting them to remote users. While it aims to make a global set of resources available uniformly to the users of the system, it retains a significant distinction between administrators and users. Furthermore, Legion aims to unify primarily generic resources (CPU time, storage space) for distributed computations, rather than unifying all of the specific data and other services that an interactive user would employ. [GWtL97].

The GLOBE project has similar goals; its unifying concept is a distributed shared object, implemented as a distributed collection of heavyweight objects that support thread, communication, and replication semantics for the object's implementation. GLOBE's scalable location service maps an object identifier to the locations where the distributed object can be found. The service supports frequent updates, but does not address the structure of names other than to say that they are location independent [vSHT99, vSHBT98].

The *grid* is a metaproject with the goal of creating "a universal source of computing power" that would enable new distributed applications with high computational require-

ments [FK98]. Participants of the project are currently defining the requirements for grids, including requirements for naming and sharing that can span the multiple administrative domains that the proposed grid itself must span. The applications that motivate the grid are novel, but as I do, the grid aims to remove administrative boundaries to the distribution of computational tasks.

Chapter 15

Conclusion and Contributions

In this dissertation, I have presented an architecture for naming and sharing with the express goal of freeing users from artificial administrative boundaries. I conclude the following.

In a world where users access resources from a variety of sources, we can make resource access uniform and simple by organizing naming naming and security mechanisms around user-to-user sharing rather than administrative domains. Although the philosophy alters the design of the system and the mechanisms that it supports, it does not limit the policies administrators can enforce. The architecture simply removes artificial boundaries to resource access.

I evaluate my architecture and existing architectures with respect to these qualities, and discover that my approach trades off performance and user cost of storage management to enable names with high mnemonic and semantic value that users can easily share.

I contribute a naming mechanism based on user-relative paths that reflects user-to-user relationships. To ensure that all applications exhibit the benefits of user-specific naming, I structure the system so that naming is a separate, user-controlled layer between applications and other system services. Name bindings should be stored on the server side of the naming interface so that they can always be shared with other users.

I contribute an authorization mechanism that enables users to uniformly specify their sharing requirements with other users, regardless of whether their colleagues are in the same administrative domain. Hence sharing reflects user-to-user relationships, not administrative hierarchy. My sharing model is founded on a formal logic and semantics, so its meaning is unambiguous and implementations can be verified against a clear standard.

I establish that my sharing model enables an end-to-end approach to authorization that has benefits even within administrative domains. It enables us to build gateways that span network scales, levels of abstraction, and protocols while maintaining the flow of authorization information from the client to the ultimate resource server.

My prototype system and applications demonstrates the naming and sharing mechanisms at work. I compare it to conventionally-organized systems and applications, and evaluate its characteristics qualitatively and quantitatively. The system exhibits the qualities we desire, and its performance roughly tracks that of conventional hop-by-hop authorization protocols with similar implementations.

The user-centered philosophy of system organization is the organizing element behind my work. I conclude that the philosophy is compatible with the usual goals of system design,

and in fact simplifies the organization of systems by reducing many administrative tasks to special applications of user tools. I hope that system architects will consider adopting my philosophy when they develop future designs.

Chapter 16

Future work

As I have discussed, the ultimate vision of my project is to replace conventionally-structured systems and applications with those structured in a user-centric way, enabling users to smoothly traverse administrative boundaries. To that end, I envision extending this work in several directions, each designed to further the practical applicability of my work. I list these directions here in decreasing order of grandiosity. The most involved goal is to extend the notion of end-to-end authorization to end-to-end secrecy. Within the scope of end-to-end authorization, I envision several extensions to my protocols and applications to make them more practical and useful. Finally, there are opportunities for performance improvements that would improve my prototype towards production quality. I detail each of these directions in the following sections.

16.1 End-to-end secrecy

I would like to cross my work on end-to-end authorization with work on models of secrecy and information flow, such as that in [BAN90], to work toward an end-to-end model that can capture notions of who should *know* what. In such an architecture I imagine a gateway that operates with only partial access to the information it translates, passing from server to client encrypted content that it need not view to accomplish its task.

16.2 Applications

My server authentication implementation touched on authenticating documents and resources relative to how the user knows them. I would like to fill out this notion to implement "self-certifying" web pages or e-mail or path names. The SFS secure file system pioneered the idea of self-certifying path names [MKKW99]. Indeed, I could integrate SFS' path names as a hop-by-hop protocol in my end-to-end authorization model.

I might also implement other protocols to add to the suite of those supported by Snowflake. A version of HTTP over SSL would be a natural choice. SPKI supports revocation through Certificate Revocation Lists (CRLs) and one-time revalidation authorities. Section 7.7 describes how my logic can model these mechanisms; I would like to implement them using my logic to capture their participation in transactions.

To encourage Snowflake's use in production systems, I might adapt production applications to support Snowflake authorization.

I could arrange for users to be able to delegate their authority over a resource to another while restricting the recipient to only access the resource through some abstracting gateway. For example, Alice might want to let Bob read only message 174 of her e-mail. To do so, she must give him access to the database server, but ensure that he only access the resource through the gateway, since the database server has no notion of individual messages, only lower-level abstractions like headers. By using conjunction, I could encode Alice's delegation to "Bob *and* the gateway," ensuring that Bob cannot exercise the delegated authority without using the abstracting gateway.

16.3 Performance

In Section 2.5.4 I said that Snowflake has no protocol to communicate whether the results of a name resolution may be cached. There indeed is such a protocol once we introduce authorization. If a name binding is backed by the authority of the resource to authenticate itself to that name, then the validity duration of that authority establishes an upper-bound the cachability of the name. Clever application of authorization may not only communicate cachability information, but enable the construction of secure shared caches.

I comment on the poor performance of the gateway in Section 12.4. Among other optimizations, I could extend the gateway to recognize the common case of a repeated request and immediately include the newly-supplied proof to the `proofRecipient`, saving an extra round-trip in the RMI protocol.

Section 12.5.3 indicates how poorly the SPKI libraries perform. I would like to repair those libraries to parse S-expressions efficiently. Once the libraries are efficient, proof transmission could be improved by using references to remove repeated occurrences of subexpressions, either within one proof or even across proofs on a long-lived communication. Proof security would be uncompromised, since a corrupted reference could only cause the proof recipient to receive an unverifiable proof. Passing proofs by reference would not only save parsing time at the recipient process, but it would also allow the recipient to skip verification of subproofs it has seen before.

Acknowledgements

I hope that this dissertation, and everything else I do in life, glorifies God. Your creations dwarf our greatest engineering feats. Thanks for the desire explore.

Thanks to my parents for love and direction, and for encouraging me to explore.

Thanks to Andy for spending hundreds of your Saturdays helping me explore the worlds of computer science, math, and engineering. Your spirit taught me to see the world as my toybox.

Thanks to my bride Christina for your faithfulness and willingness to let me explore.

Thanks to my advisor Dave for your patience and willingness to let me explore, and for shaping me as a scientist and communicator.

Thanks to the USENIX organization for funding my research.

Thanks to John Lamping for patiently helping me understand logical proof systems and semantic models. Thanks to Jon Bredin, Valeria de Paiva, Mark Montague and Larry Gariepy for discussions about the formalism, which helped refine the idea.

Thanks to Hany Farid for providing statistical intuition.

Appendix A

Proofs

A.1 Construction of ϕ_T

Definition E12 presupposed the existence of a projection function ϕ_T. We construct such a function now, and show that it satisfies the definition. Let $\overline{W} = 2^T$; that is, worlds in $\overline{M}$ are subsets of T. Define

$$\begin{aligned} &\phi_T(w) = \overline{w} \in \overline{W} \\ &\text{where } (\sigma \in \overline{w}) \equiv (w \in \mathcal{E}(\sigma))\ \forall \sigma \in T \end{aligned} \qquad \text{(Definition E26)}$$

Necessity. Given $\phi_T(w) = \overline{w} = \phi_T(w')$, we know $\forall \sigma \in T,\ \sigma \in \overline{w}$ **iff** $w \in \mathcal{E}(\sigma)$, and likewise, $\forall \sigma \in T,\ \sigma \in \overline{w}$ **iff** $w' \in \mathcal{E}(\sigma)$. Therefore $\forall s \in T,\ w \in \mathcal{E}(\sigma)$ **iff** $w' \in \mathcal{E}(\sigma)$, and we conclude $w \cong_T w'$.

Sufficiency. From the definition of $w \cong_T w'$, we know $\forall \sigma \in T,\ w \in \mathcal{E}(\sigma)$ **iff** $w' \in \mathcal{E}(\sigma)$. Let $\overline{w} = \{\sigma \in T | w \in \mathcal{E}(\sigma)\}$ and $\overline{w}' = \{\sigma \in T | w' \in \mathcal{E}(\sigma)\}$. From our hypothesis we know that the conditions on $\overline{w}$ and $\overline{w}'$ are the same, so $\phi_T(w) = \overline{w} = \overline{w}' = \phi_T(w')$.

In the following proofs, we generally use a bar ($\overline{w}$) to indicate a member of an equivalence class constructed as shown here.

A.2 Equivalence of ϕ_T^R and ϕ_T^+ definitions of $\overset{T}{\Rightarrow}$

We now justify our claim in Section 4.1 that Definition E14 and Definition E16 are equivalent.

Necessity. Assume $\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A}$ holds according to Definition E14:

$$\forall w_0' \ \left(\phi_T^w(\mathcal{R}(\mathcal{A})(w_0')) \subseteq \phi_T^w(\mathcal{R}(\mathcal{B})(w_0'))\right)$$

For all $\langle w_0, w_1\rangle$,

$$\begin{aligned}
\langle w_0, w_1\rangle \in \mathcal{R}(\mathcal{A}) &\supset w_1 \in \mathcal{R}(\mathcal{A})(w_0) \\
&\supset \ \overline{w}_1 \in \phi_T^w(\mathcal{R}(\mathcal{A})(w_0)), \\
&\qquad\quad \overline{w}_1 = \phi_T(w_1) \\
&\supset \overline{w}_1 \in \phi_T^w(\mathcal{R}(\mathcal{B})(w_0)) \qquad \text{(using the assumption)} \\
&\supset \ \exists w_1' \cong_T w_1, \\
&\qquad \langle w_0, w_1'\rangle \in \mathcal{R}(\mathcal{B})(w_0) \\
&\supset \langle w_0, w_1\rangle \in \phi_T^+(\mathcal{R}(\mathcal{B}))
\end{aligned}$$

Sufficiency. Assume $\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A}$ holds according to Definition E16:

$$\mathcal{R}(\mathcal{A}) \subseteq \phi_T^+(\mathcal{R}(\mathcal{B}))$$

Given w_0 and $\overline{w}_1 \in \phi_T^w(\mathcal{R}(\mathcal{A})(w_0))$, we know that there is some $w_1 \in \mathcal{R}(\mathcal{A})(w_0)$, with $\overline{w}_1 = \phi_T(w_1)$. We rewrite the statement $\langle w_0, w_1\rangle \in \mathcal{R}(\mathcal{A})$, and invoke the assumption to get $\langle w_0, w_1\rangle \in \phi_T^+(\mathcal{R}(\mathcal{B}))$. Now we know there exists $\langle w_0, w_1'\rangle \in \mathcal{R}(\mathcal{B})$ with $w_1' \cong_T w_1$. Changing notation again, $w_1' \in \mathcal{R}(\mathcal{B})(w_0)$. Since $w_1' \cong_T w_1$, we know $\overline{w}_1 = \phi_T(w_1')$, and we may conclude $\overline{w}_1 \in \phi_T^w(\mathcal{R}(\mathcal{B})(w_0))$.

Together, the two implications show the equivalence.

A.3 An undesirable semantics for $\overset{T}{\Rightarrow}$

Notice that ϕ_T^+ projects only the destination world of each edge in a relation. Why do we not project both ends of the relation? Such a definition actually does not preserve our most basic intuition, that $B \overset{T}{\Rightarrow} A \supset B \overset{T}{\rightarrow} A$. In the model in Figure A.1, the dotted ovals depict the equivalence classes under T; projecting both ends of the edges in $\mathcal{R}(A)$ gives $\{\langle T, \varnothing\rangle\}$, as does $\mathcal{R}(B)$. From world w_0, however, B **says** s but not A **says** s.

Given a relation $\langle w_0, w_1\rangle$, then, the reason we only project w_1 is this: w_0 is affected by what statements are true at w_1; substituting other worlds equivalent with respect to T does no harm. Substituting other worlds for w_0, on the other hand, changes what statements we consider true at w_0.

A.4 Proof of soundness

In this section, we show that our extension to Lampson's calculus is still a sound axiomatization of the presented semantics. Like Lampson's original logic, ours is based on a conventional Kripke semantics of modal logic. The conventional proofs of soundness for Axiom S1, Rule S2, Axiom S3, and Rule S4 apply. Our extensions define $\mathcal{E}$ for a new formula $(\mathcal{B} \overset{T}{\Rightarrow} \mathcal{A})$ and $\mathcal{R}$ for a new principal $(\mathcal{A} \cdot N)$, but do not perturb Abadi's original semantics for the calculus for access control. Because those semantics do not depend on any particular structure in $\mathcal{E}$ or $\mathcal{R}$, the axioms of the calculus remain sound in our extended calculus.

Our present task is to show that the axioms of our extensions are sound.

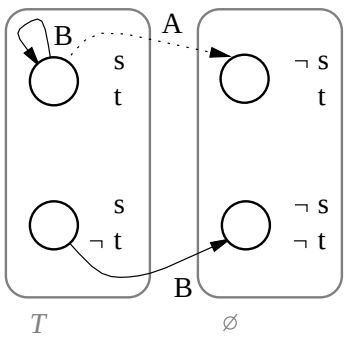


Figure A.1: *In this example,* $T = \{s\}$. *Notice that* $B \overset{T}{\not\rightarrow} A$.

Axiom E1. This axiom follows easily from Definition E14. For all w_0,

$$\begin{aligned}\phi_T^w(\mathcal{R}(\mathcal{A})(w_0)) &\subseteq \phi_T^w(\mathcal{R}(\mathcal{B})(w_0)) \\ &\subseteq \phi_T^w(\mathcal{R}(\mathcal{C})(w_0))\end{aligned}$$ □

The following lemma shows that ϕ_T^+ preserves the union operation. Let R_1 and R_2 be relations.

$$\begin{aligned}&\langle w_0, w_1\rangle \in \phi_T^+(R_1 \cup R_2) \\ &\quad\equiv \exists\, w_1' \cong_T w_1,\ \langle w_0, w_1'\rangle \in R_1 \cup R_2 \\ &\quad\equiv \exists\, w_1' \cong_T w_1, \\ &\qquad\quad \langle w_0, w_1'\rangle \in R_1 \vee \langle w_0, w_1'\rangle \in R_2 \\ &\quad\equiv \quad \exists\, w_1' \cong_T w_1,\ \langle w_0, w_1'\rangle \in R_1 \\ &\qquad \vee \exists\, w_1' \cong_T w_1,\ \langle w_0, w_1'\rangle \in R_2 \\ &\quad\equiv \langle w_0, w_1\rangle \in \phi_T^+(R_1) \vee \langle w_0, w_1\rangle \in \phi_T^+(R_2) \\ &\quad\equiv \langle w_0, w_1\rangle \in \phi_T^+(R_1) \cup \phi_T^+(R_2)\end{aligned}$$

From this equivalence we conclude

$$\phi_T^+(R_1 \cup R_2) = \phi_T^+(R_1) \cup \phi_T^+(R_2) \qquad \text{(Lemma E27)}$$

Axiom E2. We assume the premise in terms of Definition E14:

$$\forall\, w_0' \ (\phi_T^w(\mathcal{R}(\mathcal{A})(w_0')) \subseteq \phi_T^w(\mathcal{R}(\mathcal{B})(w_0')))$$

We can readily reason for all w_0:

$$\begin{aligned}
&\phi_T^w(\mathcal{R}(\mathcal{A} \wedge \mathcal{C})(w_0)) \\
&\quad = \phi_T^w((\mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{C}))(w_0)) \\
&\quad = \phi_T^w(\mathcal{R}(\mathcal{A})(w_0) \cup \mathcal{R}(\mathcal{C})(w_0)) \\
&\quad = \phi_T^w(\mathcal{R}(\mathcal{A})(w_0)) \cup \phi_T^w(\mathcal{R}(\mathcal{C})(w_0)) \\
&\quad \subseteq \phi_T^w(\mathcal{R}(\mathcal{B})) \cup \phi_T^w(\mathcal{R}(\mathcal{C})) \\
&\quad = \phi_T^w(\mathcal{R}(\mathcal{B})(w_0) \cup \mathcal{R}(\mathcal{C})(w_0)) \\
&\quad = \phi_T^w((\mathcal{R}(\mathcal{B}) \cup \mathcal{R}(\mathcal{C}))(w_0)) \\
&\quad = \phi_T^w(\mathcal{R}(\mathcal{B} \wedge \mathcal{C})(w_0))
\end{aligned}$$ □

Axiom E3. This axiom has a symmetric consequence, so we only show the first conjunct. For all worlds w_0,

$$\begin{aligned}
\phi_T^w(\mathcal{R}(\mathcal{A})(w_0)) &\subseteq \phi_T^R((\mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{B}))(w_0)) && \text{(Lemma E27)} \\
&\subseteq \phi_T^R(\mathcal{R}(\mathcal{C})(w_0)) && \text{(premise)}
\end{aligned}$$ □

We digress to point out that we may discard "identical worlds" from a model without loss of generality. That is, imagine we have a model $\mathcal{M}$ with two worlds w_1 and w_2 where $w_1 \in \mathcal{E}(\sigma)$ **iff** $w_2 \in \mathcal{E}(\sigma)$ for every formula $\sigma \in \Sigma^*$. The extra world w_2 appears in every $I(s)$ that w_1 appears in. Any edge in any relation ending in w_1 has a related edge ending in w_2 ($\langle w, w_1 \rangle \in J(\mathcal{A}) \equiv \langle w, w_2 \rangle \in J(\mathcal{A})$); likewise edges starting at w_1 have a related edge starting at w_2 in every relation. The same holds for the relations in the name interpretation function $K(\mathcal{A}, N)$. It is clear that the extension function $\mathcal{R}$, and hence $\mathcal{E}$, have the same overlap with respect to w_1 and w_2, so that $w_1 \in \mathcal{E}(\sigma) \equiv w_2 \in \mathcal{E}(\sigma)$.

Given this definition, we can build a model $\mathcal{M}' = \langle W', w_0', I', J', K' \rangle$ that discards w_2:

$$\begin{aligned}
W' &= W - \{w_2\} \\
w_0' &= \begin{cases} w_1 & \text{if } w_0 \\ w_0 & \text{otherwise} \end{cases} \\
I'(s) &= I(s) - \{w_2\} \\
J'(\mathcal{A}) &= J(\mathcal{A}) - \{\langle w, w' \rangle | w = w_2 \vee w' = w_2\} \\
K'(\mathcal{A}, N) &= K(\mathcal{A}, N) \\
&\quad - \{\langle w, w' \rangle | w = w_2 \vee w' = w_2\}
\end{aligned}$$

Happily, $\mathcal{M}'$ preserves every consequence of $\mathcal{M}$: $(\mathcal{M} \models \sigma) \equiv (\mathcal{M}' \models \sigma)$. Why? Whenever $w_0 \in \mathcal{E}(\sigma)$, $w_0' \in \mathcal{E}'(\sigma)$, either for exactly the same reasons (when $w_0 \neq w_2$), or because $w_0 = w_2$, so $w_0 = w_2 \in \mathcal{E}(\sigma) \equiv w_1 \in \mathcal{E}(\sigma)$, and then $w_0' \in \mathcal{E}'(s)$ for the same reasons that $w_1 \in \mathcal{E}(s)$.

Convinced that duplicate worlds do not alter the consequences of a model, we may now assume that no models contain identical worlds, without damaging our semantics. If we know $w_1 \neq w_2$, we can assume the existence of a formula σ with $(w_1 \in \mathcal{E}(\sigma)) \not\equiv (w_2 \in \mathcal{E}(\sigma))$,

and conclude that $w_1 \not\cong_{\mathcal{U}} w_2$ (by Definition E11). Therefore, $\phi_{\mathcal{U}}$ is bijective:

$$w_1 \neq w_2 \supset \phi_{\mathcal{U}}(w_1) \neq \phi_{\mathcal{U}}(w_2)$$

By the definition of ϕ_T^+ it is obvious that any relation $R \in \phi_T^+(R)$. But when $T = \mathcal{U}$, the converse is also true:

$$\begin{aligned}
&\langle w_0, w_1\rangle \in \phi_{\mathcal{U}}^+(R)\\
&\quad \supset \exists\ w_1' \text{ such that } \langle w_0, w_1'\rangle \in R,\\
&\qquad\quad \phi_{\mathcal{U}}(w_1') = \phi_{\mathcal{U}}(w_1)\\
&\quad \supset w_1' = w_1\\
&\quad \supset \langle w_0, w_1\rangle \in R
\end{aligned}$$

Now we have $\phi_{\mathcal{U}}^+(R) = R$.

Axiom E4. Expanding the definition of $B \overset{\mathcal{U}}{\Rightarrow} A$ and applying the previous result gives $\mathcal{R}(A) \subseteq \phi_{\mathcal{U}}^+(\mathcal{R}(B)) = \mathcal{R}(B)$, which satisfies the definition of $B \Rightarrow A$. □

Justifying axiom Axiom E5 requires two lemmas that relate representatives of equivalence classes under different projections.

First, a representative of a projection due to a small set has a "big brother" in any projection due to a superset, and the structure of the brothers is closely related:

$$\begin{aligned}
&\overline{w}_1' \in \phi_{T'}^w(S_w),\ T' \subseteq T\\
&\supset \exists\ \overline{w}_1 \in \phi_T^w(S_w),\ \overline{w}_1' = \overline{w}_1 \cap T' \qquad \text{(Lemma E28)}
\end{aligned}$$

Proof. By the first premise, there is a $w_1 \in S_w$ where $\overline{w}_1' = \phi_{T'}(w_1)$. From Definition E26 we know

$$(\sigma \in \overline{w}_1') \equiv (w_1 \in \mathcal{E}(\sigma))\ \forall\ \sigma \in T' \tag{1}$$

Let $\overline{w}_1 = \phi_T(w_1)$; since $w_1 \in S_w$, $\overline{w}_1 \in \phi_T^w(S_w)$. Having exhibited $\overline{w}_1$, we need only show $\overline{w}_1 \cap T = \overline{w}_1'$.

We again invoke Definition E26 to get

$$(\sigma \in \overline{w}_1) \equiv (w_1 \in \mathcal{E}(\sigma))\ \forall\ \sigma \in T \tag{2}$$

First, $\sigma \in \overline{w}_1 \cap T'$ means both $\sigma \in T'$, and because $T' \subseteq T$, $\sigma \in T$. The latter allows us to use (2) to write $w_1 \in \mathcal{E}(\sigma)$, and then we invoke (1) to get $\sigma \in \overline{w}_1'$. Conversely, $\sigma \in \overline{w}_1'$ means $\sigma \in T'$ and hence $\sigma \in T$. We apply (1) to get $w_1 \in \mathcal{E}(\sigma)$, and apply (2) to get $\sigma \in \overline{w}_1$. Now we have shown $\overline{w}_1 \cap T' = \overline{w}_1'$, proving the lemma. □

The second lemma is approximately the converse of the first:

$$\begin{aligned}
&\overline{w}_1 \in \phi_T^w(S_w),\ \overline{w}_1' = \overline{w}_1 \cap T'\ T' \subseteq T\\
&\supset \overline{w}_1' \in \phi_{T'}^w(S_w) \qquad \text{(Lemma E29)}
\end{aligned}$$

Proof. The first premise, by Definition E13, implies the existence of a $w_1 \in R$, and Definition E26 lets us write

$$(\sigma \in \overline{w}_1) \equiv (w_1 \in \mathcal{E}(\sigma)) \; \forall \, \sigma \in T \tag{1}$$

For every $\sigma \in T'$, all of the following hold:

$$\begin{aligned} \sigma \in T & \qquad \text{(third premise)} \\ (\sigma \in \overline{w}_1) \equiv (w_1 \in \mathcal{E}(\sigma)) & \qquad (1) \\ (\sigma \in \overline{w}_1 \cup T') \equiv (w_1 \in \mathcal{E}(\sigma)) & \\ (\sigma \in \overline{w}'_1) \equiv (w_1 \in \mathcal{E}(\sigma)) & \qquad \text{(second premise)} \end{aligned}$$

This last result implies that $\overline{w}'_1 = \phi_{T'}(w_1)$, which is sufficient to prove the conclusion of the lemma. □

Axiom E5. We take as our hypothesis $\mathcal{M} \models B \overset{T}{\Rightarrow} A$, that is:

$$\phi^w_T(\mathcal{R}(\mathcal{A})(w_0)) \subseteq \phi^w_T(\mathcal{R}(\mathcal{B})(w_0))$$

Given any world w_0 and sets $T' \subseteq T$, we assume $\overline{w}'_1 \in \phi^w_{T'}(\mathcal{R}(\mathcal{A})(w_0))$ and set out to prove $\overline{w}'_1 \in \phi^w_{T'}(\mathcal{R}(\mathcal{B})(w_0))$. By the assumption and Lemma E28, we know

$$\exists \, \overline{w}_1 \in \phi^w_T(\mathcal{R}(\mathcal{A})(w_0)), \; \overline{w}'_1 = \overline{w}_1 \cap T'$$

The hypothesis gives $\overline{w}_1 \in \phi^w_T(\mathcal{R}(\mathcal{B})(w_0))$, which satisfies the premise for Lemma E29. Hence we know $\overline{w}'_1 \in \phi^w_{T'}(\mathcal{R}(\mathcal{B})(w_0))$, and we have proven that

$$\forall \, w_0, \; (\phi^w_{T'}(\mathcal{R}(\mathcal{A})(w_0)) \subseteq \phi^w_{T'}(\mathcal{R}(\mathcal{B})(w_0))) \qquad \square$$

Theorem E6. Apply Axiom E5 twice to the premises to get two relations restricted by $S \cup T$, then apply Axiom E1 to collapse them into the relation in the conclusion. □

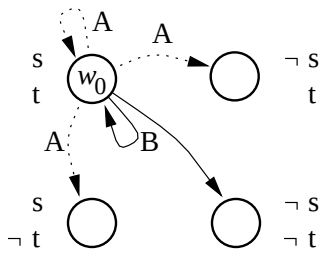
Result E7. Figure A.2 gives a counterexample that justifies the result. The diagram in the figure models $B \overset{S}{\Rightarrow} A$ and $B \overset{T}{\Rightarrow} A$. The statement $B \overset{S \cup T}{\Rightarrow}$, however, fails. Projecting the model under $S \cup T$ gives the original picture, since each world falls in a separate equivalence class. Notice that B says $\neg(s \wedge \neg t)$: that statement is true in both worlds B considers possible. But A does not believe it, since A can see the lower-left world, where the statement is false.

Why should this result be intuitive or desirable? Recall from Section 7.9 that the strength of $\overset{T}{\Rightarrow}$ means that a delegation regarding T may imply a delegation regarding a larger set T^* that includes formulas constructed from the members of T. In our example, B speaks for A regarding formulas composed exclusively with the primitive s or the primitive t, but not regarding formulas combining the two. The closure of the restriction set $S \cup T$ includes formulas such as $\neg(s \wedge \neg t)$.

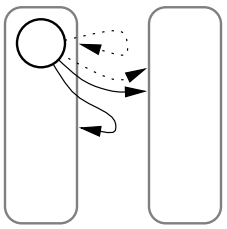
Axiom E8. Assume the premise in terms of Definition E14:

$$\forall \, w'_0 \; (\phi^w_T(\mathcal{R}(\mathcal{A})(w'_0)) \subseteq \phi^w_T(\mathcal{R}(\mathcal{B})(w'_0)))$$

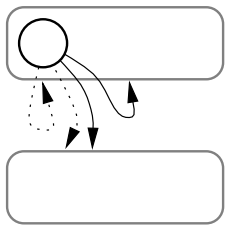
Let $\overline{w}$ belong to $\phi^w_T(\mathcal{R}(\mathcal{C}|\mathcal{A})(w_0))$. The semantics for quoting gives $\overline{w} \in \phi^w_T((\mathcal{R}(\mathcal{C}) \circ \mathcal{R}(\mathcal{A}))(w_0))$. An edge only exists in a composition if we have w_1 and w_2 such that



In this model, A's relation at w_0 is not a subset of B's.



Projected under $S = \{s\}$, however, the subset relation holds ...



... as it does under $T = \{t\}$.

Figure A.2: *A counterexample showing why two delegations for sets S and T do not imply a delegation for set $S \cup T$ (Result E7).*

$\langle w_0, w_1\rangle \in \mathcal{R}(\mathcal{C})$ and $\langle w_1, w_2\rangle \in \mathcal{R}(\mathcal{A})$; Definition E13 guarantees that we have such w_1, w_2 with $\overline{w} = \phi_T(w_2)$.

Since $w_2 \in \mathcal{R}(\mathcal{A})(w_1)$, we can use the assumption to show the existence of $w_2' \in \mathcal{R}(\mathcal{B})(w_1)$ with $\phi_T(w_2') = \phi_T(w_2) = \overline{w}$. That means that $\overline{w} \in \phi_T^w(\mathcal{R}(\mathcal{B})(w_1))$, and hence $\overline{w} \in \phi_T^w((\mathcal{R}(\mathcal{C}) \circ \mathcal{R}(\mathcal{B}))(w_0))$. By the definition of quoting, we arrive at $\overline{w} \in \phi_T^w(\mathcal{R}(\mathcal{C}|\mathcal{B})(w_0))$, which proves the conclusion. □

Result E9. The model in Figure A.3 is a counterexample for $T = \{s\}$ that shows the result. Notice that $B \overset{T}{\Rightarrow} A$: $\mathcal{R}(A)$'s only edge goes from w_0 to the equivalence class of worlds where s is true, and $\mathcal{R}(B)$ also has such an edge (the loop at w_0). When we compose the relations, however, we see that $B|C$ **says** s, but not $A|C$ **says** s. The equivalence classes of $\{C \textbf{ says } s\}$ are different than the equivalence classes of $\{s\}$.

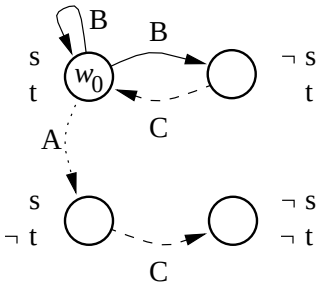


Figure A.3: *A model that demonstrates Result E9.*

Axiom E10. Inductively applying Axiom E25 and Axiom E24 shows as a theorem that $B \overset{T}{\Rightarrow} A$ implies $B \overset{T^*}{\Rightarrow} A$. Therefore, we may immediately replace the premise of this axiom with $B \overset{((T^*)C)^*}{\Rightarrow} A$, which follows by the theorem from the original premise. Herein we omit the parentheses for the postfix set operators * and C, and simply write T^*C^*.

Hence we begin with the hypothesis that

$$\mathcal{R}(\mathcal{A}) \subseteq \phi^+_{T^*C^*}(\mathcal{R}(\mathcal{B}))$$

We are given some $w_0 \in W$ and the existence of $\overline{w}_2 \in \phi_T^w(\mathcal{R}(\mathcal{A}|\mathcal{C})(w_0))$. The set can be rewritten $\phi_T^w((\mathcal{R}(\mathcal{A}) \circ \mathcal{R}(\mathcal{C}))(w_0))$, so we know that there exist w_1 and w_2, where

$$\begin{aligned} \langle w_0, w_1 \rangle &\in \mathcal{R}(\mathcal{A}) \\ \langle w_1, w_2 \rangle &\in \mathcal{R}(\mathcal{C}) \\ \overline{w}_2 &= \phi_{T^*C^*}(w_2) \end{aligned}$$

The last expression means that for all $\sigma \in T$, $\sigma \in \overline{w}_2$ if and only if $w_1 \in \mathcal{E}(\sigma)$.

Define the formula

$$\tau_2 = \bigwedge_{\sigma \in T} \begin{cases} \sigma & \text{if } \sigma \in \overline{w}_2 \\ \neg\sigma & \text{otherwise} \end{cases}$$

Intuitively, τ_2 is true at precisely those worlds that map to $\overline{w}_2$ under ϕ_T. We have constructed τ_2 such that $w_2 \in \mathcal{E}(\tau_2)$.

Since $\langle w_1, w_2 \rangle \in \mathcal{R}(\mathcal{C})$, we know $\mathcal{R}(\mathcal{C}) \not\subseteq \mathcal{E}(\neg\tau_2)$, and therefore $w_1 \notin \mathcal{E}(\mathcal{C}\,\mathbf{says}\,\neg\tau_2)$, and finally $w_1 \in \mathcal{E}(\neg\mathcal{C}\,\mathbf{says}\,\neg\tau_2)$. The propositional closure of T ensures that each conjunct of τ_2, and thus τ_2 itself and $\neg\tau_2$, appear in T^*. The modal closure over "$\mathcal{C}\,\mathbf{says}$" ensures that $(\mathcal{C}\,\mathbf{says}\,\neg\tau_2) \in T^*C$, and therefore $(\neg\mathcal{C}\,\mathbf{says}\,\neg\tau_2) \in T^*C^*$.

Now we may employ the hypothesis to show that there exists a $w_1' \in \mathcal{R}(\mathcal{B})(w_0)$ with $w_1' \cong_{T^*C^*} w_1$. It follows that:

$$\begin{aligned} w_1' &\in \mathcal{E}(\neg\mathcal{C}\,\mathbf{says}\,\neg\tau_2) \\ &= W - \mathcal{E}(\mathcal{C}\,\mathbf{says}\,\neg\tau_2) \\ &= W - \{w | \mathcal{R}(\mathcal{C})(w) \subseteq \mathcal{E}(\neg\tau_2)\} \\ &= \{w | \mathcal{R}(\mathcal{C})(w) \not\subseteq \mathcal{E}(\neg\tau_2)\} \\ &= \{w | \exists\, w_2' \in \mathcal{R}(\mathcal{C})(w),\ w_2' \notin \mathcal{E}(\neg\tau_2)\} \\ &= \{w | \exists\, w_2' \in \mathcal{R}(\mathcal{C})(w),\ w_2' \in \mathcal{E}(\tau_2)\} \end{aligned}$$

That is, we know there is a $w_2' \in \mathcal{E}(\tau_2)$, with $\langle w_1', w_2' \rangle \in \mathcal{R}(\mathcal{C})$.

With both $\langle w_0, w_1' \rangle \in \mathcal{R}(\mathcal{B})$ and $\langle w_1', w_2' \rangle \in \mathcal{R}(\mathcal{C})$, we have $\langle w_0, w_2' \rangle \in \mathcal{R}(\mathcal{B}) \circ \mathcal{R}(\mathcal{C}) = \mathcal{R}(\mathcal{B}|\mathcal{C})$. From the definition of τ_2, we know that w_2' is in $\mathcal{E}(\sigma)$ exactly when $\sigma \in \overline{w}_2$ for all $\sigma \in T$, so $\overline{w}_2 = \phi_T(w_2')$. We have shown that $\overline{w}_2 \in \phi_T^w(\mathcal{R}(\mathcal{B}|\mathcal{C})(w_0))$, and therefore that given the hypothesis, the model supports $\mathcal{B}|\mathcal{C} \overset{T}{\Rightarrow} \mathcal{A}|\mathcal{C}$. □

Axiom E17. This axiom follows from our brute-force semantics for names. Assume the premise:

$$\mathcal{R}(\mathcal{A}) \subseteq \mathcal{R}(\mathcal{B})$$

We want to show that

$$\mathcal{R}(\mathcal{A} \cdot N) \subseteq \mathcal{R}(\mathcal{B} \cdot N),$$

which is of course trivial thanks to requirement I of Definition E21.

Theorem E18. Since $(\mathcal{A} \wedge \mathcal{B}) \Rightarrow \mathcal{A}$, $(\mathcal{A} \wedge \mathcal{B}) \cdot N \Rightarrow \mathcal{A} \cdot N$ (by Axiom E17, with $T = \mathcal{U}$). The same is true for $\mathcal{B}$, proving:

$$(\mathcal{A} \wedge \mathcal{B}) \cdot N \Rightarrow (\mathcal{A} \cdot N) \wedge (\mathcal{B} \cdot N) \qquad \square$$

Axiom E19. Requirement III of Definition E21 exists to support this axiom. It says:

$$\mathcal{R}(\mathcal{A} \wedge \mathcal{B}) \cdot N \subseteq \mathcal{R}(\mathcal{A} \cdot N) \cup \mathcal{R}(\mathcal{B} \cdot N)$$

The right-hand side, by the semantics for $\wedge$, is equal to $\mathcal{R}((\mathcal{A} \cdot N) \wedge (\mathcal{B} \cdot N))$, completing the proof.

Theorem E20. Theorem E18 and Axiom E19 together show equality. □

Axiom E23. Assume $\mathcal{R}(\mathcal{B}) \subseteq \mathcal{E}(\sigma') \supseteq \mathcal{R}(\mathcal{A}) \subseteq \mathcal{E}(\sigma')$ for $\sigma' \in \{\sigma, \tau\}$. Further, assume that $\mathcal{R}(\mathcal{B}) \subseteq \mathcal{E}(\sigma \wedge \tau)$. Using the semantics of $\wedge$, we can write $\mathcal{R}(\mathcal{B}) \subseteq \mathcal{E}(\sigma) \cap \mathcal{E}(\tau)$, and hence $\mathcal{R}(\mathcal{B}) \subseteq \mathcal{E}(\sigma)$ and $\mathcal{R}(\mathcal{B}) \subseteq \mathcal{E}(\tau)$. By the first assumption, we can replace $\mathcal{B}$ in both statements with $\mathcal{A}$, use the definition of $\cap$ and the semantics of $\wedge$, and conclude that $\mathcal{R}(\mathcal{A}) \subseteq \mathcal{E}(\sigma \wedge \tau)$, justifying the axiom. □

Axiom E24. Let $T = \{\sigma, \tau\}$ and $T' = \{\sigma \wedge \tau\}$. Assume first that:

$$\phi_T^w(\mathcal{R}(\mathcal{A})(w_0')) \subseteq \phi_T^w(\mathcal{R}(\mathcal{B})(w_0')) \; \forall \, w_0' \in W$$

Second, assume we are given w_0 and $\overline{w}_1'$ such that $\overline{w}_1' \in \phi_{T'}^w(\mathcal{R}(\mathcal{A})(w_0))$. We have the existence of a $w_1 \in \mathcal{R}(\mathcal{A})(w_0)$ with $\overline{w}_1' = \phi_{T'}(w_1)$.

Let $\overline{w}_1 = \phi_T(w_1)$. By our first assumption, $\overline{w}_1 \in \phi_T^w(\mathcal{R}(\mathcal{B})(w_0))$, so there is a $w_1' \in \mathcal{R}(\mathcal{B})(w_0)$ with $\overline{w}_1 = \phi_T(w_1')$. We claim that $\phi_{T'}(w_1') = \overline{w}_1'$, a claim supported by leaning on the definition of ϕ_T:

$$\begin{aligned}
\sigma \wedge \tau \in \phi_{T'}(w_1') &\equiv w_1' \in \mathcal{E}(\sigma \wedge \tau) \\
&\equiv w_1' \in \mathcal{E}(\sigma) \wedge w_1' \in \mathcal{E}(\tau) \\
&\equiv \sigma \in \overline{w}_1 \wedge \tau \in \overline{w}_1 \\
&\equiv w_1 \in \mathcal{E}(\sigma) \wedge w_1 \in \mathcal{E}(\tau) \\
&\equiv w_1 \in \mathcal{E}(\sigma \wedge \tau) \\
&\equiv \sigma \wedge \tau \in \overline{w}_1'
\end{aligned}$$

Since $\overline{w}_1'$ is either $T = \{\sigma \wedge \tau\}$ or $\varnothing$, we have shown the equality, and that $\overline{w}_1' \in \phi_{T'}^w(\mathcal{R}(\mathcal{B})(w_0))$. Therefore the model supports $B \overset{\{\sigma \wedge \tau\}}{\Rightarrow} A$. □

Axiom E25. The structure of this proof parallels that of Axiom E24. Let $T = \{\sigma\}$ and $T' = \{\neg\sigma\}$. Assume first that:

$$\phi_T^w(\mathcal{R}(\mathcal{A})(w_0')) \subseteq \phi_T^w(\mathcal{R}(\mathcal{B})(w_0')) \; \forall \, w_0' \in W$$

Second, assume we are given w_0 and $\overline{w}_1'$ such that $\overline{w}_1' \in \phi_{T'}^w(\mathcal{R}(\mathcal{A})(w_0))$. That implies the existence of a $w_1 \in \mathcal{R}(\mathcal{A})(w_0)$, with $\overline{w}_1' = \phi_{T'}(w_1)$. By the definition of $\phi_{T'}$ we know $w_1 \in \mathcal{E}(\neg\sigma)$ if and only if $\neg\sigma \in \overline{w}_1'$. Using the semantics of $\neg$, we can rewrite that expression as

$$w_1 \in \mathcal{E}(\sigma) \;\; \textbf{iff} \;\; \neg\sigma \notin \overline{w}_1'$$

Define

$$\overline{w}_1 = \begin{cases} T & \text{if } \overline{w}_1' = \varnothing \\ \varnothing & \text{otherwise } (\overline{w}_1' = T') \end{cases}$$

Clearly $\sigma \in \overline{w}_1$ if and only if $\neg\sigma \notin \overline{w}'_1$. Now we can write

$$w_1 \in \mathcal{E}(\sigma) \textbf{ iff } \sigma \in \overline{w}_1$$

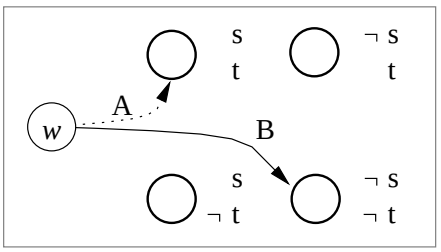
This expression satisfies the definition of ϕ_T, so we have $\phi_T(w_1) = \overline{w}_1$. Because $w_1 \in \mathcal{R}(A)(w_0)$, we know $\overline{w}_1 \in \phi_T^w(\mathcal{R}(A)(w_0))$.

Using the first assumption, we have $\overline{w}_1 \in \phi_T^w(\mathcal{R}(B)(w_0))$. Using arguments analogous to those above, we have the existence of a $w'_1 \in \mathcal{R}(B)(w_0)$, and by the definition of ϕ_T, we can show that $\overline{w}'_1$ is in $\phi_T^w(\mathcal{R}(B)(w_0))$ as well. The model supports $\mathcal{B} \overset{\neg\sigma}{\Rightarrow} \mathcal{A}$. □

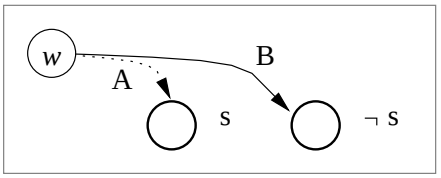
A.5 Relationships among the restricted relations

In each of the examples below, assume $T = \{s\}$.

$\overset{T}{\Rrightarrow}$ **is not stronger than** $\overset{T}{\Rightarrow}$**.** The subset relation in the projected model $\overline{M}$ of $\overset{T}{\Rrightarrow}$ holds with the possible exception of the single world $\overline{w}_T = T$ that represents the equivalence class of worlds in $\mathcal{M}$ in which all statements in T hold. Clearly ϕ_T takes every member of $\cap_{\sigma\in T}\mathcal{E}(\sigma)$ to that representative. The counterexample illustrated in Figure A.4 highlights this exception.



The set $\cap_{s\in T}\mathcal{E}(s)$ is the left pair of worlds (where s is true); the only edge belonging to $\mathcal{R}(A)$ terminates in one of those worlds. Therefore, in this model, $\mathcal{R}(A)(w) - \cap_{s\in T}\mathcal{E}(s) \subseteq \mathcal{R}(B)(w)$, and we conclude that $B \overset{T}{\Rrightarrow} A$.

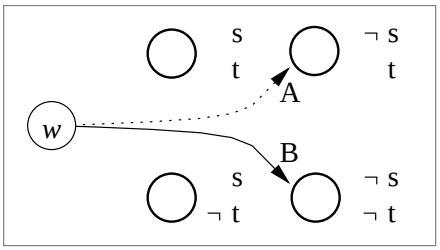


The mapping ϕ_T that reduces the worlds above to equivalence classes modulo statements in T will make this model $\mathcal{M}'$. $\phi_T^R(\mathcal{R}(A))$ includes an edge to the equivalence class labeled s, but $\phi_T^R(\mathcal{R}(B)(w))$ does not. Therefore, $B \overset{T}{\not\Rightarrow} A$.

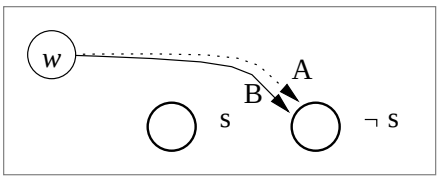
Figure A.4: *A counterexample that shows* $B \overset{T}{\Rrightarrow} A$ *does not imply* $B \overset{T}{\Rightarrow} A$.

$\overset{T}{\Rightarrow}$ **is not stronger than** $\overset{T}{\Rrightarrow}$**.** Although just showed that $\overset{T}{\Rrightarrow}$ is not quite stronger than $\overset{T}{\Rightarrow}$, it certainly seems almost so. Indeed, it is very easy to construct an example that shows that the mighty relation does not follow from the basic speaks-for-regarding relation. See Figure A.5.

$\overset{T}{\Rrightarrow}$ **implies** $\overset{T}{\rightarrow}$**.** Assume $\mathcal{R}(A) \subseteq \phi_T^+(\mathcal{R}(B))$. We will prove by contradiction that $B \overset{T}{\rightarrow} A$. To establish a contradiction, we assume there is a statement $\sigma \in T$ and a world w_0 where B **says** σ but not A **says** σ. That is, $\mathcal{R}(B)(w_0) \subseteq \mathcal{E}(\sigma)$ but $\mathcal{R}(A)(w_0) \not\subseteq \mathcal{E}(\sigma)$. The latter means that there is a world $w_1 \in \mathcal{R}(A)(w_0)$, but $w_1 \notin \mathcal{E}(\sigma)$.



Here is a model in which from w, A considers possible a world neither in $\mathcal{R}(B)(w)$ nor $\cap_{s\in T}\mathcal{E}(s)$. So $B \overset{T}{\not\Rightarrow} A$.



Projecting the model onto T, however, shows that $\phi^R_T(\mathcal{R}(A))$ and $\phi^R_T(R(B))$ completely agree on matters related to s; that is, $B \overset{T}{\Rightarrow} A$.

Figure A.5: *A counterexample that shows $B \overset{T}{\Rightarrow} A$ does not imply $B \overset{T}{\Rrightarrow} A$.*

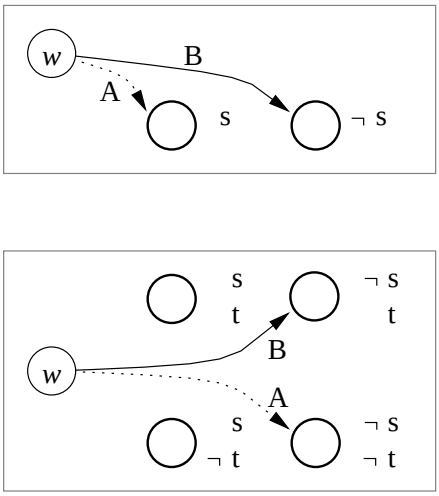
We can push $\langle w_0, w_1\rangle$ through our original assumption to find a w_1' such that $\langle w_0, w_1'\rangle \in \mathcal{R}(B)$ and $w_1' \cong_T w_1$. Definition E11 tells us that $w_1' \notin \mathcal{E}(\sigma)$, which means $\mathcal{R}(B)(w_0) \not\subseteq \mathcal{E}(\sigma)$, which contradicts our second assumption. We may conclude that for all $w_0 \in W$ and $\sigma \in T$, $\mathcal{R}(B)(w_0) \subseteq \mathcal{E}(\sigma)$ implies $\mathcal{R}(A)(w_0) \subseteq \mathcal{E}(\sigma)$. □

$\overset{T}{\Rrightarrow}$ implies $\overset{T}{\rightarrow}$. We assume

$$\mathcal{R}(A)(w_0) - \bigcap_{\tau\in T} \mathcal{E}(\tau) \subseteq \mathcal{R}(B)(w_0)$$

and that $\mathcal{R}(B)(w_0) \subseteq \mathcal{E}(\sigma)$. From the first assumption, any world $w_1 \in \mathcal{R}(A)(w_0)$ is either in $\mathcal{E}(\sigma)$ (let $\tau = \sigma$) or in $\mathcal{R}(B)(w_0)$. The former case trivially guarantees $w_1 \in \mathcal{E}(\sigma)$, and the latter case does so by the second assumption. We conclude that $\mathcal{R}(A)(w_0) \subseteq \mathcal{E}(\sigma)$. □

$\overset{T}{\rightarrow}$ is weaker than $\overset{T}{\Rightarrow}$ and $\overset{T}{\rightarrow}$ is weaker than $\overset{T}{\Rrightarrow}$. See Figure A.6 for counterexamples that illustrate these relationships.



(a) The statement $(\mathcal{R}(B)(w) \subseteq \mathcal{E}(s)) \sqsupset (\mathcal{R}(A)(w) \subseteq \mathcal{E}(s))$ has a false premise, making it vacuously true in this model. Hence this model satisfies $B \overset{T}{\rightarrow} A$. The model is its own projection onto T, however, and it is clear that $B \overset{T}{\not\Rightarrow} A$.

(b) This model satisfies $B \overset{T}{\rightarrow} A$ for the same reason as the model in part (a). The single edge terminating at $\mathcal{R}(A)(w)$, however, is in neither $\mathcal{R}(B)(w)$ nor $\cap_{s \in T} \mathcal{E}(s)$, so $B \overset{T}{\not\Rightarrow} A$.

Figure A.6: *Examples that show why the relation* $\overset{T}{\rightarrow}$ *is weaker than* $\overset{T}{\Rightarrow}$ *and* $\overset{T}{\Rrightarrow}$.

Appendix B

Review: the logic of belief

> *The Sicilian smiled and stared at the wine goblets. "Now a great fool," he began, "would place the wine in his own goblet, because he would know that only another great fool would reach first for what he was given. I am clearly not a great fool, so I will clearly not reach for your wine."*
>
> *"That's your final choice?"*
>
> *"No. Because you knew I was not a great fool, so you would know that I would never fall for such a trick. You would count on it. So I will clearly not reach for mine either."* [Gol73, p. 157]

The Sicilian's great effort went into reasoning about the beliefs of his opponent, including his opponent's beliefs about his own beliefs, and so on. His watertight reasoning is an example of modal logic, the logic of belief. One way to reason about permissions and sharing is to reason about who believes what. We call participants in a distributed system *agents*, and the symbols that represent agents in logical expressions *principals*. Principals can also represent sets of agents, or one agent quoting another; these are called *compound principals*, and we discuss them in Section B.1. If Alice believes everything Bob believes (that is, Alice trusts Bob in every matter), then if Bob believes it is good to read a given file, Alice must believe the same. In this section, we develop a model for reasoning about logic in the presence of belief.

We begin with propositional logic. Assume there is a set of primitive (uninterpreted, independent) statements Σ.[1] For our purposes of access control, we consider primitive statements such as "it is good to write to file X." This interpretation turns an imperative command into a declarative proposition. The primitive statements may be connected with *and* ($\wedge$) and *not* ($\neg$) to form arbitrary formulas. The *or* ($\vee$) and *implies* ($\supset$) operators are abbreviations for longer formulas made of $\wedge$ and $\neg$.

Next we introduce a *modal* operator **believes**.[2] If σ is a formula and principal A represents agent Alice, A **believes** σ is a formula that can be read "Alice believes σ is true." In time, we will introduce multiple **believes** operators, one per principal. For now, we would like to build a *model* that helps us understand which formulas A believes; that is, for which σ do we have A **believes** σ?

[1] Figure 4.1 provides a table of sets and variable notation used in this paper.

[2] In conventional modal logic, A **believes** σ is written $\Box_A \sigma$.

To model this logic, we build a *Kripke structure*. A Kripke structure is a tuple of sets $\mathcal{M} = \langle W, I, J \rangle$. The members of set W represent *possible worlds*. The function I maps a primitive proposition (s) to the set of worlds where it is true, and the function J maps a principal to a relation on worlds in W. Together, I and J determine the truth value of every formula in every world in W; we describe them in more detail shortly.

First, some intuition: A principal A living in world w_0 considers some other set of worlds possible, and if a formula σ is true in each of those other worlds, then A believes the formula. The interesting thing about possible worlds is that the set of worlds A considers possible captures what she does not know: if a statement σ appears in one possible world and $\neg\sigma$ appears in another, then A knows neither σ nor $\neg\sigma$. As far as she is concerned, σ could go either way, because A cannot tell which of the possible worlds she actually is in.

When we write $\mathcal{M}, w_0 \models \sigma$ (pronounced "$\mathcal{M}$ at w_0 *models* σ"), we mean that in model $\mathcal{M}$ at world w_0, the formula σ is true. The mapping I tells us immediately about the truth of primitive propositions at different worlds, but we wish to determine the truth of arbitrary statements σ, including propositional connectives and our modal operators ($\sigma = A\,\mathbf{believes}\,\tau$). We illustrate with an example structure, shown in Figure B.1.

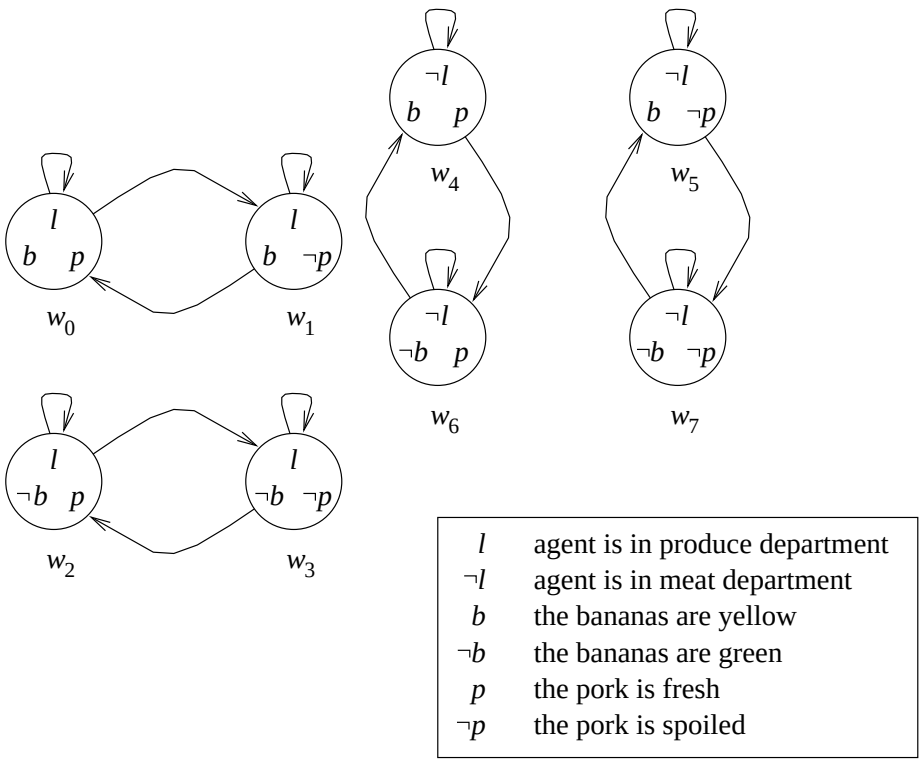


Figure B.1: *A model of eight worlds (circles), illustrating the relationship between the accessibility relation (arrows) and the the modal operator (A* ***believes****).*

The model contains three primitive statements, l, b, and p. The statement l means that our agent Alice (A) is in the produce department of a grocery store. Its negation, $\neg l$, means that Alice is in the meat department (it's a small store). The b primitive means that the

store's bananas are yellow, and the p primitive means that the store's pork is fresh.

Recall the three parts of a model, $\langle W, I, J\rangle$. W is the set of possible worlds; in our case, since there are three primitive statements, there are at most eight: $W = \{w_0, w_1, \ldots w_7\}$. I is a relation that defines which primitive statements are true at which worlds. In our example, $I(b) = \{w_0, w_1, w_4, w_5\}$, since the bananas are only yellow in those four worlds. Finally, J is a function that maps principals to relations. Because we have only one principal (Alice), J has only one mapping, written $J(A)$. The relation $J(A)$ is depicted with arrows in the diagram. For example, $\langle w_0, w_1\rangle \in J(A)$; that is, when the actual world is w_0, w_1 is a world Alice considers possible. In our example, it happens that Alice considers two worlds possible from each world.

Assume for a moment that the actual world is in fact w_0: Alice is in the produce department, the bananas are yellow and the pork is fresh. If Alice were omniscient, she would consider only w_0 possible, for that is indeed the state of things. Alice, however, is merely a shopper. She cannot see from the produce department what is going on in the meat department, and thus she cannot tell if the pork is fresh. She must also consider possible world w_1, where the pork is spoiled. She knows for certain her own location, though, so she can ignore worlds $w_4 \cdots w_7$. Because she is in the produce department and can see the bananas, she can also ignore worlds w_2 and w_3 in which the bananas are green.

We have explained the two arrows emanating from world w_0. The other arrows in the diagram, comprising the relation $J(A)$, communicate the same sort of information about any other state of affairs. For example, if the actual world were w_1 (the pork is in fact spoiled), Alice considers just the same worlds w_0 and w_1 possible, and for the same reasons.

Now that you have the intuition behind the Kripke structure, we can formally define when various statements are true. Primitive propositions are easy: the casual definition of I above becomes:

$$\mathcal{M}, w_0 \models s \qquad \text{when } w_0 \in I(s)$$

This definition can be read "Statement s is true at world w_0 in model M when w_0 is in the set $I(s)$."

What about formulas constructed from the propositional connectives $\wedge$ and $\neg$? The truth of some complex formula σ in a world is completely determined by the truth of its primitive propositions, which the model defines by the mapping I. So we can formally define an *extension* function $\mathcal{E}$ to extend the definition of I to arbitrary formulas. $\mathcal{E}$ is defined recursively starting with I, and extends as you would expect for the propositional connectives:

$$\begin{aligned}\mathcal{E}(s) &= I(s)\\ \mathcal{E}(\neg\sigma) &= W - \mathcal{E}(\sigma)\\ \mathcal{E}(\sigma \wedge \tau) &= \mathcal{E}(\sigma) \cap \mathcal{E}(\tau)\end{aligned}$$

Not surprisingly, $\neg\sigma$ holds in exactly those worlds where σ does not, and $\sigma \wedge \tau$ holds in exactly those worlds where both subformulas hold. Take a look at the example structure and convince yourself that $\mathcal{E}(b \wedge \neg p) = \{w_1, w_5\}$.

We embarked on this journey to discover when Alice believes various statements, so we need to find out when the model supports formulas including our modal belief operator.

The natural intuition is that Alice should believe a statement whenever it is true in *every* world Alice considers possible. To recall our example, b is true (the bananas are yellow) in every world Alice considers possible from w_0, so $\mathcal{M}, w_0 \models A \,\mathbf{believes}\, b$. But because Alice considers w_0 and w_1 possible, she considers both p and $\neg p$ possible; and so she can believe neither; hence we have $\neg(A \,\mathbf{believes}\, p)$ and $\neg(A \,\mathbf{believes}\, \neg p)$ at world w_0. (You can think of this situation as representing Alice's "silence" on the matter of p. Even though Alice asserts neither p nor $\neg p$, every formula is assigned a truth value. It is just that both $A \,\mathbf{believes}\, p$ and $A \,\mathbf{believes}\, \neg p$ are false.)

With this intuition, we fill out the definition of $\mathcal{E}$ to mention formulas containing our modal operator $A \,\mathbf{believes}$:

$$\mathcal{E}(A \,\mathbf{believes}\, \sigma) = \{w | J(A)(w) \subseteq \mathcal{E}(\sigma)\}$$

$J(A)(w)$ denotes the set of worlds that A considers possible from w.[3] So when σ is true in every one of these worlds (i.e., $J(A)(w) \subseteq \mathcal{E}(\sigma)$), then A believes σ (i.e. $A \,\mathbf{believes}\, \sigma$).

Of course, security is not very interesting in a world with only one agent. To introduce a second principal, we simply add a new relation $J(B)$ to our model. Now we can reason about what Bob believes ($B \,\mathbf{believes}\, \sigma$), and even about what Alice believes about what Bob believes ($A \,\mathbf{believes}\, B \,\mathbf{believes}\, \sigma$). (In our example, we could certainly discuss Alice's beliefs about her own beliefs, but for our application to access control, that is not very interesting.)

B.1 Compound principals

It is also possible to talk about *compound principals*. Lampson et al. define two operators on principals that can be used to make new compound principals. The first is fairly easy to describe: the principal $A \wedge B$ believes only things that both A and B believe. We can define a new possible-worlds relation for the compound principal in terms of the relations for A and B. To do this, we extend the mapping J to a new mapping $\mathcal{R}$ whose domain includes compound principals. Like the definition of $\mathcal{E}$, $\mathcal{R}$ is defined recursively starting with J:

$$\begin{aligned} \mathcal{R}(A) &= J(A) \\ &\quad \forall \text{ primitive principals } A \\ \mathcal{R}(\mathcal{A} \wedge \mathcal{B}) &= \mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{B}) \\ &\quad \forall \text{ arbitrary principals } \mathcal{A}, \mathcal{B} \end{aligned}$$

And R replaces J's role in the definition of $\mathcal{E}$:

$$\mathcal{E}(\mathcal{A} \,\mathbf{believes}\, \sigma) = \{w | \mathcal{R}(\mathcal{A})(w) \subseteq \mathcal{E}(\sigma)\}$$

That set union operation is surprising! What's going on? Recall that the more worlds an agent considers possible, the less the agent believes. In our example structure, Alice could not believe p because she considered world w_1 possible, where p was false. Likewise, by taking the union of the relations for principals A and B to get the relation for the

[3] Formally, $J(A)(w) = \{w' | \langle w, w' \rangle \in J(A)\}$.

compound principal $A \wedge B$, we ensure that the compound principal is at least as ignorant as either of A or B. If A and B disagree on any statement σ, then $A \wedge B$ can see both worlds where σ is true and worlds where it is false, so $A \wedge B$ can have neither belief.

The second operator for forming compound principals is written $B|A$, and pronounced "B *quoting* A." ("Quoting" may seem an odd choice of words when talking about belief; however, when we translate our terminology into that of Lampson et al., it reads more naturally.) This principal captures B's beliefs about A's beliefs: $(B|A)\,\mathbf{believes}\,\sigma$ should be synonymous with $B\,\mathbf{believes}\,(A\,\mathbf{believes}\,\sigma)$.

The relation for the compound principal $B|A$ is the composition of the relations of B and A:

$$\mathcal{R}(B|A) = \mathcal{R}(B) \circ \mathcal{R}(A)$$

What is the intuition for using composition? Suppose we have $\mathcal{M}, w_0 \models B|A\,\mathbf{believes}\,\sigma$: At world w_0, Bob (agent B) believes Alice believes σ. That means that at every world Bob considers possible from w_0 ($\mathcal{R}(B)(w_0)$), Alice believes σ. But Alice only believes σ at those worlds if σ is true at every world Alice can see from those worlds:

$$\bigcup_{w' \in \mathcal{R}(B)(w_0)} \mathcal{R}(A)(w')$$

The composition $\mathcal{R}(B) \circ \mathcal{R}(A)$ relates w_0 to just this set. So $B|A\,\mathbf{believes}\,\sigma$ is true at w_0 exactly when σ is true in every world reachable from w_0 by the composited relation given above as $\mathcal{R}(B|A)$.

B.1.1 The nature of principal relations

Now that we have a formal structure for discussing the beliefs of principals, let us consider what kinds of beliefs are reasonable, and how principals' beliefs should be related to one another's.

Recall our example structure, where in any world, Alice was either ignorant (had no belief) about either the pork or ignorant about the bananas. The first observation is that agents do not need to believe every true thing; statements about which they have neither a positive nor a negative belief represent a fact about which the agent is ignorant.

Furthermore, observe that Alice never believed anything false: in every world, if $A\,\mathbf{believes}\,\sigma$, σ also held in that world. In the parlance of modal logic, we would say Alice's belief is actually *knowledge*: although she does not have all knowledge, everything she believes is in fact true. Why was this the case? Notice that Alice's possible-worlds relation is reflexive: for every world Alice's relation includes an edge pointing back to that world. That is why Alice cannot believe anything false. If σ is not true in a given world, Alice cannot believe σ there, because the definition

$$\begin{aligned} \mathcal{M}, w \models A\,\mathbf{believes}\,\sigma \;\; &\mathbf{iff} \;\; w \in \mathcal{E}(A\,\mathbf{believes}\,\sigma) \\ &\mathbf{iff} \;\; \mathcal{R}(A)(w) \subseteq \mathcal{E}(\sigma) \end{aligned}$$

precludes it.

In modeling access control in the presence of arbitrary principals, however, we should certainly expect that some principals will believe (or at least claim to believe) untrue things. So we make no restriction of reflexivity on the relation that defines a principal's beliefs. Indeed, a principal may have an empty relation at a world: it may consider *no* worlds possible! In that case, at that world, the agent considers every statement true, since every statement is true in all of the zero worlds the agent considers possible. Indeed, the agent believes *false*. The agent's reasoning has become inconsistent; other agents would be wise not to follow this agent's beliefs.

B.1.2 Trust

Agents following one another's beliefs is exactly how we model trust. If Alice establishes that she believes everything Bob believes, then Alice does not have to be present for Bob to read one of her files: if Bob claims that reading the file would be good, Alice must agree, and the file server grants the request. To capture this trust, we observe that Alice is "less ignorant" than Bob: she believes everything Bob believes, and then perhaps more (on which Bob may remain silent). Therefore, from any actual world, Alice should consider possible a subset of the worlds Bob considers possible. When $\mathcal{R}(A) \subseteq \mathcal{R}(B)$, Alice says everything Bob says; if she says even more, it is because she disregards some possible world that leaves Bob's belief ambiguous. You should convince yourself that if Bob believes σ, Alice has to believe the same thing, for she considers possible only a subset of the worlds Bob considers possible.

Appendix C

Review: the original Calculus for Access Control

This section contains an introduction to the Calculus for Access Control due to Abadi, Lampson, et al.The reader familiar with it may skip to the next section. We have preserved here the names used for formulas in [LABW92]. We explicitly name formulas L1–L3, which are mentioned in passing in [LABW92, p. 273], and formulas A1–A4, which are mentioned in [ABLP93, pp. 712, 714, and 718].

In the preceding section, we introduced an instance of modal logic: propositional logic plus some modal operators capture the possibly ignorant, possibly false beliefs of fallible principals. The semantics we presented, based on Kripke structures, is exactly that used by Abadi to justify the calculus for access control. We introduced the semantics first, though, because conventionally the semantics is the "intuitive model" of the world, and the logic is a system for discovering theorems (statements that are true in every model) and reasoning from premises to conclusions that must appear in the model.

To apply modal logic to access control, Abadi et al. rename the operators. First, "believes" is renamed "**says**." This is meant to capture the notion that the logic is *performative*: sometimes when a principal says something, that something becomes true. The act of saying to a file server that a file should be modified, given that the file server believes you, causes that file to indeed be modified. This renaming makes the quoting operator sound more natural: $B|A$ is Bob quoting Alice. $B|A$ **says** s is meant to be a synonym for B **says** A **says** s. "Belief" is still useful intuition, however. The operator is the same; Bob's belief in σ can be inherited by Alice without Alice actually uttering σ.

A logic is a system of axioms and proof rules that let one reason from premises to conclusions: if the premise holds in a model, the conclusion holds as well. The logic of the Calculus is *sound* in that any conclusion proven in the logic holds in the model, but it is not *complete*: there are statements that are true in every model that cannot be proven in the logic. Abadi suggests that in fact the model may be *undecidable*: no logic system is adequate to prove every valid statement of the model.

The logic of access control is the same (up to variations in notation) as the conventional modal logic system K_n. The subscript n indicates that there are multiple modal operators [HC96, FHMV95, p. 51]. We present that system here.

First, we write $\vdash \sigma$ if a statement σ is valid in the logic: either taken as an axiom, or

provable as a theorem from other axioms and the proof rules. We prove theorems using the following:

$$\text{If } \sigma \text{ is a tautology of propositional calculus,}$$
$$\text{then } \vdash \sigma \qquad \text{(Axiom S1)}$$

The axiom lets us pull in the theorems of propositional calculus without explicitly mentioning the axioms and proof rules that produce them.

$$\frac{\vdash \sigma \qquad \vdash \sigma \supset \tau}{\vdash \tau} \qquad \text{(Rule S2)}$$

The proof rule (modus ponens) says that if both σ and the implication $\sigma \supset \tau$ are valid (provable), then τ is provable as well. It lets us prove theorems about formulas that include the modal operators (**says**) by reasoning from premises to conclusions.

We also have the *Distribution Axiom* (known in modal logic as the axiom **K**, from which the name of the system K_n derives):

$$\vdash A \,\mathbf{says}\, (\sigma \supset \tau) \supset (A \,\mathbf{says}\, \sigma \supset A \,\mathbf{says}\, \tau) \qquad \text{(Axiom S3)}$$

Intuitively it means that agents understand and believe all of the consequences of their beliefs. Furthermore, they believe every theorem:

$$\forall A, \ \frac{\vdash \sigma}{\vdash A \,\mathbf{says}\, \sigma} \qquad \text{(Rule S4)}$$

That is, agents know all of the theorems of the logic.

There is a subtle but important distinction between implication in the metalogic (the proof rule above) and implication in the logic. The logical symbol $\vdash$ means that the premises on its left prove the conclusions on its right. The proof rule condition $\vdash \sigma$ means that no premises are required to prove σ; that is, σ is a theorem. When that is true, we may conclude $\vdash A \,\mathbf{says}\, \sigma$: it is proven that $A \,\mathbf{says}\, \sigma$.

In contrast, the corresponding statement in the logic (not the metalogic) does not hold. The statement $\nvdash \sigma \supset A \,\mathbf{says}\, \sigma$ is read "it is not provable that σ implies $A \,\mathbf{says}\, \sigma$." The premise of the implication is an arbitrary statement σ (unlike the theorem $\vdash \sigma$ in the proof rule); it is not true that principals say every true statement. They say every theorem (those statements true in every world), but not every true statement (those statements true in the actual world from which the statement is being uttered).

C.1 The calculus of principals

The symbol $=$ is an equivalence relation on principals; by $A = B$ we mean that A and B have the same relation and therefore the same beliefs.[1] (Later in the paper we also use $=$ to denote set equality; its use should be clear from context.)

[1] Abadi et al. "note that A and B can have the same beliefs without having the same possible worlds relation; however, because principals are identified by their relations in the semantics, we define equality in terms of relations." This is only possible if the model has two distinct worlds in W that belong to all the same I sets; that is, the model has two separate but indistinguishable worlds.

We have presented the logical tools for reasoning about formulas of statements. Recall that we can also combine principals into principal formulas. For example, $A \wedge B$ is the principal that believes (says) only things that A and B agree upon. In the logic, $A \wedge B$ is defined in terms of its relationship to statements:

$$\vdash (A \wedge B)\,\mathbf{says}\,\sigma \equiv (A\,\mathbf{says}\,s) \wedge (B\,\mathbf{says}\,\sigma) \qquad \text{(Definition P1)}$$

Principal conjunction is associative, commutative, and idempotent:

$$\vdash (A \wedge B) \wedge C = A \wedge (B \wedge C) \qquad \text{(Axiom P4)}$$
$$\vdash A \wedge B = B \wedge A \qquad \text{(Axiom P4)}$$
$$\vdash A \wedge A = A \qquad \text{(Axiom P4)}$$

Quoting $(B|A)$ is defined as:

$$\vdash (B|A)\,\mathbf{says}\,\sigma \equiv B\,\mathbf{says}\,(A\,\mathbf{says}\,\sigma) \qquad \text{(Definition P2)}$$

In a sense, the quoting operator "curries" a **says** operation from the propositional formula into the principal formula, so that one can talk about a principal quoting another without yet mentioning the specific statement being quoted.

Quoting is associative and distributes over conjunction in both arguments:

$$\vdash (A|B)|C = A|(B|C) \qquad \text{(Axiom P5)}$$
$$\vdash A|(B \wedge C) = (A|B) \wedge (A|C)$$
$$\vdash (A \wedge B)|C = (A|C) \wedge (B|C) \qquad \text{(Axiom P6)}$$

C.2 The "speaks for" relation

A central concept of the calculus is the "speaks for" relation ($\Rightarrow$), which defines a partial order over all principals. This relation encodes the notion of one principal trusting another that we introduced in Section B.1.2. The statement $B \Rightarrow A$ is read "B speaks for A," and means that whenever B says something, A certainly agrees. Formally, we define

$$\vdash (B \Rightarrow A) \equiv (B = B \wedge A) \qquad \text{(Definition P7)}$$

Why is this the case? If A trusts B, then A says everything B says. So the set of things $B \wedge A$ say must be the same as the set of things B says. It cannot be greater, by its semantic definition in Section B.1, and it cannot be less, or else there is something B says that A does not.

From the definition we can derive:[2]

$$\vdash (B \Rightarrow A) \supset ((B\,\mathbf{says}\,\sigma) \supset (A\,\mathbf{says}\,\sigma)) \qquad \text{(Theorem P8)}$$

[2]Surprisingly, Abadi et al. drop Definition P7 and instead treat Theorem P8 as an axiom. Doing so precludes theorems with conclusions containing $\Rightarrow$, since we are left with no axioms with $\Rightarrow$ in the conclusion. In fact, Theorem P8 requires only the weaker operator $\rightarrow$ in its premise, which we discuss in Section 4.1.1.

When $B \Rightarrow A$, B is a stronger principal than A in the sense that B can do everything A can do (by making A believe the appropriate performative statement), and perhaps more.

Using the associativity of $\wedge$ for principals, it is clear that $\Rightarrow$ is a transitive relation:

$$\vdash (B \Rightarrow A) \wedge (C \Rightarrow B) \supset C \Rightarrow A \qquad \text{(Theorem L1)}$$

(The $\wedge$ in the theorem is that for statements. We would like to use a different symbol for clarity, but we stick with the notation of Abadi et al. here.) Both the $\wedge$ and $|$ operators on principals are monotonic with respect to $\Rightarrow$:

$$\vdash (A \Rightarrow B) \supset ((A \wedge C) \Rightarrow (B \wedge C)) \qquad \text{(Axiom L2)}$$

$$\begin{array}{l} \vdash (A \Rightarrow B) \supset ((A|C) \Rightarrow (B|C)) \\ \vdash (A \Rightarrow B) \supset ((C|A) \Rightarrow (C|B)) \end{array} \qquad \text{(Axiom L3)}$$

With the speaks-for relation, we can finally see why quoting is a useful operation. One can let $C|B \Rightarrow A$, so that C can only speak for A when it quotes B. Without quoting, we would need a formal accounting for universal quantification over formulas: $\forall \sigma, C \,\mathbf{says}\, B \,\mathbf{says}\, \sigma \supset A \,\mathbf{says}\, \sigma$.

The semantics of $\Rightarrow$ falls out fairly directly. Definition P7 requires that

$$\begin{array}{l} \mathcal{M}, w \models B \Rightarrow A \\ \qquad \textbf{iff } \mathcal{R}(B) = \mathcal{R}(B \wedge A) = \mathcal{R}(B) \cup \mathcal{R}(A) \\ \qquad \textbf{iff } \mathcal{R}(A) \subseteq \mathcal{R}(B) \end{array}$$

Notice that the condition on the $\mathcal{R}$ relations is independent of the world w. So the extension function $\mathcal{E}$ is all-or-nothing for speaks-for formulas:

$$\mathcal{E}(B \Rightarrow A) = \begin{cases} W & \text{if } \mathcal{R}(A) \subseteq \mathcal{R}(B) \\ \varnothing & \text{otherwise} \end{cases} \qquad \text{(Definition A1)}$$

C.3 Access Control Lists

The speaks-for relation, because it is transitive, lets us reason broadly about how principals' beliefs affect one another. In the end, however, the server wants to convince itself that some primitive proposition s, perhaps to be interpreted "it is okay to change the contents of the file," is true. To support this, Abadi, Lampson et al. use the construct A *controls* s to indicate that principal A's beliefs about s are taken to be truth. It is defined as:

$$A \textit{ controls } s \equiv ((A \,\mathbf{says}\, s) \supset s) \qquad \text{(Definition A2)}$$

Now suppose B wants to write to the file that s describes, and the assumptions $\vdash B \Rightarrow A$ and $\vdash A$ *controls* s hold. Then the file server will be able to verify a proof of $\vdash s$, convincing itself that "it is okay to change the contents of the file."

Lampson et al. encode access control lists (ACLs) using *controls* assumptions:

$$\text{ACL}\,(O_1) = \left\{ \begin{array}{l} \vdash A \textit{ controls } s_{read}, \\ \vdash A \textit{ controls } s_{write}, \\ \vdash B \textit{ controls } s_{read} \end{array} \right\}$$

By adjusting which principals' assertions are believed, the ACLs allow or disallow agents to effect action.

C.4 Higher-level operators

The operating system that instantiates the calculus requires resource servers to construct and then verify all necessary proofs [WABL94]. Wobber calls it a *pull* model: it is the servers' job to pull in necessary assumptions and proof components needed to verify an agent's access. Building such proofs, when assumptions include speaks-for formulas with arbitrary combinations of $\wedge$ and $|$ operators, takes exponential time. To make the decision problem tractable, Lampson et al. define two high-level operators, **as** and **for** , in terms of the lower-level operators. Each operator is designed to reflect an idiomatic usage pattern of the calculus. The higher-level operators can combine in fewer ways than the lower-level operators, allowing an implementation to exploit characteristics such as associativity and idempotence. In the abstract, the operators can be treated as abbreviations and replaced by their definitions, and they do not affect the calculus. We cover them here to demonstrate the idioms they represent.

C.5 Roles and the "as" operator

Abadi et al. define a distinguished, disjoint set of principals called roles. By quoting a role, a principal restricts its own authority. For example, define the roles R_{user} and R_{admin} representing a person acting as a user and as an administrator, respectively. Suppose the ACLs in the system include $A|R_{\text{admin}}$ *controls* s_1 and $A|R_{\text{user}}$ *controls* s_2. In her daily work, Alice may step into her role as user by quoting R_{user}; when she needs to perform administrative tasks, Alice can explicitly quote R_{admin} to gain access to objects such as s_1 that mention her administrative role. More interestingly, Alice can delegate just one of her roles to another principal by arranging that $B \Rightarrow A|R_{\text{user}}$. Now Bob can do anything Alice could do as a user, but he cannot access her administrative resources. Roles can also be used to sandbox untrusted code. When running untrusted software, Alice might delegate to it only authority over $A|R_{\text{untrusted}}$, preventing the code from accessing the bulk of her resources.

The **as** operator stands for quoting when the quoted principal is a role (Axiom R1 in [LABW92]). In a sense, **as** adds strong typing, requiring that its right-hand argument be a role. In contrast to general principals, quoting is idempotent and commutative for roles, and all principals automatically speak for themselves in every role:

$$R|R = R \qquad \forall R \in \mathit{Roles} \qquad \text{(Axiom A3)}$$

$$R'|R = R|R' \qquad \forall R, R' \in \mathit{Roles} \qquad \text{(Axiom A4)}$$

$$A \Rightarrow A \textbf{ as } R \qquad \forall R \in \mathit{Roles} \qquad \text{(Axiom R2)}$$

By virtue of these special features of roles and its strong typing, the **as** operator takes on idempotence and commutativity. This helps make the access control problem tractable.

C.5.1 Semantics for Roles

The axioms above are not supported for general quoting, and yet **as** is simply an abbreviation for quoting. Therefore, the axioms must be justified by some restriction on the

possible-worlds relations of the roles themselves. First we define a special principal **1**, the *identity*, who believes everything that is true and nothing that is not:

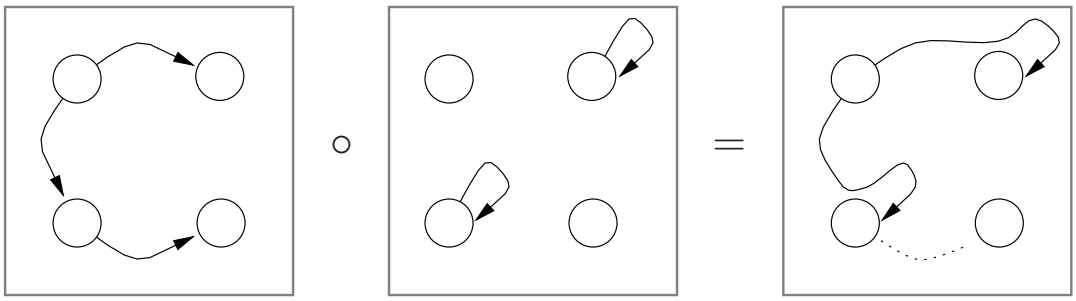
$$\mathcal{R}(\mathbf{1})(w) = w \qquad \forall w \in W$$

In any given world, **1** considers only that world possible. Therefore, it only tells the truth (the relation is reflexive), and it tells the whole truth (no world has multiple arrows, so it is confused about nothing). The identity serves as the most trusted role a principal can assume. Why? A **as 1** is shorthand for $A|\mathbf{1}$, so $\mathcal{R}(A \textbf{ as } \mathbf{1}) = \mathcal{R}(A) \circ \mathcal{R}(\mathbf{1}) = \mathcal{R}(A)$: the identity role does not limit A's authority at all.

All roles are principals whose relations are constrained as follows:

$$\mathcal{R}(R_1) \subseteq \mathcal{R}(\mathbf{1})$$

This means that the role relation may contain some edges $\langle w, w \rangle$ and not others, but no edges that take one world to another world. A role, when composed with another principal's relation, cannot expand the set of worlds the principal considers possible, only reduce it. See Figure C.1 for an illustration.



An arbitrary principal relation $\mathcal{R}(A)$ ... ... composed with a role relation $\mathcal{R}(R)$ ... ... gives a new relation that is always a subset of $\mathcal{R}(A)$.

Figure C.1: *Roles reduce relations with which they are composed.*

We are now prepared to justify the axioms for roles. The first property is idempotence. $\mathcal{R}(R_1)$ takes each world to either itself or nowhere, so composing $\mathcal{R}(R_1)$ with itself should do the same. The second property is commutativity. An arrow appears in $\mathcal{R}(R_1) \circ \mathcal{R}(R_2)$ exactly when it appears in $\mathcal{R}(R_1) \cap \mathcal{R}(R_2)$, and $\cap$ is commutative. Finally, $A \Rightarrow A$ **as** R_1 is automatically true when R_1 is a role. Why? Composing $\mathcal{R}(R_1)$ onto $\mathcal{R}(A)$ cannot introduce any new worlds (since the arrows of $\mathcal{R}(R_1)$ are all reflexive), but may eliminate worlds (when $\mathcal{R}(R_1)(w) = \varnothing$). Hence

$$\mathcal{R}(A) \circ \mathcal{R}(R_1) \subseteq \mathcal{R}(A)$$

and we conclude $A \Rightarrow A$ **as** R_1.

C.6 Delegation and the "for" operator

Besides encoding roles, quoting can be used to encode delegations to trusted principals in a restricted way. Here is the problem: Imagine that both Alice and Bob log in to machine M. Using just the speaks-for operator, Alice might establish that $M \Rightarrow A$ and Bob that $M \Rightarrow B$. But then when Bob (sitting at his terminal to machine M) tries to read a file that only A has permission to read, M would say the request, and the server would reason that A believed it. In this situation, the access-control system cannot help the server reason about whether the file should be read, since M has not provided enough information.

Instead, A could require that M explicitly mention A whenever it makes requests on A's behalf: $M|A \Rightarrow A$. Now when M is working for B, it will be quoting B, not A, and A's file is safe. If M were corrupt, of course, it could still abuse the authority granted it by A. But quoting principals helps an honest M pass the right information to resource servers for access-control decisions.

Lampson et al. define a slightly more complicated concept of delegation from A to B, written as the compound principal B **for** A. The key idea behind delegation is that both the delegator A and the delegate B must take some explicit action for the delegation to take effect:

$$A \textbf{ says } B|A \Rightarrow (B \textbf{ for } A)$$
$$B|A \textbf{ says } B|A \Rightarrow (B \textbf{ for } A)$$

from which, using the definition of **for** in [LABW92, p. 295], we can conclude

$$(B|A) \Rightarrow (B \textbf{ for } A)$$

Then A installs B **for** A in ACLs for any resources it wishes to allow B to access on its behalf.

The difference between B simply taking care to always quote A and B receiving a delegation to B **for** A is subtle. In both cases, A must explicitly hand off authority to B. And in both cases, B has to take some explicit action to accept the delegation; in the first case, that action is to quote A, in the second, it must also make a separate statement accepting the delegation.

Like **as**, **for** seems to be introduced for its special properties, to enable a more efficient pull-style theorem-proving implementation.

We have completed our review presentation of the calculus due to Abadi, Lampson et al.

Appendix D

Review: The Simple Public Key Infrastructure

The Simple Public Key Infrastructure 2.0 (SPKI, pronounced "spooky") is an Internet Experimental Protocol created by Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen [EFL+99]. As its name suggests, it is designed to be a unifying standard for supporting public key authorization across the global Internet. We highlight here some of the features of SPKI relevant to this work.

First, SPKI's primary goal is to provide a server with evidence that the holder of a given cryptographic key is ultimately authorized for a request signed by that key. This goal contrasts with that of other public-key infrastructure efforts that attempt to bind keys to identities, and leave authorization to be handled in the conventional fashion by ACLs that map identity to authorization.

In this section, we review the types of certificates that SPKI supports, and outline the procedure used to determine whether a given certificate chain supports a requested operation.

D.1 Certificate types

SPKI defines its own certificate format, as well as an internal representation of certificates to which it can map other inputs, such as PGP certificates, X.509 certificates, or locally maintained ACL entries. Authorization results can be constructed from inputs providing information in one of three forms:

- $\langle \text{authorization}, \text{key} \rangle$
- $\langle \text{authorization}, \text{name} \rangle$
- $\langle \text{name}, \text{key} \rangle$

The first form coincides with SPKI's design philosophy of mapping keys directly to authorizations. Inputs of the latter two forms must ultimately be combined to form a $\langle \text{authorization}, \text{key} \rangle$ mapping to become useful.

Inputs of the first two forms are mapped into a data structure called a *5-tuple* for internal processing; inputs of the latter form are mapped into a data structure called a *4-tuple*.

D.2 The SPKI 5-tuple

A 5-tuple has the following fields:

- issuer: the public key granting the permission defined by the 5-tuple
- subject: a public key or name to which the permission is being granted
- delegation-control: a boolean value indicating whether this permission may be further delegated
- authorization: a set of primitive permissions being granted
- validity dates: a date range limiting the validity of this delegation

The intended meaning is that the issuer grants the subject the permission described in the authorization field for the duration of the validity dates. If the delegation-control bit is set, the subject may further delegate any or all of the permission to another subject.

The subject in a 5-tuple (or a 4-tuple, which we present shortly) may be a k-of-n threshold function. In this case, the permission is delegated to any principal that can prove it is authorized to speak for any k of the n "subordinate" subjects listed in the threshold function.

The authorization fields contain primitive permissions whose interpretation is left to the application employing the SPKI authorization engine. These permissions are represented using *auth tags*. Tags encode infinitely large sets of primitive statements in a form that permits a compact representation of certain subsets. Notably, a tag can represent only a set of primitive symbols; never a formula made from the negation or conjunction of primitive symbols. Tags admit a simple intersection algorithm that always yields a compact representation of the intersected set.

SPKI certificates may also indicate an on-line mechanism for verifying that the issuer considers a certificate still valid. Two of the checks, the certificate revocation list (CRL, a negative list of revoked certificates) and the timed revalidation (a positive list of still-valid certificates), are performed by consulting a list revised more frequently than the original certificate being checked. The one-time revalidation check, which "represents a validity interval of zero" [EFL+99, p. 21], is performed by contacting the specified server to verify that the server still approves the certificate.

D.3 The SPKI 4-tuple

Symbolic names are always interpreted relative to a globally unambiguous name, usually a public key. As a consequence, the definition of a symbolic name is never ambiguous; it is always the definition supplied by the key that grounds the name. The SPKI authors contrast

this situation with that of PGP, where symbolic names reside in a global namespace, and their meaning depends on the beholder and the "introducers" that the beholder trusts.

A symbolic name ultimately is defined as one or more keys, although a single 4-tuple may define a name in terms of a chain of other names grounded in a key. In that case, other 4-tuples must participate in the reduction of the name chain to a final key. A 4-tuple has the following fields:

- issuer: the public key defining this name in its private name space.
- name: the name being defined
- subject: a public key or name to which the name is bound.
- validity dates: a date range limiting the validity of this delegation

The intended meaning of a 4-tuple is that the issuer defines the symbolic name, when grounded by the issuer's key, to be equal to the key identified by the subject for the duration of the validity dates. It is easy to read this definition backwards. Note that a name definition tuple does not give the issuer control over the subject, but the subject control over any permission elsewhere granted to the grounded name "issuer: name." Hence a threshold subject is also meaningful as the subject of a 4-tuple; its use means that if a principal speaks for k of the n subordinate subjects, that principal also speaks for "issuer: name," and hence garners any permission granted to that name.

D.4 Tuple reduction

The SPKI access-control decision procedure is called "tuple reduction." Once the appropriate certificates for an access-control decision have been gathered, the on-line checks performed, and the certificates converted into internal tuples, the tuples are "reduced." If the reduction results in a 5-tuple issued by the server that grants the requested permission to the key that signed the request, then the request is authorized.

Reduction proceeds as follows. First, 4-tuples are reduced to resolve names. 4-tuples that define a name in terms of another grounded chain of names are reduced using 4-tuples that define a name in terms of a key. Eventually, 4-tuples of the former form are reduced to 4-tuples of the latter form. The validity date stored in the outcome of each reduction is the intersection of the validity dates of the 4-tuples being reduced.

Then the $\langle \text{name}, \text{key} \rangle$ bindings formed by the reduced 4-tuples are applied to resolve names in 5-tuples back to keys, again carrying validity dates through with intersection operations. This operation turns $\langle \text{authorization}, \text{name} \rangle$ 5-tuples into $\langle \text{authorization}, \text{key} \rangle$ tuples.

At this point, each 5-tuple represents a subject key (or threshold subject defined as a set of keys) with authorization to perform some set of actions on behalf of the issuer key. When two 5-tuples form a chain of delegation (the issuer of the second is the subject of the first, and the first tuple allows further delegation), the 5-tuples are reduced to a new tuple whose subject is the subject of the second tuple and whose issuer is the issuer of the first. The reduced tuple carries the intersection of the authorizations of the source tuples as its

authorization, and the intersection of the validity dates of the source tuples as its validity dates. Finally, the reduced tuple carries the same delegation control bit as the second tuple did. Think of the delegation control bit as the coupling on the back of a boxcar; if the first tuple lacks it, the cars cannot couple; if the second tuple lacks it, the cars may couple, but the resulting "super-car" will also lack a rear coupling.

Appendix E

Experimental data

In this chapter, I present the experimental data from which the tables in Chapter 12 derive. The index in Table E.1 connects the summary table in Chapter 12 to the experiment number of the raw data shown on the following pages.

On each page, I plot the data points observed. The plot legend groups the data points into a few categories; within each category, only the dependent variable labeled on the x axis changes from one experiment to the next. The *variables* section describes the parameters that characterize each category, and the *constants* section lists the parameters that were held constant for every data point plotted on that page.

I define the experimental variables below.

authenticateServer (boolean) In Snowflake experiments, the client expects and verifies the server's proof of the document's authenticity, and maps the issuer of that proof to an *final* principal in the client's `Prover`. A final principal is one that the client controls, such as a public key for which the `Prover`has the corresponding private key.

cacheContext (boolean) In SSL experiments, the SSL context object is reused for each request. This reuse amortizes the cost of loading certificates.

cacheSessions (boolean) In SSL experiments, the SSL session-caching optimization is

Table	Experiments
Table 12.2	23, 24
Table 12.3	25, 26
Table 12.4	17, 18, 3, 4
Table 12.5	10, 11
Table 12.6	1, 2
Table 12.7	19, 20
Table 12.8	13, 14, 15, 16
Table 12.9	8, 9

Table E.1: *The correspondence between summary tables and the experimental data they summarize*

enabled, amortizing the public-key encryptions required to establish a session.

cacheSigns (boolean) In Snowflake experiments, the server caches its signatures on documents until the modification date on the document file changes.

client When set to "fastget," the client is my trivial C HTTP client. Otherwise, the client is my Java client using Jetty stream- and header-parsing tools.

clientCachesProofs (boolean) In Snowflake RMI experiments, the client caches any proofs it generates in its `Prover`, amortizing the cost of the public-key encryption used to sign the delegation.

experimentType When set to "RMIExp," the experimental operation is an RMI transaction; otherwise, it is an HTTP transaction.

fileLength (integer, bytes) The length of user data returned by the experimental operation.

identicalRequests (boolean) In HTTP experiments, when this variable is false, the client increments the value in an extraneous header to ensure that the request differs from previous requests so that a proof of the authority of the previous request cannot be directly reused for the current request.

locality When this value is "local," the client and server process are colocated on the same host; when it is "remote," they are separated by a shared 10Mbps Ethernet segment.

numberOfConnections (integer) This value indicates the number of times the client connects to the server in a single run. A "run" is the set of experimental operations whose total wall-clock runtime appears as a data point in the plots that follow.

port (integer) The port number of the experimental server handling the request; this is an internal value determined by the socket and server variables.

protocol In HTTP experiments, this value indicates whether I use HTTP/1.0 or HTTP/1.1 requests.

registryService When this value is "TestRMIServer0," the RMI calls are transmitted on plain TCP sockets. When it is "TestRMIServer2," the RMI calls are transmitted using my `SshSocketFactory` sockets. When it is "TestRMIServer3," the calls use `SshSocketFactory` sockets, and the authority of the client is challenged using the Snowflake protocol.

requestsPerConnection (integer) In an HTTP experiment, the number of requests sent over a single connection before the connection is discarded. This variable is always 1 if the protocol is not HTTP/1.1.

server In an HTTP experiment, a value of "apache" indicates that the HTTP server is Apache. A value of "simple" indicates a simple Java web server using the `java.io.Socket` interface and trivial request parsing. A value of "Jetty" indicates a

Java Jetty server with either the standard Jetty file handler or our file servlet adapted to understand Snowflake HTTP.

serverCachesProofs (boolean) In an RMI experiment, the server caches authorization proofs received from the client to amortize transmission, parsing, and verification time.

signFiles (boolean) In Snowflake experiments, the server signs delegations proving the authority of files it serves to the client.

socket In HTTP experiments, a value of "plain" indicates a plain TCP socket, and a value of "SSL" indicates an SSL socket.

uri (string) In HTTP experiments, an internal value derived from fileLength.

useMacs (boolean) In Snowflake HTTP experiments, the client requests a secret MAC to enable inexpensive hash-based request authorization.

useSnowflake (boolean) In HTTP experiments, indicates that Snowflake is used to authorize the client's request.

Experiment 1

Variables:

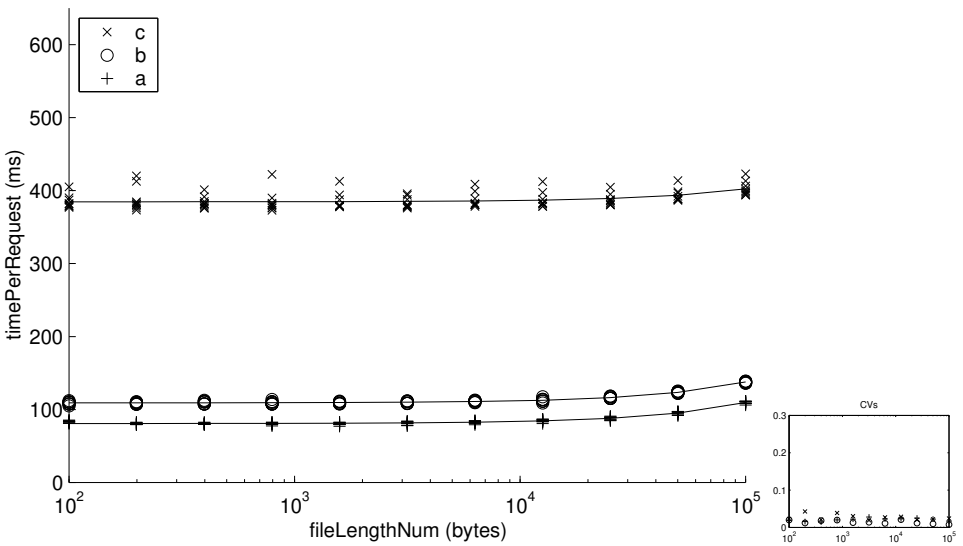
c identicalRequests=false useMacs=false authenticateServer=false
σ=10.4ms R^2=0.21 b_0=384.61 ± 0.0 ms b_1=186.98 ± 0.0 ms/MB

b identicalRequests=false useMacs=true authenticateServer=false
σ= 1.6ms R^2=0.97 b_0=109.15 ± 0.0 ms b_1=299.40 ± 0.0 ms/MB

a identicalRequests=true useMacs=true authenticateServer=false
σ= 1.9ms R^2=0.95 b_0=80.79 ± 0.0 ms b_1=301.49 ± 0.0 ms/MB

Constants:

authenticateServer	false
cacheSigns	false
numberOfConnections	10
port	8041
protocol	1.0
requestsPerConnection	1
server	simple
signFiles	false
socket	plain
useSnowflake	true
network locality	local



Experiment 2

Variables:

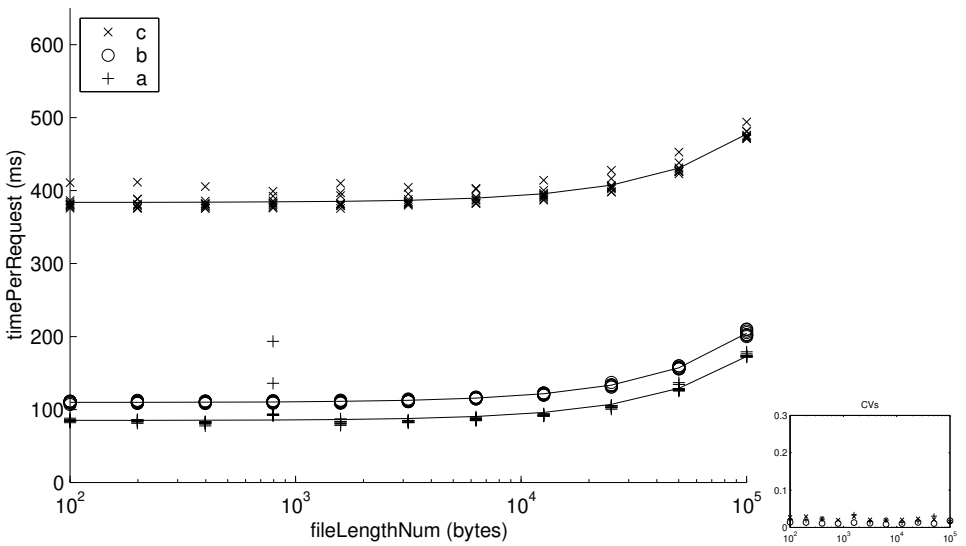
c identicalRequests=false useMacs=false authenticateServer=false
σ= 8.8ms R^2=0.91 b_0=383.77 ± 0.0 ms b_1=982.02 ± 0.0 ms/MB

b identicalRequests=false useMacs=true authenticateServer=false
σ= 1.7ms R^2=1.00 b_0=109.67 ± 0.0 ms b_1=994.38 ± 0.0 ms/MB

a identicalRequests=true useMacs=true authenticateServer=false
σ=12.7ms R^2=0.81 b_0=85.04 ± 0.0 ms b_1=918.10 ± 0.0 ms/MB

Constants:

authenticateServer	false
cacheSigns	false
numberOfConnections	10
port	8041
protocol	1.0
requestsPerConnection	1
server	simple
signFiles	false
socket	plain
useSnowflake	true
network locality	remote



Experiment 3

Variables:

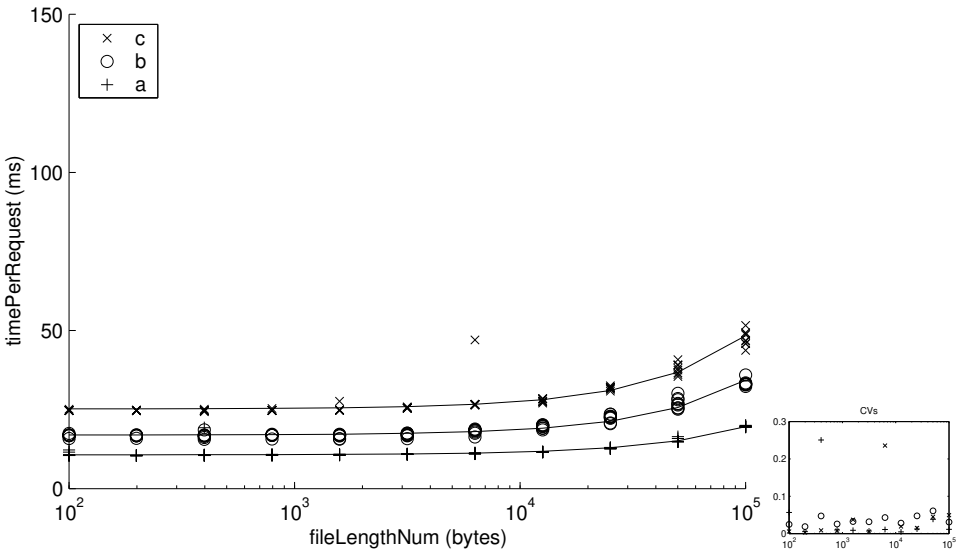
c server=Jetty
σ= 2.3ms R^2=0.90 b_0=25.22 ± 0.0 ms b_1=242.83 ± 0.0 ms/MB

b server=simple
σ= 1.0ms R^2=0.96 b_0=16.95 ± 0.0 ms b_1=181.94 ± 0.0 ms/MB

a server=apache
σ= 0.9ms R^2=0.89 b_0=10.70 ± 0.0 ms b_1=93.16 ± 0.0 ms/MB

Constants:

cacheContext	true
cacheSessions	true
numberOfConnections	20
protocol	1.0
requestsPerConnection	1
socket	plain
useSnowflake	false
network locality	local



Experiment 4

Variables:

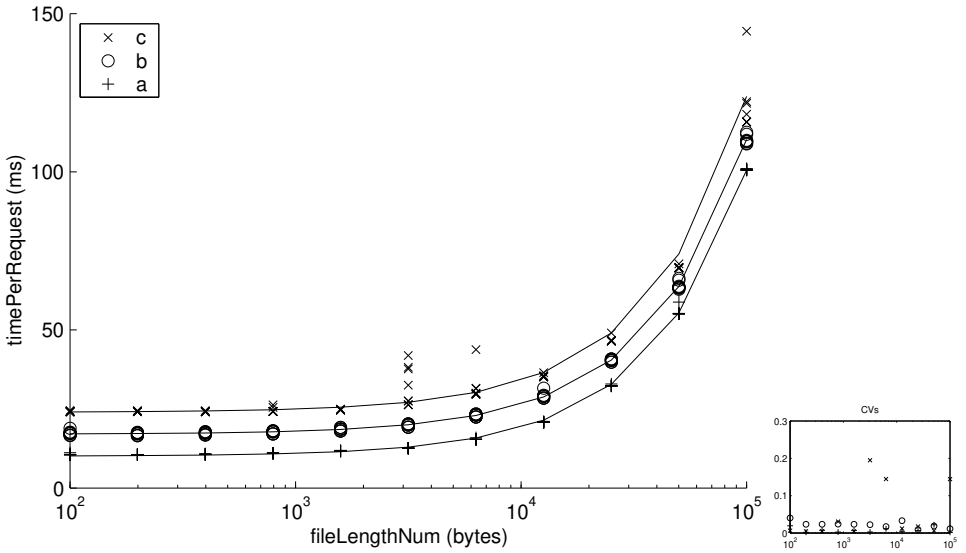
c server=Jetty
σ= 6.1ms R^2=0.96 b_0=23.98 ± 0.0 ms b_1=1047.05 ± 0.0 ms/MB

b server=simple
σ= 0.7ms R^2=1.00 b_0=17.07 ± 0.0 ms b_1=976.40 ± 0.0 ms/MB

a server=apache
σ= 0.5ms R^2=1.00 b_0=10.09 ± 0.0 ms b_1=948.42 ± 0.0 ms/MB

Constants:

cacheContext	true
cacheSessions	true
numberOfConnections	20
protocol	1.0
requestsPerConnection	1
socket	plain
useSnowflake	false
network locality	remote



Experiment 8

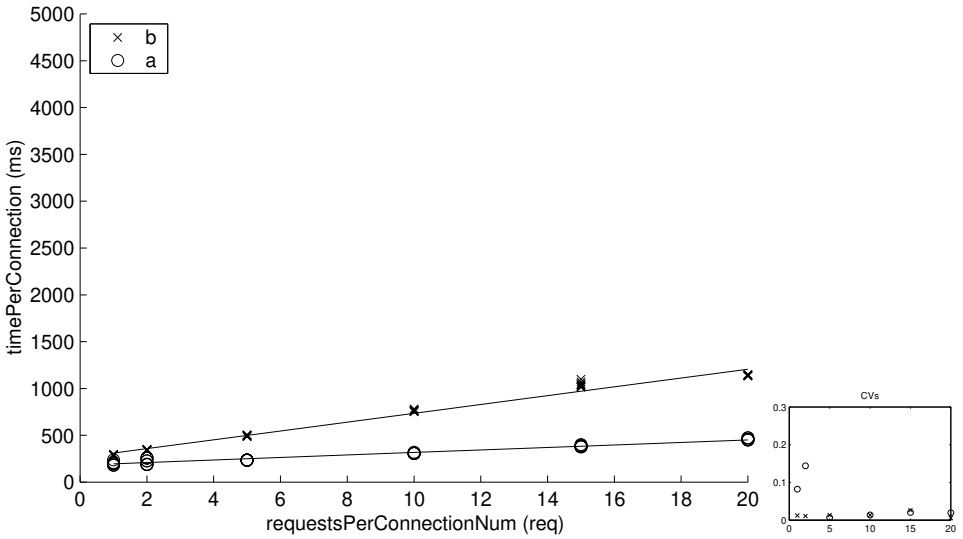
Variables:

b server=Jetty fileLength=000100
σ=45.2ms R^2=0.98 b_0=263.56 ± 0.2 ms b_1=47.13 ± 0.0 ms/req

a server=apache fileLength=000100
σ=17.5ms R^2=0.97 b_0=182.40 ± 0.0 ms b_1=13.46 ± 0.0 ms/req

Constants:

cacheContext	true
cacheSessions	true
fileLength	000100
numberOfConnections	10
protocol	1.1
socket	SSL
uri	/timing/data-000100.txt
useSnowflake	false
network locality	local



Experiment 9

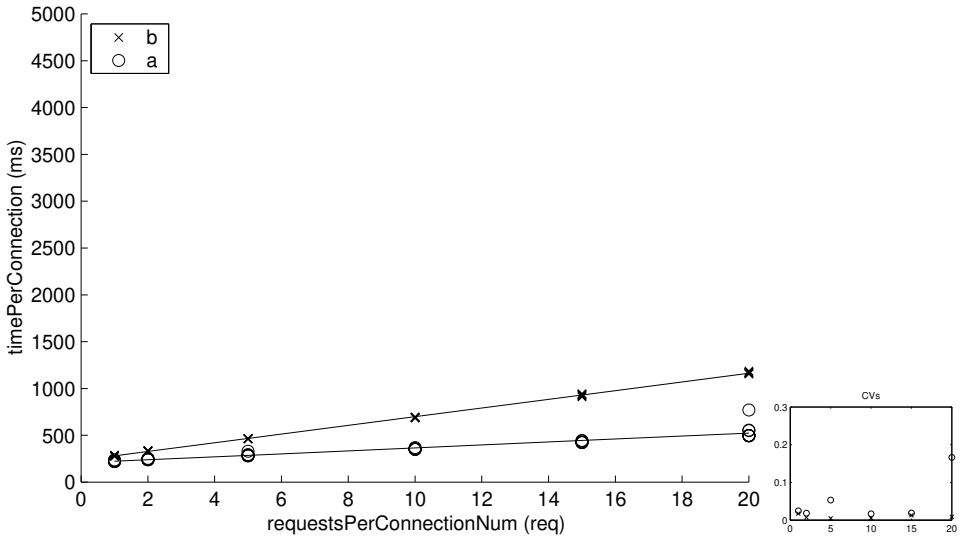
Variables:

b server=Jetty fileLength=000100
σ= 8.5ms R^2=1.00 b_0=235.04 ± 0.0 ms b_1=46.39 ± 0.0 ms/req

a server=apache fileLength=000100
σ=37.6ms R^2=0.90 b_0=207.16 ± 0.1 ms b_1=15.85 ± 0.0 ms/req

Constants:

cacheContext	true
cacheSessions	true
fileLength	000100
numberOfConnections	10
protocol	1.1
socket	SSL
uri	/timing/data-000100.txt
useSnowflake	false
network locality	remote



Experiment 10

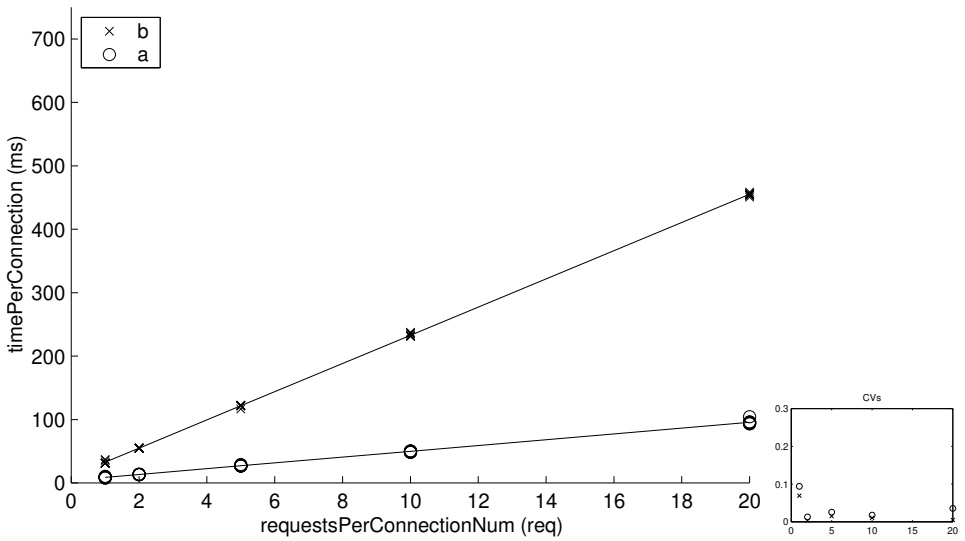
Variables:

b server=Jetty fileLength=000100
σ= 2.0ms R^2=1.00 b_0=10.39 ± 0.0 ms b_1=22.24 ± 0.0 ms/req

a server=apache fileLength=000100
σ= 1.6ms R^2=1.00 b_0= 4.27 ± 0.0 ms b_1= 4.56 ± 0.0 ms/req

Constants:

cacheContext	true
cacheSessions	true
numberOfConnections	10
protocol	1.1
socket	plain
useSnowflake	false
network locality	local



Experiment 11

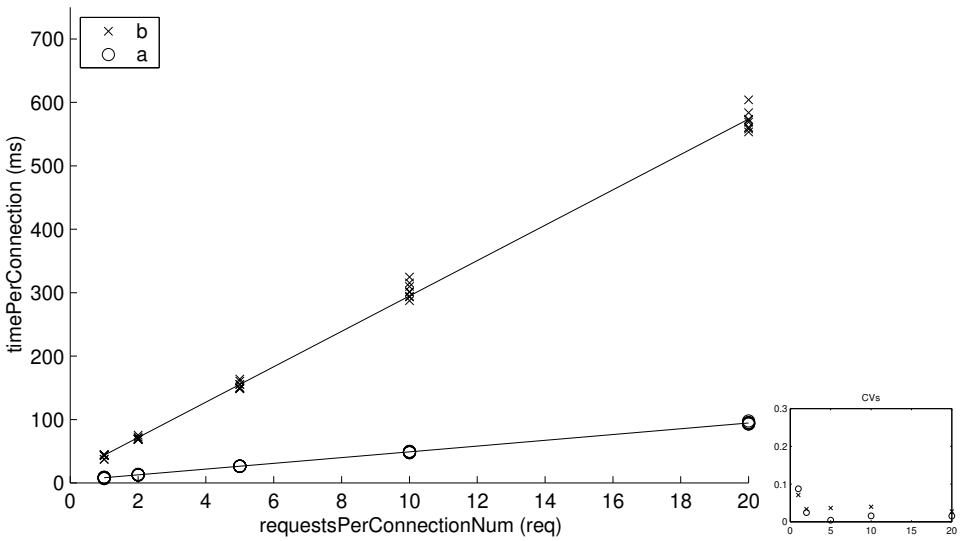
Variables:

b server=Jetty fileLength=000100
σ= 9.9ms R^2=1.00 b_0=15.76 ± 0.0 ms b_1=27.90 ± 0.0 ms/req

a server=apache fileLength=000100
σ= 0.8ms R^2=1.00 b_0= 3.64 ± 0.0 ms b_1= 4.54 ± 0.0 ms/req

Constants:

cacheContext	true
cacheSessions	true
numberOfConnections	10
protocol	1.1
socket	plain
useSnowflake	false
network locality	remote



Experiment 13

Variables:

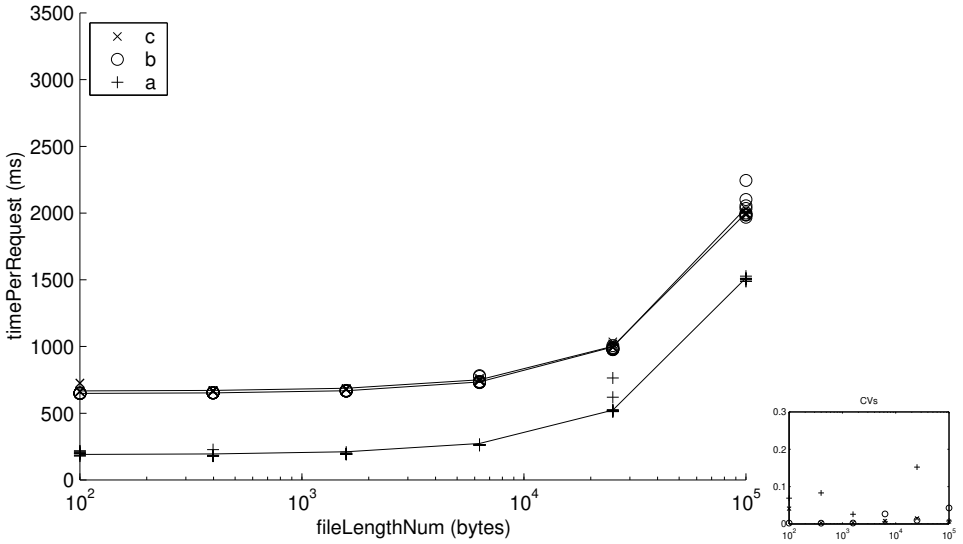
c server=apache cacheContext=false cacheSessions=false
σ=14.7ms R^2=1.00 b_0=666.34 ± 0.0 ms b_1=13977.52 ± 0.0 ms/MB

b server=apache cacheContext=true cacheSessions=false
σ=35.8ms R^2=1.00 b_0=647.17 ± 0.1 ms b_1=14590.47 ± 0.0 ms/MB

a server=apache cacheContext=true cacheSessions=true
σ=38.8ms R^2=0.99 b_0=190.33 ± 0.1 ms b_1=13872.77 ± 0.0 ms/MB

Constants:

numberOfConnections	10
protocol	1.1
requestsPerConnection	1
socket	SSL
useSnowflake	false
network locality	local



Experiment 14

Variables:

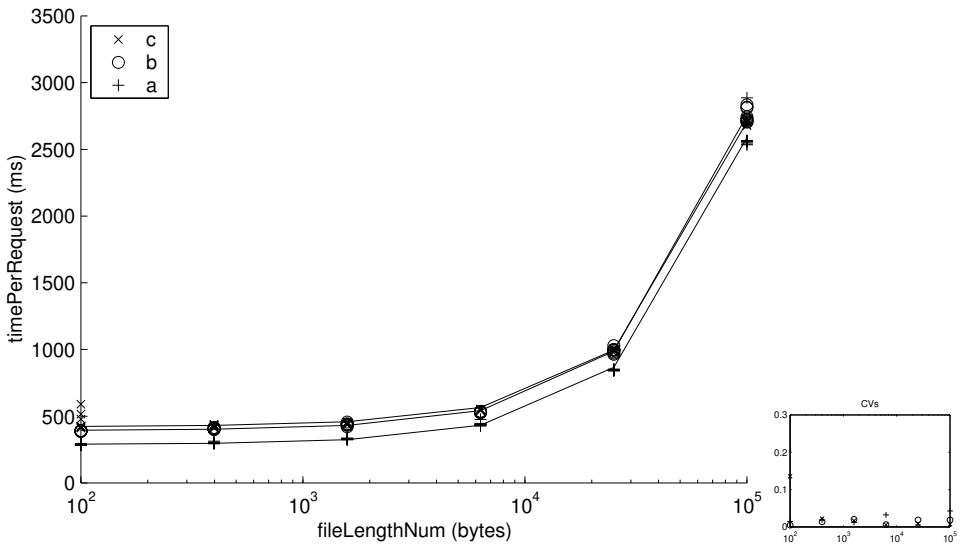
c server=Jetty cacheContext=false cacheSessions=false
σ=30.6ms R^2=1.00 b_0=422.05 ± 0.0 ms b_1=23943.09 ± 0.0 ms/MB

b server=Jetty cacheContext=true cacheSessions=false
σ=23.0ms R^2=1.00 b_0=393.45 ± 0.0 ms b_1=24760.23 ± 0.0 ms/MB

a server=Jetty cacheContext=true cacheSessions=true
σ=45.0ms R^2=1.00 b_0=288.01 ± 0.1 ms b_1=24117.47 ± 0.0 ms/MB

Constants:

numberOfConnections	10
protocol	1.1
requestsPerConnection	1
socket	SSL
useSnowflake	false
network locality	local



Experiment 15

Variables:

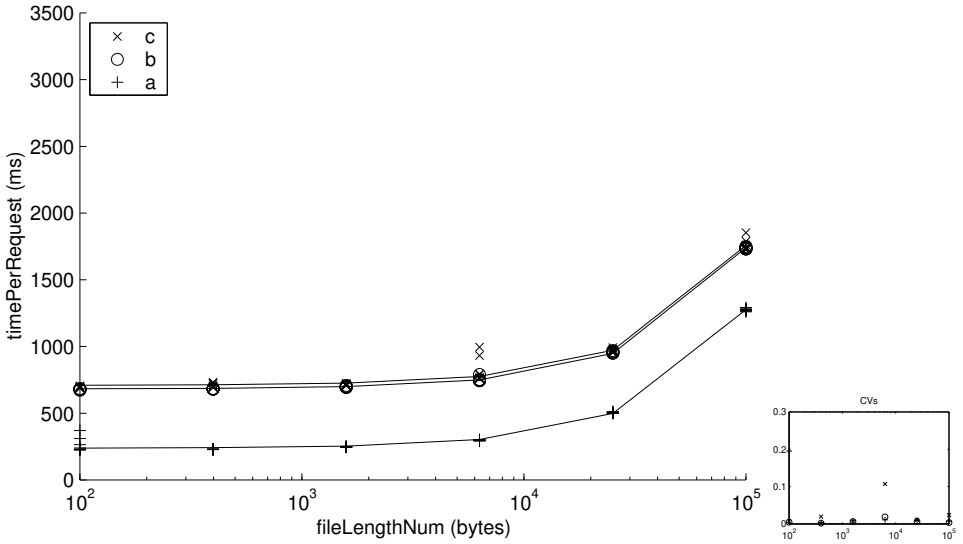
c server=apache cacheContext=false cacheSessions=false
σ=42.8ms R^2=0.99 b_0=709.13 ± 0.1 ms b_1=10993.92 ± 0.0 ms/MB

b server=apache cacheContext=true cacheSessions=false
σ= 8.5ms R^2=1.00 b_0=682.42 ± 0.0 ms b_1=11089.74 ± 0.0 ms/MB

a server=apache cacheContext=true cacheSessions=true
σ=22.9ms R^2=1.00 b_0=237.71 ± 0.0 ms b_1=10885.78 ± 0.0 ms/MB

Constants:

numberOfConnections	10
protocol	1.1
requestsPerConnection	1
socket	SSL
useSnowflake	false
network locality	remote



Experiment 16

Variables:

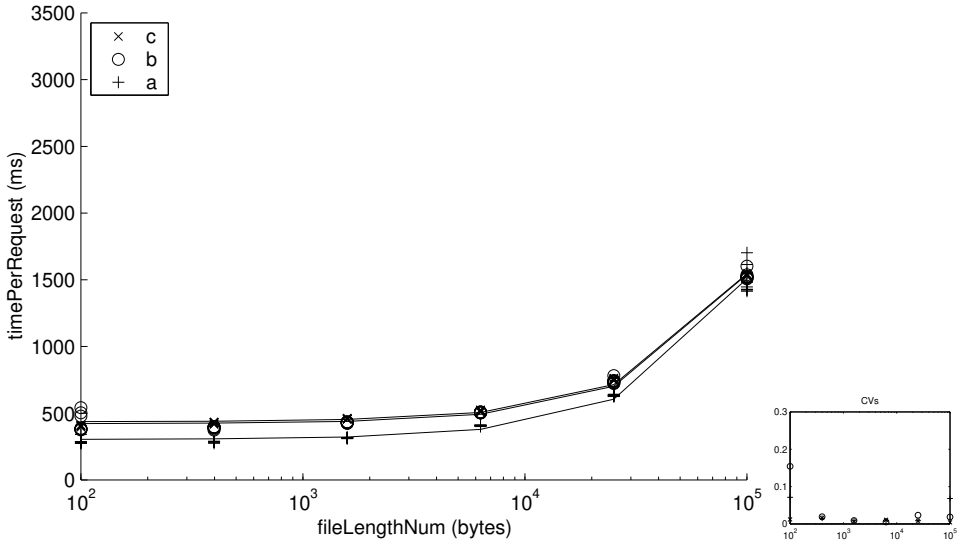
c server=Jetty cacheContext=false cacheSessions=false
σ=23.1ms R^2=1.00 b_0=436.01 ± 0.0 ms b_1=11624.15 ± 0.0 ms/MB

b server=Jetty cacheContext=true cacheSessions=false
σ=37.0ms R^2=0.99 b_0=421.37 ± 0.1 ms b_1=11723.18 ± 0.0 ms/MB

a server=Jetty cacheContext=true cacheSessions=true
σ=45.9ms R^2=0.99 b_0=303.33 ± 0.1 ms b_1=12624.89 ± 0.0 ms/MB

Constants:

numberOfConnections	10
protocol	1.1
requestsPerConnection	1
socket	SSL
useSnowflake	false
network locality	remote



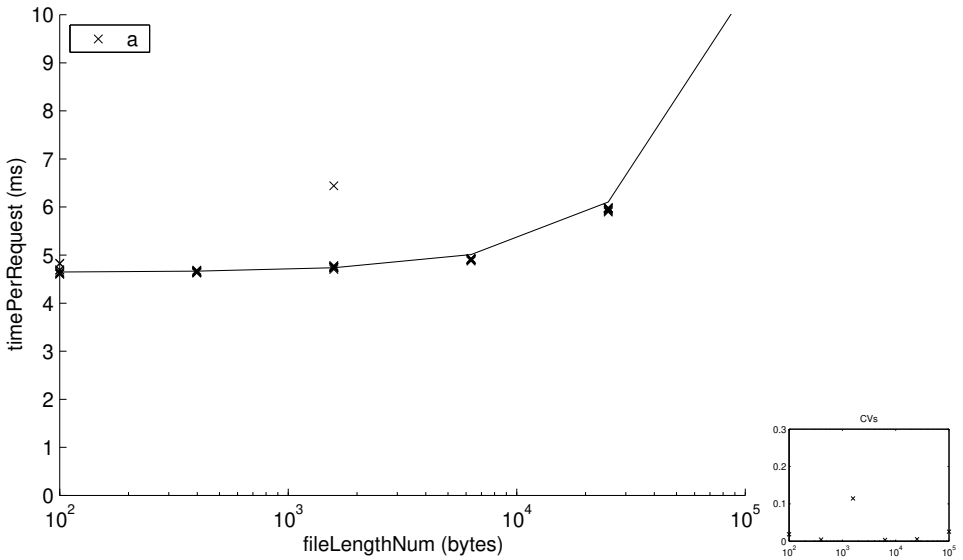
Experiment 17

Variables:

a server=apache
σ= 0.3ms R^2=0.98 b_0= 4.64 ± 0.0 ms b_1=61.05 ± 0.0 ms/MB

Constants:

client	fastget
numberOfConnections	200
protocol	1.1
server	apache
network locality	local



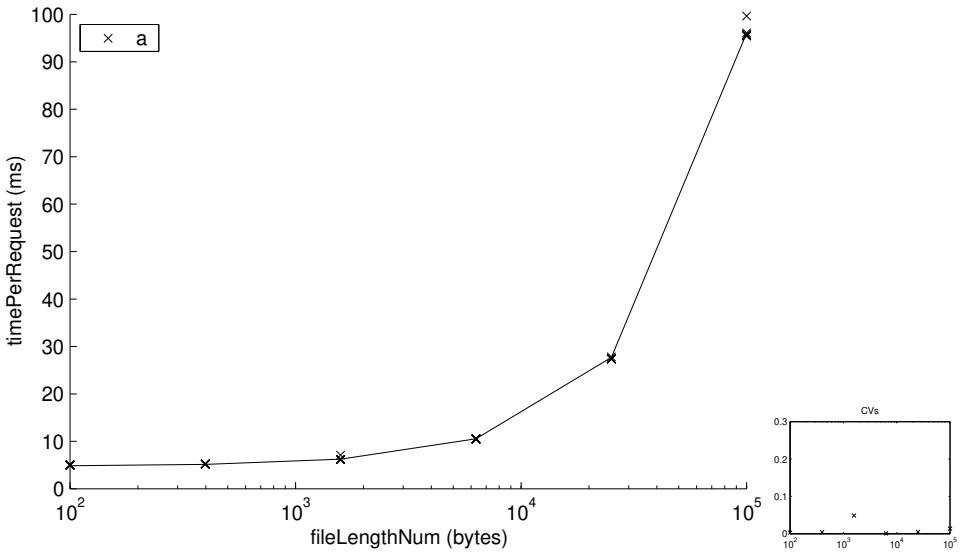
Experiment 18

Variables:

a server=apache
σ= 0.6ms R^2=1.00 b_0= 4.79 $\pm$ 0.0 ms b_1=956.89 $\pm$ 0.0 ms/MB

Constants:

client	fastget
numberOfConnections	200
protocol	1.1
server	apache
network locality	remote



Experiment 19

Variables:

d authenticateServer=true signFiles=true cacheSigns=false
σ= 9.6ms R^2=0.95 b_0=485.52 ± 0.0 ms b_1=1197.81 ± 0.0 ms/MB

b authenticateServer=false signFiles=true cacheSigns=false
σ= 7.0ms R^2=0.93 b_0=425.38 ± 0.0 ms b_1=764.17 ± 0.0 ms/MB

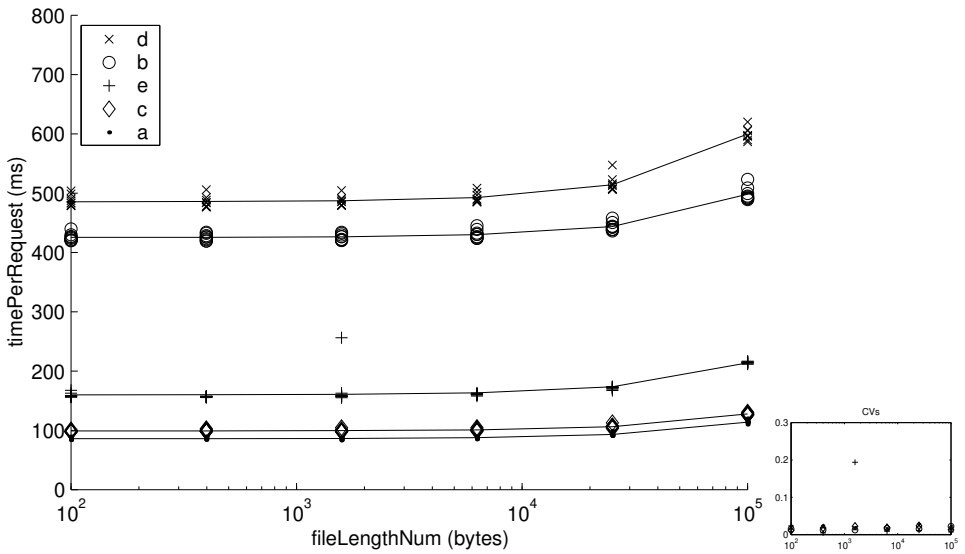
e authenticateServer=true signFiles=true cacheSigns=true
σ=13.6ms R^2=0.68 b_0=159.99 ± 0.0 ms b_1=563.97 ± 0.0 ms/MB

c authenticateServer=false signFiles=true cacheSigns=true
σ= 1.9ms R^2=0.97 b_0=99.13 ± 0.0 ms b_1=299.95 ± 0.0 ms/MB

a authenticateServer=false signFiles=false cacheSigns=false
σ= 2.2ms R^2=0.96 b_0=86.04 ± 0.0 ms b_1=293.03 ± 0.0 ms/MB

Constants:

identicalRequests	true
numberOfConnections	10
port	8041
protocol	1.0
requestsPerConnection	1
server	simple
socket	plain
useMacs	true
useSnowflake	true
network locality	local



Experiment 20

Variables:

d authenticateServer=true signFiles=true cacheSigns=false
σ=10.7ms R^2=0.97 b_0=481.71 ± 0.0 ms b_1=1757.37 ± 0.0 ms/MB

b authenticateServer=false signFiles=true cacheSigns=false
σ= 9.0ms R^2=0.97 b_0=426.82 ± 0.0 ms b_1=1397.21 ± 0.0 ms/MB

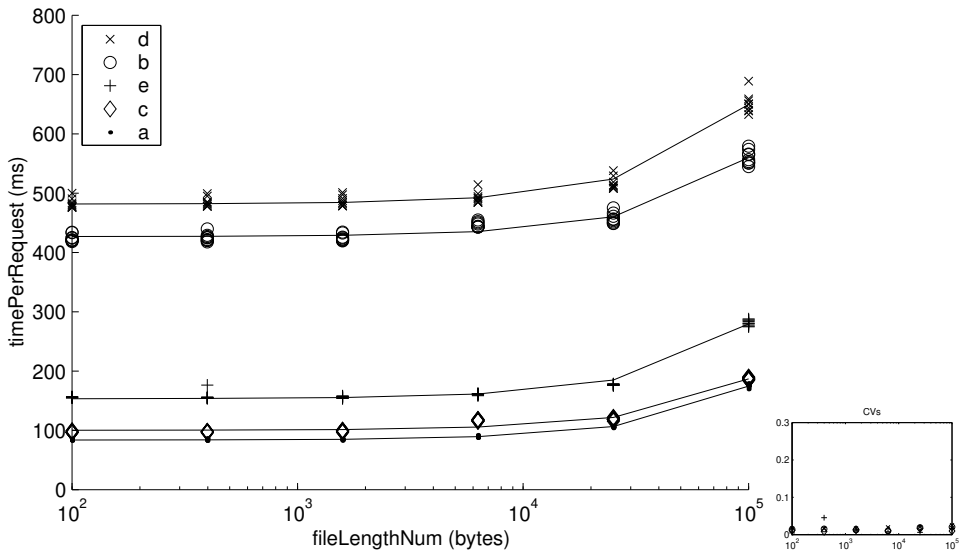
e authenticateServer=true signFiles=true cacheSigns=true
σ= 5.2ms R^2=0.99 b_0=153.42 ± 0.0 ms b_1=1321.35 ± 0.0 ms/MB

c authenticateServer=false signFiles=true cacheSigns=true
σ= 5.5ms R^2=0.97 b_0=100.11 ± 0.0 ms b_1=910.74 ± 0.0 ms/MB

a authenticateServer=false signFiles=false cacheSigns=false
σ= 1.9ms R^2=1.00 b_0=83.64 ± 0.0 ms b_1=955.37 ± 0.0 ms/MB

Constants:

identicalRequests	true
numberOfConnections	10
port	8041
protocol	1.0
requestsPerConnection	1
server	simple
socket	plain
useMacs	true
useSnowflake	true
network locality	remote



Experiment 23

Variables:

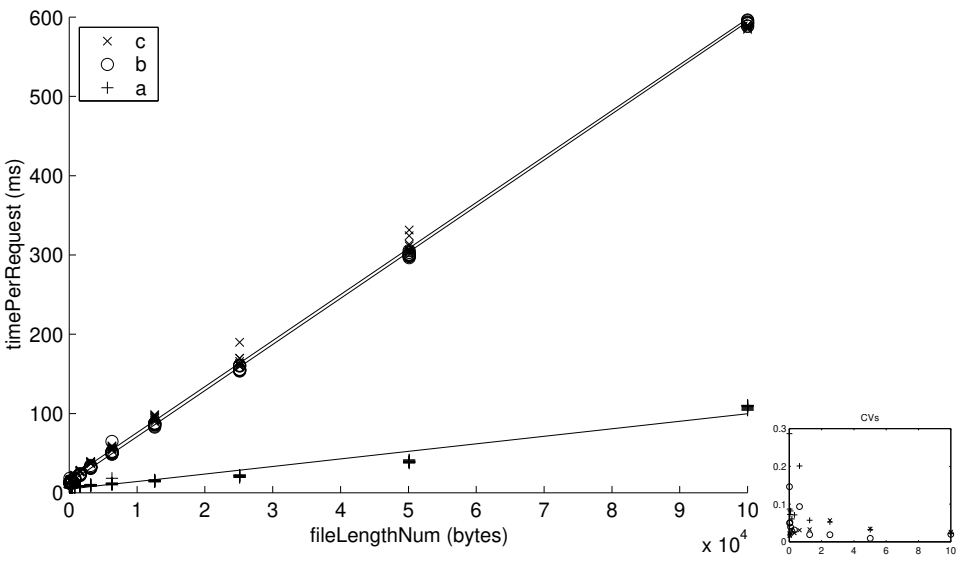
c registryService=TestRMIServer3
σ= 6.7ms R^2=1.00 b_0=17.90 ± 0.0 ms b_1=6083.68 ± 0.0 ms/MB

b registryService=TestRMIServer2
σ= 4.4ms R^2=1.00 b_0=12.67 ± 0.0 ms b_1=6097.16 ± 0.0 ms/MB

a registryService=TestRMIServer0
σ= 5.7ms R^2=0.96 b_0= 4.78 ± 0.0 ms b_1=994.51 ± 0.0 ms/MB

Constants:

experimentType	RMIExp
numberOfConnections	100
port	8143
requestsPerConnection	1
network locality	local



Experiment 24

Variables:

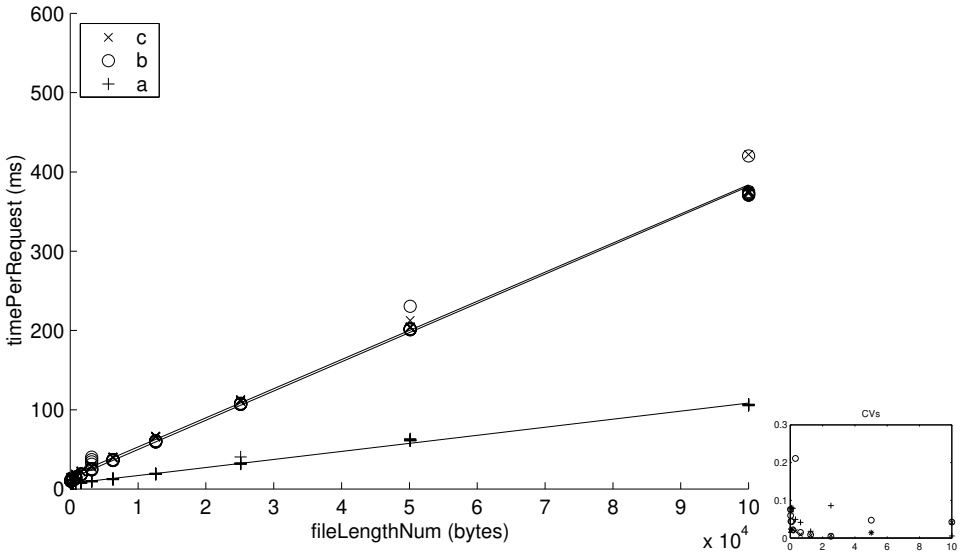
c registryService=TestRMIServer3
σ= 5.3ms R^2=1.00 b_0=16.44 ± 0.0 ms b_1=3852.20 ± 0.0 ms/MB

b registryService=TestRMIServer2
σ= 6.6ms R^2=1.00 b_0=12.98 ± 0.0 ms b_1=3867.08 ± 0.0 ms/MB

a registryService=TestRMIServer0
σ= 1.7ms R^2=1.00 b_0= 6.94 ± 0.0 ms b_1=1063.75 ± 0.0 ms/MB

Constants:

experimentType	RMIExp
numberOfConnections	100
port	8143
requestsPerConnection	1
network locality	remote



Experiment 25

Variables:

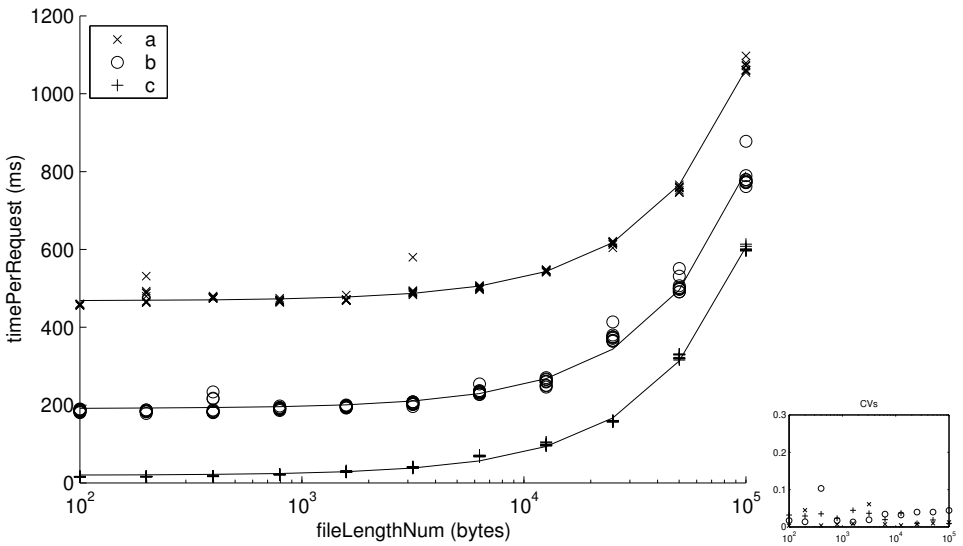
a clientCachesProofs=false serverCachesProofs=false
σ=14.4ms R^2=0.99 b_0=468.29 ± 0.0 ms b_1=6235.17 ± 0.0 ms/MB

b clientCachesProofs=true serverCachesProofs=false
σ=18.8ms R^2=0.99 b_0=191.27 ± 0.0 ms b_1=6377.39 ± 0.0 ms/MB

c clientCachesProofs=true serverCachesProofs=true
σ= 7.1ms R^2=1.00 b_0=19.77 ± 0.0 ms b_1=6149.06 ± 0.0 ms/MB

Constants:

experimentType	RMIExp
numberOfConnections	30
port	8143
registryService	TestRMIServer3
requestsPerConnection	1
network locality	local



Experiment 26

Variables:

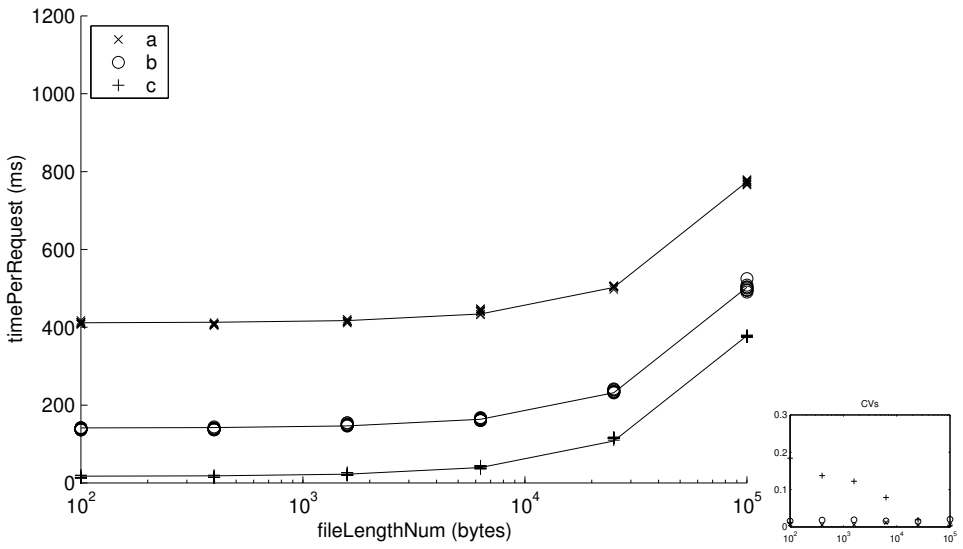
a clientCachesProofs=false serverCachesProofs=false
σ= 5.1ms R^2=1.00 b_0=411.43 ± 0.0 ms b_1=3798.24 ± 0.0 ms/MB

b clientCachesProofs=true serverCachesProofs=false
σ= 5.5ms R^2=1.00 b_0=140.97 ± 0.0 ms b_1=3796.35 ± 0.0 ms/MB

c clientCachesProofs=true serverCachesProofs=true
σ= 4.4ms R^2=1.00 b_0=16.94 ± 0.0 ms b_1=3790.53 ± 0.0 ms/MB

Constants:

experimentType	RMIExp
numberOfConnections	15
port	8143
registryService	TestRMIServer3
requestsPerConnection	1
network locality	remote



Bibliography

[Aba97] M. Abadi. Secrecy by typing in security protocols. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS 97)*, pages 611–638, 1997.

[Aba98] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[ABLP93] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[AEK96] Ahmed Amer and Amr El-Kadi. Beating bottlenecks in the design of distributed file systems. *;login: the USENIX Association Newsletter*, 21(6):14–23, December 1996.

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: the Spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[AISS98] A.D. Alexandrov, M. Ibel, K.E. Schauser, and C.J. Scheiman. Ufo: a personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[AN96] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[AT91] M. Abadi and M.R. Tuttle. A semantics for a logic of authentication. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 201–216, August 1991.

[Aur98] Tuomas Aura. On the structure of delegation networks. In *Proceedings of the Eleventh IEEE Computer Security Foundations Workshop*, pages 14–26, 1998.

[AWSBL99] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. *ACM Operating Systems Review*, 33(5):186–201, December 1999.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[BCE+95] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN: An extensible microkernel for application-specific operating system services. *ACM Operating Systems Review*, 29(1):74–77, January 1995.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[BGJ+92] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–31, 1992.

[BLNS82] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[BLNS86] A.D. Birrell, B.W. Lampson, R.M. Needham, and M.D. Schroeder. A global authentication service without global trust. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 223–230, 1986.

[BM91] David F. Belding and Kevin J. Mitchell. *Foundations of Analysis*. Prentice-Hall, 1991.

[BM97] Annette Bleeker and Lambert Meertens. A semantics for BAN logic. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[BM98] G. Banga and J.C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 1–12, 1998.

[BP88] B.N. Bershad and C.B. Pinkerton. Watchdogs — extending the unix file system. *Computing Systems*, 1(2):169–188, Spring 1988.

[BRS+85] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. Mach-1: an operating environment for large-scale multiprocessor applications. *IEEE Software*, 2(4):65–67, July 1985.

[BSP+95] Brian Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995. ACM Press.

[BVAD98] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS wide area security architecture. In *Proceedings of the Seventh USENIX Security Symposium*, pages 15–29, January 1998.

[CL98] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the Seventh USENIX Security Symposium*, pages 1–14, January 1998.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, pages 271–307, November 1994.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[dLPT+99] Eyal de Lara, Karin Petersen, Douglas B. Terry, Anthony LaMarca, Jim Thornton, Mike Salisbury, Paul Dourish, Keith Edwards, and John Lamping. Caching documents with active properties. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS)*, March 1999.

[DMST95] Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux, and William H. Tetzlaff. Evaluation of design alternatives for a cluster file system. In *Proceedings of the 1995 USENIX Technical Conference*, pages 35–46, January 1995.

[EFL+98] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Internet draft `draft-ietf-spki-cert-structure-05.txt` (expired), March 1998.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory, October 1999. Internet RFC 2693.

[Eli98] Jean-Emile Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, 1998.

[FHBH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication, June 1999. Internet RFC 2617.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.

[FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[FK98] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[Fou96] Free Software Foundation. Towards a new strategy of OS design, 1996. Available at: `http://www.gnu.ai.mit.edu/software/hurd/hurd.html`.

[FS96] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 91–105, October 1996.

[GJSJ91] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. *ACM Operating Systems Review*, 25(5):16–25, October 1991.

[GM98] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. *ACM Operating Systems Review*, pages 265–278, February 1998.

[Gol73] William Goldman. *The Princess Bride*. Ballantine, 1973.

[GPR+98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A global layer Unix for a network of workstations. *Software—Practice and Experience*, 28(9):929–961, July 1998.

[GS98] J. W. Gray III and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73–90, 1998.

[GWtL97] Andrew S. Grimshaw, Wm. A. Wulf, and the Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[HC96] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[HEV+98] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software—Practice and Experience*, 28(9):901–928, July 1998.

[How99] Jon Howell. Straightforward Java persistence through checkpointing. In Ron Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems*, pages 322–334. Morgan Kaufmann, 1999.

[HT96] N. Heintze and J.D. Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22(1):16–30, January 1996.

[HvdM99] Joseph Y. Halpern and Ronald van der Meyden. A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 111–122, 1999.

[Joh93] M. St. Johns. Identification protocol, February 1993. Internet RFC 1413.

[Jon93] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[Lam86] Butler W. Lampson. Designing a global name service. In *Proceedings of the Fourth ACM Symposium on Principles of Distributed Computing*, pages 1–10, Minaki, Ontario, August 1986. ACM Press.

[Lan81] C.E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[LN98] I. Lehti and P. Nikander. Certifying trust. In *Proceedings of the First International Workshop on Practice and Theory in Public Key Cryptography*, pages 83–98, February 1998.

[Loe92] K. Loepere. Mach 3 kernel principles. Technical Report MK67, Open Software Foundation and Carnegie Mellon University, 1992.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software—Concepts and Tools*, 17(3):93–102, 1996.

[LRD95] Anders Lindström, John Rosenberg, and Alan Dearle. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 66–71, May 1995.

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[LSM+98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the Twenty-First National Information Systems Security Conference*, Arlington, Virginia, October 1998.

[Mas97] F. Massacci. Reasoning about security: a logic and a decision method for role-based access control. In *Proceedings of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning*, pages 421–435, 1997.

[MGH+94] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *Proceedings of COMPCON '94*, pages 122–131, 1994.

[Mio98a] A. Mione. A look at PKI design efforts. *Digital Systems Report*, 20(3):1–6, 1998.

[Mio98b] A. Mione. A look at some more PKI design efforts. *Digital Systems Report*, 20(4):15–21, 1998.

[MKD+00] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, and Arthur van Hoff. Delta encoding in HTTP. Internet draft `draft-mogul-http-delta-03.txt` (work in progress), March 2000.

[MKKW99] D. Mazières, M. Kaminsky, M.F. Kaashoek, and E. Witchel. Separating key management from file system security. *ACM Operating Systems Review*, 33(5):124–139, December 1999.

[Mor98] Alexander Morcos. A Java implementation of Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Technology, May 1998.

[Mos98] Karl Moss. *Java Servlets*. Computing McGraw-Hill, July 1998.

[MSC+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[MWSK94] K. Murray, T. Wilkinson, T. Stiemerling, and P. Kelly. Angel: resource unification in a 64-bit microkernel. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 106–115, January 1994.

[Neu92] B. Clifford Neuman. The Prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, Fall 1992.

[Neu93] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[OD83] D.C. Oppen and Y.K. Dalal. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, 1983.

[Pik00] Rob Pike. Personal communication, March 2000. Bell Laboratories.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[Riv97] R. Rivest. S-Expressions. Internet draft `draft-rivest-sexp-00.txt` (expired), May 1997.

[RLML86] J. Rees, P.H. Levine, N. Mishkin, and P.J. Leach. An extensible I/O system. In *USENIX Association Summer Conference Proceedings, Atlanta 1986*, pages 114–125, 1986.

[Sal78] J.H. Saltzer. Naming and binding of objects. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pages 99–208. Springer-Verlag, 1978.

[SBN84] M.D. Schroeder, A.D. Birrell, and R.M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 213–227. USENIX Association, October 1996.

[Sol85] Karen Rosin Sollins. *Distributed name management*. PhD thesis, Massachusetts Institute of Technology, February 1985.

[Sol88] Karen R. Sollins. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 156–163, 1988.

[SP95] W. Richard Stevens and Jan-Simon Pendry. Portals in 4.4BSD. In *Proceedings of the 1995 USENIX Technical Conference*, pages 1–10, January 1995.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SSL+99] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.

[Syv93] Paul F. Syverson. Adding time to a logic of authentication. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 164–173, November 1993.

[Tan95] Andrew S. Tanenbaum. A comparison of three microkernels. *Journal of Supercomputing*, 9(1):7–22, March 1995.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[UKS95] Ronald C. Unrau, Orran Krieger, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, pages 105–134, March 1995.

[VAB91] V. Varadharajan, P. Allen, and S. Black. An analysis of the proxy problem in distributed systems. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 255–275, 1991.

[VDAA99] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active names: Flexible location and transport of wide-area resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999.

[VK83] V.L. Voydock and S.T. Kent. Security mechanisms in high-level network protocols. *Computing Surveys*, 15(2):135–171, 1983.

[vSHBT98] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the Globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[vSHT99] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: a wide area distributed system. *IEEE Concurrency*, 7(1):70–78, January 1999.

[WABL94] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[Wal98] Jim Waldo. Jini architecture overview. Sun Microsystems white paper, 1998. Available at: `http://www.java.sun.com/products/jini/whitepapers/architectureoverview.pdf`.

[Wat81] R. W. Watson. Identifiers (naming) in distributed systems. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed systems—architecture and implementation: an advanced course*, chapter 9, pages 191–210. Springer-Verlag, 1981.

[Wel94] Brent Welch. A comparison of three distributed file system architectures: Vnode, Sprite, and Plan 9. *Computing Systems*, 7(2):175–199, Spring 1994.

[WF98] D.S. Wallach and E.W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, 1998.

[WO86] Brent Welch and John Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the Sixth International Conference on Distributed Computer Systems*, pages 184–189, 1986.

[YKS+98] Tatu Ylonen, Tero Kivinen, Markku-Juhani O. Saarinen, Timo J. Rinne, and Sami Lehtinen. Ssh protocol architecture. Protocol specification, August 1998. Available at: `http://www.ssh.fi/drafts/`.

[Ylo96] Tatu Ylonen. The SSH (secure shell) remote login protocol. Internet draft `draft-ylonen-ssh-protocol-00.txt` (expired), May 1996.