Access Control for the Web via Proof-carrying Authorization

Ljudevit Bauer

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 2003

Abstract

After a short period of being not much more than a curiosity, the World-Wide Web quickly became an important medium for discussion, commerce, and business. Instead of holding just information that the entire world could see, web pages also became used to access email, financial records, and other personal or proprietary data that was meant to be viewed only by particular individuals or groups. This made it necessary to design mechanisms that would restrict access to web pages. Unfortunately, most current mechanisms are lacking in generality and flexibility—they interoperate poorly and can express only a limited number of security policies.

We view access control on the web as a general distributed authorization problem and develop a solution by adapting the techniques of proof-carrying authorization, a framework for defining security logics based on higher-order logic.

In this dissertation we present a particular logic for modeling access-control scenarios that occur on the web. We give this application-specific logic a semantics in higher-order logic, thus ensuring its soundness, and use it to implement a system that regulates access to web pages. Our system uncouples authorization from authentication, allowing for better interoperation across administrative domains and more expressive security policies. Our implementation consists of a web server module and a local web proxy. The server allows access to pages only if the web browser can demonstrate that it is authorized to view them. The browser's local proxy accomplishes this by mechanically constructing a proof of a challenge sent to it by the server. Our system supports arbitrarily complex delegation, and we implement a framework that lets the web browser locate and use pieces of the security policy that have been distributed across arbitrary hosts. Our system was built for controlling access to web pages, but could relatively easily be extended to encompass access control for other applications as well.

Acknowledgments

Many have had an impact on this dissertation by offering their support, advice, and wisdom. I am grateful to all of them, and cognizant that I could not have done it alone.

I would like to thank my collaborators in this work: Andrew Appel, Ed Felten, and Michael Schneider. Their contributions are directly responsible for a large part of this project.

I am especially indebted to Andrew Appel. He has been a great advisor, and I feel fortunate to have been his student. His door was always open, and he was always a dependable source of ideas, advice, and encouragement.

I have had interesting discussions with many researchers about logic-based distributed authorization. A few conversations with Mike Reiter and Carl Ellison about the present and future of such technologies were particularly inspiring. I would also like to thank Dave Walker, who repeatedly allowed himself to be badgered into answering questions about logic.

The research described in this dissertation was supported by NSF grants CCR-9870316 and CCR-0208601 and by DARPA grant F30602-99-1-0519.

Contents

List of Figures

Chapter 1

Introduction

1.1 Access Control on the Web

After a short period of being not much more than a curiosity, the World-Wide Web quickly became an important medium for discussion, commerce, and business. Instead of holding just information that the entire world could see, web pages also became used to access email, financial records, and other personal or proprietary data that was meant to be viewed only by particular individuals or groups.

This made it necessary to design mechanisms that would restrict access to web pages. The most widely used mechanism is for the user to be prompted for a user name and password before he is allowed to see the content of a page [27]. A web administrator decides that a certain page will be visible only if a user enters a correct user name/password pair that resides in an appropriate database file on the web server. A successful response will often result in the client's browser being given a cookie; on later visits to the same or related web pages, the cookie will be accepted as proof of the fact that the user has already demonstrated his right to see those pages and he won't be challenged to prove it again [49]. An organization such as a university may require that all people wishing to see a restricted web page first visit a centralized login page which handles authentication for all of the organization's web sites. The cookie placed on the client's browser then contains information which any of the organization's web servers can use to verify that it was legitimately issued by the organization's authentication service. In cases such as this, the functions of authentication (verifying an identity) and authorization (granting access) are separated into two distinct processes.

More modern methods of controlling access to web pages separate these functions even further, not as an optimization, but as a basic element of their design. Increasingly in use are systems in which a certificate, such as a Kerberos ticket [34, 44] or an X.509 certificate [32], is obtained by a user through out-of-band means; a web browser and a web server are augmented so that the web browser can pass the certificate to the web server and the web server can use the certificate to authorize the user to access a certain page. The advantage of these mechanisms is that, in addition to providing more secure implementations of protocols similar to basic web authentication, they make it possible for different services to authorize access based on the same token. An organization can now provide a single point of authentication for access to web pages, file systems, and Unix servers.

Though growing increasingly common, most notably due to the use of Kerberos in new versions of the Windows operating system, these systems have not yet gained wide acceptance. This is partly because they don't adequately deal with all the requirements for authorization on the Web, so their undeniable advantages may not be sufficient to justify their cost.

One of the chief weaknesses of these systems is that they are not good at providing interoperability between administrative domains, especially when the domains use different security policies or authorization systems. Having a centralized authentication server that issues each user a certificate works well when a large number of web servers are willing to trust that particular authentication server (at a university, for example), but not when such trust is absent (such as between two universities). There have been attempts to build systems that cross this administrative divide [24] but the problem still awaits practical solution.

We have built a system that addresses this issue. Our system even further uncouples authorization from authentication, allowing for superior interoperation across administrative domains and more expressive security policies. Our implementation consists of a web server module and a local web proxy. The server allows access to pages only if the web browser can demonstrate that it is authorized to view them. The browser's local proxy accomplishes this by mechanically constructing a proof of a challenge sent to it by the server. Our system supports arbitrarily complex delegation, and we implement a framework that lets the web browser locate and use pieces of the security policy (e.g., delegation statements) that have been distributed across arbitrary hosts. Our system was built for controlling access to web pages, but could relatively easily be extended to encompass access control for other applications (e.g., file systems) as well.

1.2 Goals and Design

In designing our system for access control of web pages we had several criteria that we wanted to address:

- interoperability and expressivity;
- convenience to the user;

- ease of implementation and integration with web servers and web browsers;
- efficiency;
- applicability to spheres other than web access control.

1.2.1 Interoperability and Expressivity

Even the most flexible of current systems for web access control are limited in their ability to interoperate across administrative boundaries, especially when the domains use different security policies or authorization systems. One of the main reasons for this is that though current systems attempt to separate the functions of authorization and authentication, they overwhelmingly continue to express their security policy—the definition of which entities are authorized to view a certain web page—in terms of the identities of the users. Though the web server often isn't the entity that authenticates a user's identity, basing the security policy on identity makes it very difficult to provide access to users who can't be authenticated by a server in the same administrative domain.

The way we choose to resolve this issue is by making the security policy completely general—the right to access to a page can be described by an arbitrary predicate. This predicate is likely to, but need not, be linked to a verification of identity—it could be that a particular security policy grants access only to people who are able to present the proof of Fermat's last theorem. Since the facts needed to satisfy this arbitrary authorization predicate are likely to include more than just a verification of identity, in our access-control system we replace authentication servers with more general fact servers. In this scenario the problem of deciding whether a particular client should be granted access to a particular web page becomes a general distributed-authentication problem, which we solve by adapting previously developed techniques from that field.

Distributed authorization systems [14, 21, 23, 32] provide a way for implementing and using complex security policies that are distributed across multiple hosts. The methods for distributing and assembling pieces of the security policy can be described using logics [1, 30], and distributed authorization systems have been built by first designing an appropriate logic and then implementing the system around it [2, 3, 10]. Appel and Felten recently introduced proof-carrying authorization (PCA), the most general of the logic-based frameworks [8].

Appel and Felten argue that the underlying reasoning framework of a distributed authorization system should be a general logic like higher-order logic [17]. In a traditional scenario the use of an undecidable logic like higher-order logic would be infeasible, since a server might not be able to decide whether a set of credentials implies that access should be granted. Appel and Felten reason, however, that the authorization process associated with any particular client's request—that is, the facts or credentials that support the request and the way they can be assembled into a proof of access—can be described using a subset of higher-order logic that corresponds to a simple and decidable application-specific logic. The client can use this application-specific logic to construct a proof of the fact that it should be allowed access. This proof, along with the definition of the application-specific logic in terms of the higher-order logic, is sent to the server. To determine whether the client should be granted access, the server merely has to verify that the proof is valid.

The server, using only the common underlying logic, can check proofs from all clients, regardless of the method they used to generate the proof or the application-specific logic they chose. This technique perfectly satisfies our goal of interoperability—as long as a server bases its access-control policy on the underlying logic, interoperation with systems in different administrative hierarchies need be no more difficult than interoperation with local ones.

1.2.2 Ease of Use

One of the major criticisms of many existing security mechanisms is that they place too great a burden on the user, who typically is not aware of and does not care to learn about all the security implications of the operations he wishes to perform. Users often take advantage of only a very few of the many capabilities such systems offer and are often unaware of the systems' fundamental principles. As a consequence, many systems provide a far weaker level of security than was intended.

One of the simplest ways of avoiding some of these pitfalls is to make the security mechanism as invisible to the user as possible. A more powerful and expressive access-control mechanism is generally likely to require more user interaction than a simple one, but in our system we succeed in confining the complexity to the setup phase. After our system has been installed and configured for a particular user and application (for example, controlling access to a university's course web pages), using it to successfully access protected web pages is completely transparent to the user.

1.2.3 Convenience of Implementation

An important goal for a web access-control system that aspires to be practical is that it be implementable without major modification of the existing infrastructure—that is, web browsers and web servers. Our access-control system involves three types of players: web browsers, web servers, and fact servers (which issue tokens that can certify not only successful authentication—as do ordinary authentication servers—but also any other type of fact that they store).

We enable the web browser to understand our authorization protocol by implementing a local web proxy. The proxy intercepts a browser's request for a protected page and then executes the authorization protocol and generates the proof needed for accessing the page; the web browser sees only the result—either the page that the user attempted to access or an appropriate failure message. Each user has a unique cryptographic key held by the proxy. Users' identities are established by certificates stored on fact servers that bind names to keys. The use of keys makes it unnecessary to prompt the user for a password, making the authorization process quicker and more transparent.

For tighter integration with the browser and for better performance, the proxy could be packaged as a browser plugin. This would make it less portable, however, as a different plugin would have to be written for each type of browser; we did not feel this was within the scope of our prototype implementation.

The web server part of our system is built around an unmodified web server. The web server is extended through the use of a servlet which intercepts and handles all PCA-related requests. The two basic tasks that take place on the server's side during an authorization transaction are generating the proposition that needs to be proved and verifying that the proof provided by the client is correct. Each is performed by a separate component, the proposition generator and the checker, respectively.

Fact servers hold the facts a client must gather before it can construct a proof. Each fact is a signed statement in the PCA logic. We implement an off-line utility for signing statements, which lets us use a standard web server as a fact server. The fact server can also restrict access to the facts it publishes with a servlet, in the same manner as the web server.

1.2.4 Efficiency

The access-control process is transparent to a user. To be practical, it must also be efficient. Assembling the facts necessary to construct a proof may involve several transactions over the network. The actual construction of the proof, the cryptographic operations done during the protocol, and proof checking are all potential performance bottlenecks.

Though our system is a prototype and not of production quality, it demonstrates that this approach can be made to perform well enough to be acceptable in practice. Heavy use of caching limits the need to fetch multiple facts over the network and speculative proving makes it possible to shorten the conversation between the web proxy and the servlet.

1.2.5 Generality

The best current web authorization mechanisms have the characteristic that they are not limited to providing access control for web pages; indeed, their strength is that they provide a unified method that also regulates access to other resources, such as file systems. Our system, while implemented specifically for access control on the Web, can also be extended in this manner. The idea of proof-carrying authorization is not specific to web access control, and the mechanisms we develop, while implemented in a web proxy and a servlet, could easily be modified to provide access control for other resources.

1.3 Related Work

Access control for the web can be thought of as a special instance of distributed authorization. This section reviews the most influential efforts to design comprehensive solutions to the problem.

1.3.1 X.509

One of the notable forerunners and the building block of many distributed authorization systems is the X.509 standard [32]. X.509 defines certificates that bind public keys to X.500 Distinguished Names [52]. X.500 was designed to be a global standard for keeping track of named entities in a hierarchical fashion. An X.509 certificate was meant to indicate who was allowed to modify the portion of the X.500 database described by a particular Distinguished Name. X.500 postulated a globally unique naming scheme that made widespread adoption of X.500 infeasible, rendering the intended function of X.509 certificates useless. X.509 certificates later started to be used, and are currently used, as a mechanism for associating a public key with a person's name. The authenticity and integrity of an X.509 certificate is guaranteed by a digital signature. The signing key's own X.509 certificate and a certificate chain leading to a primary root certification authority are attached to the signed certificate. X.509 thus provides a public-key infrastructure useful for many applications [20, 46, 47]. In addition, X.509 certificates can be extended to describe more than just keys and names, making them a convenient vehicle for carrying security relevant data [25].

1.3.2 SPKI/SDSI

SPKI/SDSI (now known as SPKI 2.0) [23] is a digital certificate scheme that focuses on authorization rather than just authentication. It represents the merging of two efforts: SPKI [22], in particular, its method of binding authorization privileges to keys; and SDSI [48], which contributed its mechanism for linking keys and names.

SPKI/SDSI certificates come in three forms: name-to-key, authorization-to-name, and authorization-to-key bindings. Bindings of names to keys are meant to be interpreted locally [1, 30]. If the principal *Alice* binds the name *Bob* to a particular key, this binding is considered to be true only within *Alice*'s name space. An outsider wishing to refer to *Bob* can do it by specifying that he is talking about *Alice*'s *Bob*, in other words, the *Bob* whose name-to-key binding is specified by *Alice*. *Alice* herself is identified either with her public key or with a chain of names rooted at a known key. In this manner, any named entity can be globally identified by fully qualifying the name space in which it is defined.

SPKI/SDSI specifies that certificates, in addition to mapping names to keys, can also be used to describe bindings of privileges, or authorizations, to keys or names. In such cases, a certificate is interpreted to mean that the given key has the right to exercise the named privilege. Privileges are represented as strings and can be combined using a few simple combinators. Each certificate has a boolean flag that describes whether the key may delegate the privilege. A delegated privilege can be further delegated by the recipient.

An interesting feature of SPKI/SDSI is its sensible treatment of certificate revocation lists (CRLs). A certificate that supports validation must identify the keys that issue CRLs and the locations where they are published, and CRLs must carry non-intersecting validity dates. This makes it possible to accept certificates only in the presence of a valid CRL, and it ensures that only a single CRL will be valid at a given time.

Each certificate is represented as a 4- or 5-tuple. The authorization process involves verifying the validity of certificates, translating the uses of names to a canonical form [19, 21], and computing the intersection of the privileges described in authorization tuples. This yields a single authorization tuple which describes whether or not the named principal may access a particular resource.

SPKI/SDSI has been used to develop an access-control mechanism for protecting web pages [42, 18]. In this system, the web server presents a browser with the access-control list (ACL) protecting the page that the client is trying to access. The client maintains a local cache of SPKI/SDSI certificates, which it uses to generate an appropriate certificate chain. The client then sends the certificate chain to the server, which verifies it before allowing access to the web page.

1.3.3 Taos

The Taos operating system [50] was a test bed for a distributed authentication system [35, 53] based on a formal logic (ABLP logic) [2]. The distributed system consisted of a number of nodes connected via a network, each running the Taos OS. To access a resource, a principal first has to authenticate himself to it, and then demonstrate that he is authorized to access it. A local agent running on each node is in charge of managing credentials and communicating with other nodes.

The Taos authentication system is based on a modal logic with a very rich notion of principals: they can be users, machines, channels, groups, roles, principals quoting other principals, etc. A principal *says* a formula to indicate that he supports the statement or assertion that it represents. The rich algebra of principals is used to specify precisely the origin of a request or belief (for example, instead of stating that the user Bob *says* a formula, the logic is able to express the much more precise statement that a particular machine running a particular operating system speaking on Bob's behalf *says* the formula). In addition to standard axioms present in modal logic, the ABLP logic contains axioms that capture ideas particular to distributed authentication (the delegation axiom, for example, states that if principal *A* *says* that *B* speaks on its behalf then this becomes a global truth that can be used in future inferences). The principal who controls a resource may delegate the right to access it to whomever he chooses. A principal *A* is allowed to access a resource if there exists a delegation chain from the controlling principal to *A*.

The ABLP logic is too powerful to be decidable, so Taos uses a subset with a much restricted notion of principals. Each node running the Taos OS exports a simple API that is sufficient for one node to convince another that some principal A *says* the formula F. Resources are protected by access-control lists. Each element of an ACL is interpreted to mean that the named principal controls the resource. A request to access a resource is successful if a chain of delegations allows the requester to speak on behalf of the principal controlling the resource.

1.3.4 PolicyMaker

PolicyMaker [14] represents another attempt to design a universal system for expressing and verifying security policies. It deals specifically with distributed authorization in decentralized environments, which it dubs “trust management.”

PolicyMaker represents policies and credentials in a common language, and uses a general-purpose compliance checker to verify that a particular policy and set of credentials meet a given request. Credentials are interpreted locally; if one of them is a delegation statement, for example, it can be used to transfer authority only if the local policy has explicitly indicated that such delegation statements should be trusted. Each credential is a program written in a general-purpose programming language. The compliance checker labels each credential with a string representing the identity of its source. A policy is a credential labeled with the keyword “POLICY”. A principal’s privileges are encoded as strings, and the semantics of the encoding are provided locally by each credential. Credentials and policies take as input the currently known privileges (each labeled with a string representing its source), and output the set of privileges (again, labeled) that can be inferred from them. To verify whether a resource R may be accessed, the compliance checker runs the various credentials; if the policy, run on the resulting set of privileges, outputs the string *R*, access to the resource is granted.

PolicyMaker's compliance checker has a standardized algorithm to decide in which order and how many times to execute each credential. To ensure that the algorithm always terminates and produces the correct answer, the scope of credentials and policies had to be restricted: they are required to run in polynomial time and to be monotonic. The monotonicity requirement asserts that if an input I causes a credential to output the action string *R*, then the credential will also output the string *R* when given as input any superset of I.

KeyNote [13, 12] specializes the PolicyMaker system to authorization in public-key infrastructures. The language for specifying credentials is simpler and human-readable, and credential labels now represent the public keys of their issuers. The naming of privileges is again left to the particular application of the system.

REFEREE [16] applies the ideas of PolicyMaker and KeyNote to the web. As before, policies and credentials are programs, this time written in a language adapted to the web. To determine whether an action should be allowed, typically, whether it's OK for a browser to display a web page, the system will execute the policy associated with that action. A policy may consult other credentials or policies; this is done by dynamically downloading the appropriate program and then allowing it to execute.

1.3.5 Others Efforts

Although not as well publicized, several other efforts are also worth mentioning. QCM [29] is a public key infrastructure that incorporates a language for specifying policies. Its mechanism for verifying whether a request should be granted includes the facility for automatically retrieving needed credentials.

Delegation Logic (DL) [37] is a logic-based language for describing policies, credentials, and requests. A tractable, monotonic subset, D1LP [38, 39], has been implemented as a translation to ordinary logical programs.

The RT framework [40] is a family of languages that adds to the ideas of systems such as DL and SPKI/SDSI the notion of role-based access control [28, 41], providing access control based on roles and attributes in addition to identity. Principals can assert that other principals have specific attributes, and one type of attribute can be used to reason about another, eventually leading to an access-control decision. RT thus resolves the most obvious deficiency of SPKI/SDSI while remaining tractable in the style of D1LP.

1.4 Proof-carrying Authorization

Most distributed authorization systems try to provide support for notions such as the ability to delegate privileges and treat groups as principals. While they strive to be as expressive as possible—that is, to be able to represent as many security policies as possible—they are constrained by how they choose to implement these ideas. The SPKI/SDSI notion of delegation, for example, includes a boolean flag that describes whether the delegated privilege may be redelegated by the recipient. PolicyMaker, on the other hand, requires any redelegation to be explicitly approved by the security policy. Each choice may be the best one for a particular situation, but no one particular choice can be the ideal one for *all* situations.

Proof-carrying authorization follows a different approach to providing generality. Unlike other systems, in which axioms that define ideas like delegation are part of the logic which describes the system, PCA is based on a standard higher-order logic [17]. The only nonstandard axioms added to the logic are rules for decoding digital signatures.

Higher-order logic is undecidable—there is no algorithm which will always be able to prove the truth of every true statement—which raises the question: how can such a logic can be used in an authorization system? A server is typically presented with a list of credentials and has to decide whether they are sufficient for access to be granted. If the logic that models this is undecidable, the server might not be able to come to the correct conclusion.

PCA solves this problem by making it the client's responsibility to prove that access should be granted. All the server needs to do is verify that the client's proof is valid, which can be done efficiently even if the proof is expressed in an undecidable logic. Transferring the burden of proof from the server to the client ensures that the server's task is tractable, but doesn't explain how the client is able to construct a proof. What makes the client's task possible is that any particular client doesn't need the full expressivity of the undecidable logic. Instead, a particular client is probably content to construct proofs in some decidable subset of higher-order logic—an application-specific subset that corresponds to a particular notion of delegation, a particular way of defining groups or roles, etc. Each of the operators in this subset can be given a definition in higher-order logic, and each of the inference rules about these operators can be defined as a lemma. When constructing a proof using these operators and inference rules the client doesn't need to pay any particular attention to the fact that they are merely placeholders for higher-order logic terms—he can manipulate them as if they were a regular application-specific logic, for example, the logic that models SPKI/SDSI. The server verifying the proof, on the other hand, doesn't care which particular application-specific logic the client uses. As long as each operator and rule has a definition in higher-order logic, the server sees it as just another higher-order logic proof, which it knows how to verify.

This approach gives PCA great flexibility. In traditional systems we have to prove metatheorems about the logic before we can trust the system to perform correctly. In PCA much of this security comes from embedding the application-specific logic in a framework that is known to be sound. Since they all use the same underlying framework, it is not only easy to extend a particular application-specific logic, but it is also possible to make different application-specific logics interact.

1.5 Overview

This dissertation describes how proof-carrying authorization can be used to build a practical access-control system for the Web, and presents its design and implementation. In the process it defines the minimal extensions to a standard higher-order logic necessary to support such a system.

The first step in building a PCA system is to identify a class of access-control scenarios and develop an application-specific logic for reasoning about it (Chapter 2). The application-specific logic then needs to be encoded and given semantics in the PCA higher-order logic (Chapter 3). After we have settled on a particular logic, we must build the infrastructure that allows web servers and web browsers to use the logic to control access to web pages. This infrastructure comprises communication protocols, mechanisms for generating and verifying the validity of proofs, a scheme for distributing and collecting facts across a network, and a method for converting access requests to proof goals. Chapter 4 describes the prototype system that we have built, which addresses these issues. (Much of the implementation work was described in a prior conference paper [11].)

While our original application-specific logic is simple and effective, it presents a relatively low-level interface to the authorization process. Chapter 5 describes how the logic can be extended to encompass several more abstract concepts and present a more refined interface. Finally, Chapter 6 reviews the contributions of and discusses some of the research challenges raised by this work.

Much of this work was done in collaboration with others. The application-specific logic and its semantics were developed in cooperation with Andrew W. Appel and Edward W. Felten, and the prototype system was implemented jointly with Michael A. Schneider and Edward W. Felten.

Chapter 2

A Logic for Access Control

A PCA system, like any distributed authorization system, consists of a client that wants to access a resource, a server that controls this resource, and a number of other hosts that hold pieces of the security policy that protects the resource. The security policy is the set of facts that is used to demonstrate to the server that the client is allowed access to the resource.

The first step in building a PCA system is designing a logic that models these facts and the procedure used to assemble them into a proof. In contrast with other distributed authorization systems, in which the logic is a compromise driven by the need to be as comprehensive as possible, in PCA one can design an application-specific logic that is precisely tailored to a given scenario. This chapter introduces a simple logic designed for access control on the web. The semantics of this application-specific logic, i.e., its encoding in the PCA higher-order logic, is described in Chapter 3. Our purpose in presenting this particular logic and its syntax is to explain the principles of PCA; the logic itself is too simple to describe real scenarios. Chapter 5 discusses some extensions to the logic aimed both at providing a higher-level interface to the security policy architect and making the logic powerful enough to solve realistic problems.

Figure 2.1 shows a scenario that is typical of the types of access-control problems we would like to solve; it will be used as a running example throughout the rest of this dissertation. In this scenario we have three entities: Alice, the client; Bob, the server; and the Registrar, a fact server that holds a piece of the security policy.

Bob is a professor who teaches the class CS101. He recently gave his students a midterm, and wants to publish statistics about the results on the web. He wants these statistics to be accessible only to the students taking his class, and only after 8 P.M. on a particular day (a tardy student is still taking the midterm). Furthermore, he doesn't want to be troubled to keep an up-to-date list of students in his high-attrition class. Conveniently for him, the university where he teaches has a Registrar that keeps track of which classes students are taking. Finally, Alice is a student in Bob's class who wants to access the midterm results page. The goal in this scenario is for Alice to prove to Bob that he should let her see the web page.

This scenario captures most of the ideas that we will need to express in our application-specific logic. Obviously, to describe Alice and Bob, our logic will have to have a notion of principals. Alice wishes to access a particular web page—we will need a way of describing the particular URL that she wants to reach. Bob believes that it's OK for certain principals to access the web page, so we will need a notion of belief. Furthermore, we will need delegation, to describe Bob relinquishing to the Registrar the responsibility to decide who is taking his class. Since Bob's security policy specifies that a page is accessible only after 8 P.M., our logic will also need to reason about time.

The example is not developed in sufficient detail to accurately describe a similar, real situation. For instance, we assume that the principals all know each other's public keys.

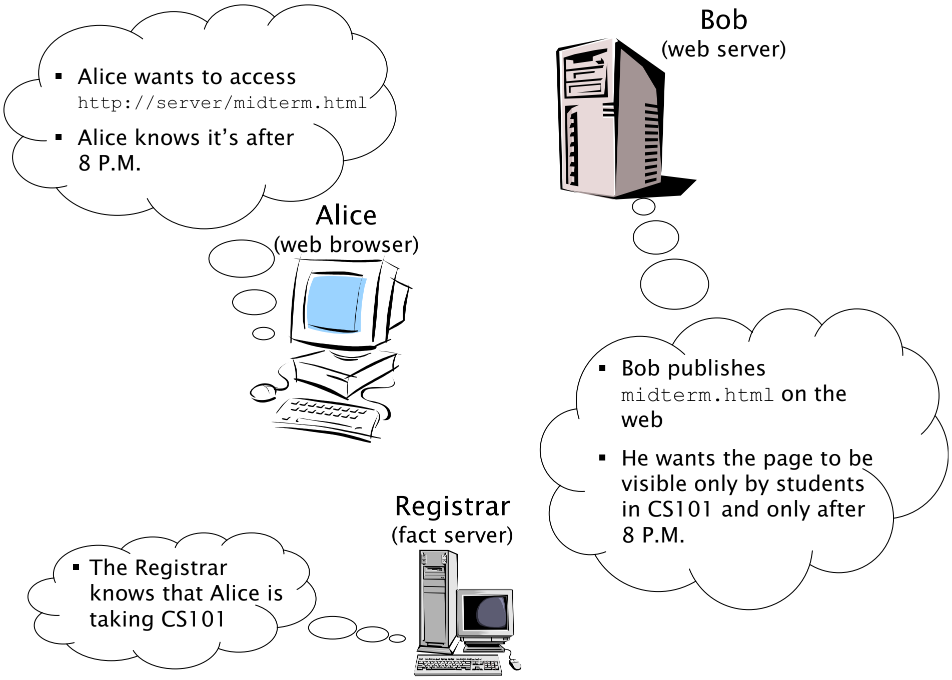
Alice wants to access http://server/midterm.html
Alice knows it's after 8 P.M.
Alice
(web browser)
Bob
(web server)
Bob publishes midterm.html on the web
He wants the page to be visible only by students in CS101 and only after 8 P.M.
Registrar
(fact server)
The Registrar knows that Alice is taking CS101

Figure 2.1: A typical scenario for access control on the web.

As the reader will see, however, the logic we develop to describe the example can easily be extended to model a more complicated situation; for instance, one where the principals use a public key infrastructure. This and other extensions to the logic are discussed in Chapter 5.

2.1 Syntax

Our application-specific logic is inhabited by terms and formulas.

The terms denote strings, natural numbers, and principals. Strings and natural numbers are the base types of our application-specific logic.

Simple principals are created by applying the **name** constructor to the string that represents a principal's public key. For example, if $\texttt{pubkey}_{\texttt{Alice}}$ is Alice's public key, then **name**$(\texttt{pubkey}_{\texttt{Alice}})$ is the principal that represents Alice. For simplicity, we will often abbreviate **name**$(\texttt{pubkey}_{\texttt{Alice}})$ as *Alice*.

Principals may want to refer to other principals or to create local name spaces—this gives rise to the notion of compound principals. We will write $\mathit{Alice}.\texttt{secretary}$ to denote the principal whom Alice calls "secretary." The nature of this principal will be determined by the privileges with which Alice endows him. $\mathit{Alice}.\texttt{secretary}$, for example, could be a principal that represents an actual person, or a role that Alice adopts. For now, we will allow only a single level of local names.

More formally, the terms of our application-specific logic can be described as follows:

$$\begin{array}{rcl} t & ::= & s \mid n \mid p \\ p & ::= & \mathbf{name}(s) \mid \mathbf{name}(s).s \end{array}$$

where s and n are strings and natural numbers, and **name** ranges over strings.

The formulas of our logic describe principals' beliefs. If Alice believes that the formula F is true, we write *Alice* **says** F. To indicate that she believes a formula is true, a principal signs it with her private key—we describe this in logic with the **signed** predicate. If Alice cryptographically signs the formula F, the resulting sequence of bits will be represented with the formula $\texttt{pubkey}_{\texttt{Alice}}$ **signed** F.

To describe the URL that a client wants to access, we introduce the **goal** constructor. The first parameter to this constructor is a string representing the URL. In a PCA system a client obtains access to a resource by presenting a proof—the proof acts as a capability that allows access to the resource. To make sure that a proof cannot be reused, we make the second parameter of the **goal** constructor a nonce. A principal believes the formula

goal$(URL, nonce)$ if she thinks that it is OK to access *URL* during the PCA session identified by *nonce*.

Delegation is described with the **speaksfor** and **delegate** predicates. The formula *Alice* **speaksfor** *Bob* indicates that Alice is authorized to speak on Bob's behalf; in other words, that Bob has delegated to Alice the authority to make access-control decisions about any URL. **delegate** $(Bob,\ Alice,\ URL)$ transfers to Alice only the authority to access the particular resource at *URL*.

To indicate that a delegation holds only after a given moment, delegation statements can be enclosed in an **after** statement. The formula **after** $(N,\ F)$ indicates that the formula F is true after time N. The time N is represented as a natural number, but in examples we will often use the hour of the day, e.g., **after** $(8\ \text{P.M.},\ F)$.

The **says** and **signed** predicates are the only formulas that can occur at top level.

The formulas of our logic are described by the following syntax:

$$\begin{array}{rcl} \phi & ::= & s\ \textbf{signed}\ \phi' \mid p\ \textbf{says}\ \phi' \\ \phi' & ::= & \textbf{goal}\ (s,\ s) \mid \textbf{after}\ (n,\ \phi'') \mid \phi'' \\ \phi'' & ::= & p\ \textbf{speaksfor}\ p \mid \textbf{delegate}\ (p,\ p,\ s) \end{array}$$

where s ranges over strings, n natural numbers, and p principals.

2.2 Context and Judgments

As explained, in our system principals express their beliefs by signing the formulas that represent them. The resulting set of signed statements represents a global environment, or context, which we use as a starting point for reasoning about the beliefs of principals.

$$\Gamma \quad ::= \quad \cdot \mid (s\ \textbf{signed}\ \phi'),\ \Gamma$$

The judgment $\Gamma \vdash F$ means that the formula F is true with respect to some environment Γ. The formula F will be true either if F is itself in the environment, or if F can be derived by applying inference rules to formulas in the environment.

The inference rules of our logic do not change the environment; that is, in each rule the premises and conclusions are all evaluated with respect to the same environment. It is normally clear which statements are in the environment (all statements of the form s **signed** ϕ', and no other statements, are in the environment), so we abbreviate for the sake of readability each judgment $\Gamma \vdash F$ to F.

2.3 Inference Rules

In this section we precisely define the meaning of each operator by specifying the rules that show how it can be derived or how it can be used to derive other connectives.

Rules that show how an operator can be constructed are called introduction rules; rules that show how an operator can be decomposed or used to derive other operators are elimination rules. The names of our rules will have a -I or -E suffix to indicate whether the rule is an introduction or an elimination rule.

The purpose of the **signed** operator is to reflect digital signatures which exist outside of the logic. There are no rules, therefore, which would allow us to manipulate the parameters of the **signed** operator, since the existence of such rules would suggest that we are able to forge a signature.

The existence of a signature implies that the signer believes something. Hence our first rule, which transforms a **signed** statement into a statement that represents a notion of belief which we can manipulate.

$$\frac{\mathit{pubkey}\ \textbf{signed}\ F}{(\textbf{name}(\mathit{pubkey}))\ \textbf{says}\ F} \qquad (\textsc{says-i})$$

Rather than defining general rules that describe the behavior of the **says** operator, as do other security logics, we will limit ourselves to describing the meaning of the **says** operator only with respect to delegation and time.

The principle behind the inference rules that describe delegation is that if a principal *A* believes that he is delegating his own authority to principal *B*, then this delegation should hold. Delegating someone else's authority, on the other hand, is meaningless.

If *A* indicates that *B* speaks on *A*'s behalf then it should be true that anything that is said by *B* has the same force that it would if it were said directly by *A*. This is a general delegation—*A* is delegating to *B* the right to make access-control decisions about any resource.

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A) \quad B\ \textbf{says}\ (\textbf{goal}(\mathit{URL},\mathit{nonce}))}{A\ \textbf{says}\ (\textbf{goal}(\mathit{URL},\mathit{nonce}))} \qquad (\textsc{speaksfor-e})$$

Similarly, a principal is allowed to delegate the privileges that are held by the local names that he controls.

$$\frac{A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A.S) \quad B\ \textbf{says}\ (\textbf{goal}(\mathit{URL},\mathit{nonce}))}{A.S\ \textbf{says}\ (\textbf{goal}(\mathit{URL},\mathit{nonce}))} \qquad (\textsc{speaksfor-e}2)$$

More specific delegation statements, with which A grants to B the right to control only a particular URL, follow the same scheme.

$$\frac{A\ \textbf{says}\ (\textbf{delegate}\ (A,\ B,\ U)) \quad B\ \textbf{says}\ (\textbf{goal}(U,N))}{A\ \textbf{says}\ (\textbf{goal}(U,N))} \qquad (\textsc{delegate-e})$$

$$\frac{A\ \textbf{says}\ (\textbf{delegate}\ (A,\ B.S,\ U)) \quad B.S\ \textbf{says}\ (\textbf{goal}(U,N))}{A\ \textbf{says}\ (\textbf{goal}(U,N))} \qquad (\textsc{delegate-e}2)$$

Both specific and general delegations can be made conditional on the current time. If A believes that a delegation becomes valid at time N, then at any point after time N we can conclude that the principal A believes that the delegation statement holds.

$$\frac{A\ \textbf{says}\ (\textbf{after}\ (N,\ F))}{A\ \textbf{says}\ F}\ \textit{time} > N \qquad (\textsc{after-e})$$

2.4 Putting Our Logic to Work

The logic we have developed is sufficiently expressive to precisely describe the scenario of Alice, Bob, and the Registrar. We will first show how each of the principal’s beliefs is modeled using our logic, and then explain how these statements can be manipulated by the inference rules of our logic to demonstrate that Alice should be allowed access to the midterm results web page.

2.4.1 Beliefs

To describe his beliefs, a principal signs each of a list of formulas that represent them. In the case of our example, each principal's beliefs can be expressed using a single formula.

Bob believes that the midterm results should be accessible only after 8 P.M., and then only to students that the Registrar believes are taking his class. The Registrar maintains a local name corresponding to each class taught at the university. The local name that describes Bob's class is CS101, so the principal that represents the students in Bob's class is called *Registrar*.`CS101`. Bob delegates the right to control access to `midterm.html` to the principal *Registrar*.`CS101`, but makes the delegation conditional on the current time by enclosing it in an **after** statement.

To make it easier to refer to this formula in the future, we call it $\mathcal{P}_1$.

$$\mathcal{P}_1 = \texttt{pubkey}_{\texttt{Bob}}\ \textbf{signed}\ (\textbf{after}\ (8P.M., \ \textbf{delegate}\ (Bob,\ Registrar.\texttt{CS101},\ \texttt{midterm.html})))$$

The Registrar believes that Alice is taking Bob's class. He indicates this by signing a formula stating that Alice speaks on behalf of the principal *Registrar*.`CS101`.

$$\mathcal{P}_2 = \texttt{pubkey}_{\texttt{Registrar}}\ \textbf{signed}\ (Alice\ \textbf{speaksfor}\ Registrar.\texttt{CS101})$$

Alice believes that it's OK for her to access the midterm results.

$$\mathcal{P}_3 = \texttt{pubkey}_{\texttt{Alice}}\ \textbf{signed}\ (\textbf{goal}(\texttt{midterm.html}, \texttt{nonce}))$$

Bob will provide Alice with the particular nonce `nonce` that she will have to use in her proof; this and other details regarding how principals communicate are described in Chapter 4.

2.4.2 Proving Access

Alice's task is now to determine whether and how Bob's and the Registrar's beliefs can be used to convince Bob that he believes that it's OK for Alice to be given access to the midterm results web page. In other words, Alice must prove the following formula.

$$\textit{Bob}\ \textbf{says}\ (\textbf{goal}(\texttt{midterm.html}, \texttt{nonce}))$$

Since she wants to reason about the implications of Bob's and the Registrar's beliefs, Alice's first step will be to use the (SAYS-I) inference rule to convert each of the pieces of the security policy (formulas $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$) into a **says** statement that she can manipulate. The most interesting piece of the security policy is Bob's, since ultimately only his beliefs can be used to conclude that Bob believes that access should be authorized.

$$\frac{\mathcal{P}_1}{\textit{Bob}\ \textbf{says}\ (\textbf{after}\ (8\textit{P.M.},\ \textbf{delegate}\ (\textit{Bob},\ \textit{Registrar}.\texttt{CS101},\ \texttt{midterm.html})))}\ \text{by SAYS-I}$$

Bob's delegation statement does not become valid until 8 P.M. Knowing that it's currently 9 P.M., Alice can assume that the time according to Bob's clock must be after 8 P.M., so she asserts that the local time is after 8 P.M. and reasons that Bob also believes this.

$$\frac{\textit{Bob}\ \textbf{says}\ (\textbf{after}\ (8\textit{P.M.},\ \textbf{delegate}\ (\textit{Bob},\ \textit{Registrar}.\texttt{CS101},\ \texttt{midterm.html})))}{\textit{Bob}\ \textbf{says}\ (\textbf{delegate}\ (\textit{Bob},\ \textit{Registrar}.\texttt{CS101},\ \texttt{midterm.html}))}\ \text{by AFTER-E}$$

Alice has succeeded in demonstrating that Bob delegates to the Registrar the authority to access the midterm results. Before she can make use of that fact she has to show that the Registrar, too, believes that access should be granted. To do this, she combines her own assertion that access should be granted with the Registrar's delegation statement that makes her a member of the group CS101.

$$\dfrac{\begin{array}{c}\textit{Registrar}\ \mathbf{says}\ (\textit{Alice}\ \mathbf{speaksfor}\ \textit{Registrar}.\mathtt{CS101}) \\ \textit{Alice}\ \mathbf{says}\ (\mathbf{goal}(\mathtt{midterm.html},\mathtt{nonce}))\end{array}}{\textit{Registrar}.\mathtt{CS101}\ \mathbf{says}\ (\mathbf{goal}(\mathtt{midterm.html},\mathtt{nonce}))}\ \text{by }\textsc{speaksfor-e}2$$

The final step is to combine this result with Bob's statement that delegates authority to the Registrar.

$$\dfrac{\begin{array}{c}\textit{Bob}\ \mathbf{says}\ (\mathbf{delegate}\ (\textit{Bob},\ \textit{Registrar}.\mathtt{CS101},\ \mathtt{midterm.html})) \\ \textit{Registrar}.\mathtt{CS101}\ \mathbf{says}\ (\mathbf{goal}(\mathtt{midterm.html},\mathtt{nonce}))\end{array}}{\textit{Bob}\ \mathbf{says}\ (\mathbf{goal}(\mathtt{midterm.html},\mathtt{nonce}))}\ \text{by }\textsc{delegate-e}2$$

Alice has successfully proven that the beliefs of the three principals are cumulatively sufficient to demonstrate to Bob that she is allowed to see the web page. The derivation that shows that Bob believes that it's OK to access the web page represents a proof of this fact.

$$\dfrac{\dfrac{\dfrac{\mathcal{P}_1}{\textit{Bob}\ \mathbf{says}\ldots}\ \text{by }\textsc{says-i}}{\textit{Bob}\ \mathbf{says}\ldots}\ \text{by }\textsc{after-e} \qquad \dfrac{\dfrac{\mathcal{P}_2}{\textit{Registrar}\ \mathbf{says}\ldots}\ \text{by }\textsc{says-i} \quad \dfrac{\mathcal{P}_3}{\textit{Alice}\ \mathbf{says}\ldots}\ \text{by }\textsc{says-i}}{\textit{Registrar}.\mathtt{CS101}\ \mathbf{says}\ldots}\ \text{by }\textsc{speaksfor-e}2}{\textit{Bob}\ \mathbf{says}\ (\mathbf{goal}(\mathtt{midterm.html},\mathtt{nonce}))}\ \text{by }\textsc{delegate-e}2$$

2.5 Soundness

We would like to be able to argue that our logic accurately describes the authorization problems we are trying to solve. In particular, we would like to confirm that a formula like *Bob* **says** $(\mathbf{goal}(\mathit{URL}, \mathit{nonce}))$ is provable only if it is the case that the principal *Bob* is willing to give access to *URL*. This property is called soundness.

More formally, a logic is said to be sound if $\Gamma \vdash F$ implies that $\Gamma \models F$; that is, if a formula F can be proven from a set of assumptions Γ then, under any model, that formula must be true if the assumptions in Γ are true. Whether or not the assumptions and conclusion are true depends on the model that gives the logic its meaning. To prove that a logic is sound, therefore, we need to have a model for the logic.

Soundness is especially important in security applications. The purpose of an access-control logic, for example, is to make sure that only authorized people can gain access to a resource. An unsound logic could easily lead to unauthorized access of protected resources, so proofs of soundness of such a logic are of more than just theoretical interest.

For a security logic to be trustworthy, we would ordinarily have to give it a semantic model and then prove the logic sound under that model. This is often a complicated and tedious task; whenever a logic is changed, for example to accommodate a new kind of delegation, soundness must be proven anew.

PCA simplifies the task of writing trustworthy application-specific logics. Each application-specific logic is given a semantics by defining all its operators in terms of the underlying PCA higher-order logic. Higher-order logic is known to be sound, which guarantees that any logic that can be expressed in it is also sound [4, Thm. 5402]. Hence, the security-logic designer need only provide an encoding of his logic into higher-order logic, and he gets soundness almost for free.

Chapter 3 describes the encoding of our application-specific logic into higher-order logic and explains how this guarantees soundness.

Although the semantics of the application-specific logic is defined by its encoding into higher-order logic, i.e., we do not yet have a formal model for our application-specific logic, it is still useful to be able to informally argue that the logic “makes sense.” To “make sense” means that it is not possible to mistakenly conclude that a principal is willing to give access to a resource (i.e., that $A\ \textbf{says}\ (\textbf{goal}(URL, nonce)))$.

There are two ways to conclude that a principal is willing to give access to a URL $(A\ \textbf{says}\ (\textbf{goal}(URL, nonce)))$. One way is for the principal to sign a statement of the appropriate goal $(pubkey_A\ \textbf{signed}\ (\textbf{goal}(URL, nonce)))$; the other is for that principal to delegate authority (either in general, or just over that URL) to a principal that is willing to give access to the URL (e.g., $A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A)$ and $B\ \textbf{says}\ (\textbf{goal}(URL, nonce)))$.

Neither case allows us to arrive at the conclusion in error. In the first case, the formula representing a signed statement is derived from a sequence of bits generated by the principal; the formula cannot be created by applying the inference rules of our logic, so it cannot be falsified or arrived at by mistake.

The second case is safe if neither the delegation statement nor the beliefs of the principal to whom rights are being delegated can be falsified. A formula that represents a delegation statement (e.g., $A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A)$) can only be derived if the principal A has certified the delegation by signing it (e.g., $pubkey_A\ \textbf{signed}\ (B\ \textbf{speaksfor}\ A)$ or $pubkey_A\ \textbf{signed}\ (\textbf{after}\ (N,\ (B\ \textbf{speaksfor}\ A)))$). That leaves the principal to whom privileges were delegated as the only potential weak point. However, this principal’s beliefs can also be arrived at only through signing or delegation; hence, by induction, they are valid. It is not possible to arrive at the conclusion that $A\ \textbf{says}\ (\textbf{goal} URL, nonce)$ in an unintended manner; therefore, our logic “makes sense.”

2.6 Decision Procedure

One of the goals of designing a simple application-specific logic is that there should be an efficient decision procedure for finding proofs of true formulas. In most distributed authorization systems this is critical, because the server that runs the decision procedure must always correctly conclude whether or not to grant access. In PCA systems the decidability of an application-specific logic is not crucial, since an inefficient or undecidable logic will stymie only the particular client using it, not the server or the system as a whole. However, it is certainly desirable that an application-specific logic be decidable, because even if its undecidability cannot harm the server or the system, it makes the logic much less useful, if not completely useless, to the client. In this section we present an informal argument of why our logic always allows a correct, polynomial-time access-control decision, and describe a simpler algorithm we use in practice.

When using our application-specific logic in scenarios such as the one between Alice, Bob, and the Registrar, the client's task will always be to prove that a formula of the form Server **says** $(\mathbf{goal}(\mathit{URL},\mathit{nonce}))$ can be derived from the premises that comprise the security policy. Since our logic is straightforward, it is not difficult to describe a polynomial-time algorithm that proves formulas of this kind.

Each formula in the context has the form S **signed** F, where F is either a goal statement or a delegation statement, and optionally valid only after a certain time. To find whether the initial assumptions support the formula that describes the goal, we represent the assumptions as a directed graph in which nodes represent principals and edges represent delegations.

Suppose that we want to find a proof of Server **says** $(\mathbf{goal}(\mathtt{URL},\mathtt{nonce}))$. First, we discard unneeded assumptions. These can be delegation statements in which a principal attempts to delegate someone else's authority (e.g., $pubkey_A$ **signed** $(B$ **speaksfor** $C))$ or delegation statements which delegate authority over a resource other than `URL`. The former are always invalid, since there are no inference rules for manipulating them, and the latter are useless for finding the proof we are currently interested in, since there are no rules that can connect delegation statements about two different resources.

Second, we process every formula from the context. If the formula is an assertion, and if it asserts the correct goal (i.e., the one with `URL` and `nonce`), we add to the graph a node for the principal that made the assertion, and label it as start node. If the node for that principal already exists, we merely label it a start node. If the formula is a delegation, we add nodes for the source and the target of the delegation, and an edge from the target to the source (i.e., for the formula $pubkey_A$ **signed** $(B$ **speaksfor** $A)$ we add nodes for A and B and an edge from B to A). We do not add any edges or nodes that already exist in the graph. If the formula is a delegation that is valid only after a time N, we either discard it or treat it as a regular delegation, depending on whether the current time is before or after N.

Proving the formula $Server$ **says** $($**goal**$(\texttt{URL}, \texttt{nonce}))$ is akin to finding a path from a source node to the node that represents the principal $Server$. The path can be found using a depth-first search. All potentially useful delegations are represented as edges in the graph and a depth-first search will touch all edges reachable from the source node, so if a path cannot be found then there can exist no proof of the goal, and vice versa.

Both the search and the construction of the graph take polynomial time in the number of initial assumptions. During the construction of the graph each assumption is considered only once. Furthermore, each assumption can lead to the creation of at most two nodes and one edge, so if there are N assumptions there can be at most $2N$ nodes and N edges. If the running time of the depth-first search is $O(v+e)$, it will be $O(2N + N) = O(N)$ when expressed relative to the number of assumptions. Hence, determining whether a set of assumptions supports a particular access-control decision takes linear time.

Figure 2.2 shows how this algorithm can be used to show that Alice is allowed to access Bob's URL. The security policy (represented by formulas $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$) is the same as in Section 2.4. The first processing pass discards unneeded formulas. None of $\mathcal{P}_1$ through $\mathcal{P}_3$ is unneeded or invalid. The second pass constructs the graph. First considered (though the order is unimportant) is Bob's delegation. The delegation holds only after 8 P.M.; assuming that the current time is after 8 P.M. it is treated as a normal delegation. Nodes representing *Bob* (the issuer of the delegation) and *Registrar*.`CS101` are added to a graph, as well as an edge from `CS101` to *Bob* (Figure 2.2a). Next examined is the Registrar's delegation, which gives *Alice* the authority to speak on behalf of *Registrar*.`CS101`. The graph already contains a node for *Registrar*.`CS101`, so only a node representing *Alice* is added, as well as an edge from *Alice* to *Registrar*.`CS101` (Figure 2.2b). The last piece of the security policy is Alice's assertion that it is OK to access the URL. Again, a node for *Alice* already exists in the graph, so we merely label it as the start node (Figure 2.2c). The graph is now fully constructed. To determine whether Alice may access Bob's URL, we look for a path from the node labeled "Start" to Bob's node (Figure 2.2d). A path (show dashed) exists, so access should be allowed.

In practice, we make access-control decisions using an algorithm that is easier to implement.

The search performed by the graph algorithm can also be done by a tactical theorem prover operating directly on the set of initial assumptions without building a graph. The running time of DFS without marking on an acyclic graph is exponential in the worst case. In our logic principals may mutually delegate authority to each other (e.g.,

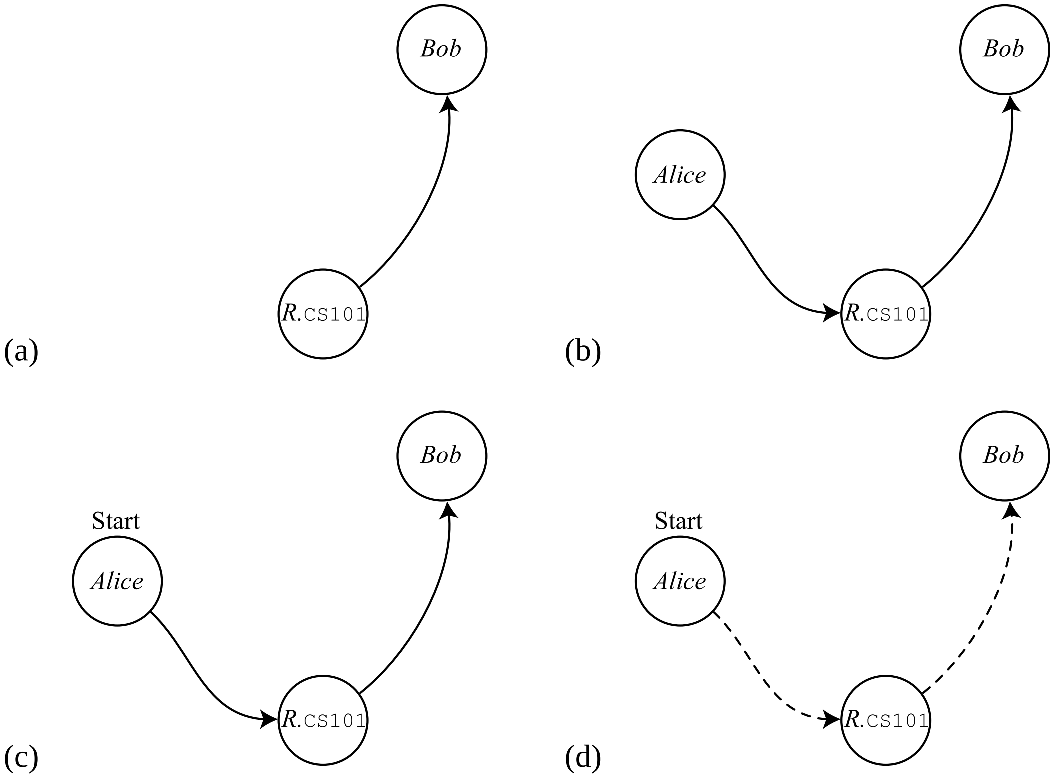

Figure 2.2: A graph generated from Alice's, Bob's, and the Registrar's beliefs. Bob's delegation statement, which holds since the time is currently after 8 P.M., is processed first (a), followed by the Registrar's specific delegation (b), and Alice's statement of goal (c). The existence path from the node labeled "Start" to the node representing Bob demonstrates that Alice is able to access the URL (d).

$pubkey_A$ **signed** $(B$ **speaksfor** $A)$ and $pubkey_B$ **signed** $(A$ **speaksfor** $B))$, so the graph may be cyclic and the running time of DFS without marking infinite. In both situations, however, DFS with marking would be exponential in the worst case.

To make our tactical prover as simple as possible, we constrain the set of initial assumptions by disallowing cyclic delegations, and search for the proof using DFS without marking.

Each tactic in the theorem prover corresponds to an inference rule of the logic (i.e., the tactic that corresponds to the SPEAKSFOR-E rule specifies that a proof of the formula A **says** $(\mathbf{goal}(\mathtt{URL}, \mathtt{nonce}))$ can be derived from the proofs of A **says** $(B$ **speaksfor** $A)$ and

B **says** $(\mathbf{goal}(\texttt{URL}, \texttt{nonce}))$. The theorem prover thus works in the opposite direction from the graph algorithm, following delegation chains from the goal to assertions. Although the theorem prover is less general, it is more convenient for our purposes, as will be discussed in Section 4.1.3. The tactics that comprise our prover are shown in Appendix A.

Chapter 3

Semantics for an Access-control Logic

The second step in building a PCA system is to give the application-specific logic a semantics in the PCA logic. We will begin by explaining the PCA logic in detail, after which we will describe how we use the PCA logic to encode our application-specific logic. Finally, we will explain why giving such a semantics to the application-specific logic guarantees soundness.

3.1 The PCA Framework

The PCA logic is standard higher-order logic with a few extensions that make it more suitable for defining security logics.

Before we delve into the details of the PCA logic it is useful to review why higher-order logic is particularly useful for our purposes. One might ask, for example, why we do not use a simpler logic, like predicate logic; or one with more built in operators, like a linear or a modal logic.

One of our primary requirements for a substrate for defining logics is that it be sufficiently powerful to describe a broad range of application-specific logics. Many security logics, such as our application-specific logic, have higher-order features like relations that range over formulas (the **says** relation, for example)—these are expressed most naturally in higher-order logic. A decidable logic like propositional logic is not powerful enough for our purposes. Although some *k*th-order logic could conceivably be sufficient to describe the relations we desire, there is nothing to be gained by using a *k*th-order logic in favor of higher-order logic; higher-order logic is both more general and provides for a more natural encoding of many relations. In addition, in many cases higher-order logic makes it possible to write proofs about *k*th-order terms more efficiently than it would be possible in *k*th-order logic [5, Section 1.4]. Partly thanks to logical frameworks like HOL and Isabelle, the convenience of using higher-order logic as a tool for describing logics has been well established.

Higher-order logic allows us to encode notions like modality and linearity (as we will see, for example, **says** is a modal operator). One could argue that it would sometimes be more convenient for those notions to be built into the logic. In our experience, however, defining such operators using higher-order logic has not been unduly complicated or inefficient. In addition, using higher-order logic makes our framework more general, and the trusted computing base smaller, than it would be if we used a more specialized logic.

3.1.1 Higher-order Logic

Our presentation of higher-order logic is standard. The simple types are numbers, strings, principals, and formulas; compound types are functions from types to types and pairs. The primitive constructors of the logic allow function abstraction (λ) and application,

Figure 3.1: The inference rules of higher-order logic.

$$\dfrac{\begin{array}{c}[A]\\ B\end{array}}{A \rightarrow B}\ \rightarrow\text{I} \qquad \dfrac{A \rightarrow B \quad A}{B}\ \rightarrow\text{E}$$

$$\dfrac{[y/x]A}{\forall x.A(x)}\ \forall\,\text{I} \qquad \dfrac{\forall x.A(x)}{A(Y)}\ \forall\,\text{E}$$

$$\dfrac{F}{(\lambda x.[x/Y]F)(Y)}\ \beta\,\text{I} \qquad \dfrac{(\lambda x.F)(Y)}{[Y/x]F}\ \beta\,\text{E}$$

$$\dfrac{}{F(X) \rightarrow F(\mathrm{fst}(\langle X,Y\rangle))}\ \text{FST} \qquad \dfrac{}{F(X) \rightarrow F(\mathrm{snd}(\langle Y,X\rangle))}\ \text{SND}$$

$$\dfrac{}{\neg(\neg(F)) \rightarrow F}\ \text{NOT-NOT-E}$$

implication (→), universal quantification (∀), and creating and decomposing pairs. The logic contains nine inference rules for manipulating these constructors (Figure 3.1), as well as a set of rules about arithmetic.

Other standard operators (e.g., ∧, ∨) can be defined as abbreviations from the existing ones (see Figure 3.2 for some examples). The standard introduction and elimination rules can be proved as theorems based on these abbreviations.

3.1.2 Extensions

In addition to the standard features of higher-order logic, the PCA logic contains a few primitives that are useful specifically for defining security logics. In the present work we have refined the PCA logic so that the number of additions to higher-order logic is minimal.

Figure 3.2: Common constructors defined as abbreviations.

$$\begin{array}{rcl}
A = B & \overset{\text{def}}{=} & \forall P.P(B) \to P(A) \\
A \wedge B & \overset{\text{def}}{=} & \forall C.(A \to B \to C) \to C \\
A \vee B & \overset{\text{def}}{=} & \forall C.(A \to C) \to (B \to C) \to C \\
\exists F & \overset{\text{def}}{=} & \forall B.(\forall X.F(X) \to B) \to B \\
\bot & \overset{\text{def}}{=} & \forall A.A \\
\neg A & \overset{\text{def}}{=} & A \to \bot
\end{array}$$

Most security logics are likely to have some notion of principals, proof goals, cryptography, and time. A framework for developing security logics should make it convenient to represent these ideas. At the same time, the mechanism that makes embedding these ideas in the PCA logic convenient should not restrict the semantics that they might have in different security logics.

To represent principals, we add to the PCA logic the type `worldview`. This type is implemented as an abbreviation for the `string` type. A user of the logic, however, need not be aware of this; to him, `worldview` is an abstract type whose implementation remains opaque.

$$\texttt{worldview} = \texttt{string}$$

To create terms of type `worldview`, we use the **name** constructor.

$$\textbf{name} : \texttt{string} \to \texttt{worldview}$$

Unlike principals, which are terms, both proof goals and cryptographic primitives can more naturally be represented as formulas. The constructor for goals allows a goal specified with two strings to be interpreted as a formula.

$$\textbf{goal} : \texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{formula}$$

The only cryptographic construct supported in the PCA logic is a public-key signature, which is specified by a public key and a formula that was signed by the corresponding private key.

$$\textbf{signed} : \texttt{string} \rightarrow \texttt{formula} \rightarrow \texttt{formula}$$

If we wanted to use the PCA logic to encode security logics that used other cryptographic primitives, such as group signatures, for example, it might be helpful to add additional constructors.

An important component of most security logics is the notion of time. To support such logics the PCA logic has the constant **localtime**, a natural number whose value represents the current (universal standard) time on the local host. For any number N that is greater than the current value of **localtime**, the system will provide users with a proof that **localtime** $> N$; corresponding proofs will also be made available about numbers less than **localtime**. Details of how these proofs can be accessed by a user will be described in Chapter 4.

We added the constructors described in this section because they provided a convenient way of expressing concepts that we encountered while designing security logics. The PCA logic is intended to be used to describe arbitrary security logics, so we did not endow these constructors with any meaning particular to our application-specific logic. In particular, we have added to the PCA logic no inference rules that describe how the constructors behave, which leaves the semantics of each constructor up to the designer of any particular application-specific logic.

Since we have added no rules, our extensions fit neatly into higher-order logic. Type theory permits constants of arbitrary type [5, Section 1.2], which allows the inclusion of our constructors, **name**, **goal**, and **signed**, and the **localtime** constant.

3.2 Defining Operators and Inference Rules

Our task is now to devise a definition, in the PCA logic, of each of the operators of our application-specific logic. The operators should be defined in a way that makes it possible to derive from their definitions all the inference rules (e.g., SPEAKSFOR-E) of the application-specific logic. In other words, each of the inference rules presented in Chapter 2 must be proved as a theorem.

All the definitions and theorems presented in this chapter have been checked by machine.

Some of the operators of the application-specific logic map cleanly onto the operators provided by the PCA logic and need no definition. The **signed** operator of the application-specific logic, for example, can be represented by the same operator of the PCA logic. Because formulas like S **signed** F cannot be deduced using the inference rules of the application-specific logic (or by inference rules of the PCA logic), the semantics of the **signed** operator will be specified by the definitions of the operators (like **says**) that make use of it. Similarly to the **signed** operator, **name** and **goal** map directly to the appropriate constructors of the PCA logic.

3.2.1 Belief

The key notion in our application-specific logic is the notion of belief. A server's willingness to let a resource be accessed, a client's intention to access it, and delegation statements are all couched as beliefs. This makes it particularly important for the definition of belief to be accurate, because even a slightly misstated definition could affect any reasoning done with our logic (unlike, for example, the definition of a particular kind of delegation, which, if incorrect, might not affect the behavior of any other operators).

We contend that principals should be both rational and accountable[1]. Before we formulate the definition of the **says** operator, we should try to exactly characterize the beliefs of such a principal. It is clear, for example, that a formula like $pubkey_A$ **signed** F should imply that A believes F, i.e., A **says** F. This behavior, and others, we specified in our application-specific logic by the inference rules that make use of the **says** operator. From these inference rules, and from our intuition about the meaning of the **says** operator, we want to extrapolate a more general set of characteristics that describe the operator's behavior. Since our goal is to devise a definition in the PCA logic, we want these characteristics also to be expressed in the PCA logic.

From the application-specific rules about delegation (e.g., SPEAKSFOR-E and DELEGATE-E), and according to the principle of accountability, we can conclude that a particular belief (e.g., A **says** F) can be the result of some other belief held by the same principal (e.g., A **says** $(B$ **speaksfor** $A))$ combined with an external fact (e.g., B **says** F). If a principal believes a set of formulas he should also believe all the formulas that can be derived from that set. In other words, something very similar to the modus ponens rule should hold within the **says** operator. It seems reasonable that a principal's beliefs should be consistent both internally and in combination with facts that are globally true, hence this modus-ponens-like rule should allow as premises both facts that A believes and facts that are generally true. We can express this more formally using the following two rules.

[1] While this may be a poor model of reality, it is nevertheless a useful and reasonable assumption for a security logic.

$$\frac{F}{A \textbf{ says } F} \tag{3.1}$$

$$\frac{A \textbf{ says } (G \to F) \quad A \textbf{ says } G}{A \textbf{ says } F} \tag{3.2}$$

The second rule (3.2) ensures that A's beliefs are internally consistent. The first rule (3.1) requires that A believe anything that is globally true; together with the second rule, this means one of A's beliefs can be combined with a generally true statement to derive a new belief, which is exactly what happens in the application-specific rules that describe delegation.

Rules 3.1 and 3.2 account for the behavior of **says** in the application-specific rules about delegation, but they do not help explain the SAYS-I rule. The SAYS-I rule takes as a premise a specific sort of formula that cannot be further decomposed, so there is no more general way of characterizing the behavior the rule describes. Hence, we make SAYS-I the third rule that describes the behavior of the **says** operator.

$$\frac{S \textbf{ signed } F}{(\textbf{name } S) \textbf{ says } F} \tag{3.3}$$

Now that we have described the high-level behavior of **says** with several rules written in the PCA logic (3.1–3.3), we need to define **says** in a way that embodies those rules. The most precise way to do so is to define **says** as the most restrictive operator that obeys those rules. In particular, if there are infinitely many two-argument relations S that range over principals and formulas, a subset of them will relate principals to formulas according to the rules we want **says** to follow. We quantify over all possible relations, so one of the relations in that subset will relate principals to formulas only according to those three rules—we define the **says** operator to be that relation.

$$\begin{aligned} A\ \textbf{says}\ F \quad \stackrel{\text{def}}{=} \quad & \forall S\, \forall A'\, \forall F'\, \forall G'\, \forall K. \\ & \big(\big(F' \to S\,(A', F')\big) \\ & \ \wedge\ \big(\big(S\,(A', F')\ \wedge\ S\,(A',\ (F' \to G'))\big) \to S\,(A', G')\big) \\ & \ \wedge\ \big(K\ \textbf{signed}\ F' \to S\,\big(\textbf{name}(K), F'\big)\big)\big) \\ & \to S(A, F) \end{aligned}$$

The **says** operator is the intersection of all the relations S that obey the three rules.

Now that we have defined the **says** operator, we should be able to use its definition to prove some theorems about the operator's behavior. We have yet to define the **speaksfor** and **delegate** operators, so we cannot prove as theorems all the inference rules of our application-specific logic, but we can at least prove that **says** obeys the properties described by rules 3.1–3.3. Once proved as theorems, these properties will become the introduction rules for the **says** operator. Rule 3.3 is the SAYS-I rule of the application-specific logic; proving it as a theorem confirms that at least one of the inference rules of our application-specific logic can be derived from the definitions of the **says** operator. Rules 3.1 and 3.2 are important properties that we will often use in proofs; we will prove them as theorems also, and label them SAYS-I2 and SAYS-I3.

$$\frac{S\ \textbf{signed}\ F}{(\textbf{name}\ S)\ \textbf{says}\ F} \tag{SAYS-I}$$

$$\frac{F}{A\ \textbf{says}\ F} \tag{SAYS-I2}$$

$$\frac{A\ \mathbf{says}\ (G \rightarrow F) \quad A\ \mathbf{says}\ G}{A\ \mathbf{says}\ F} \tag{SAYS-I3}$$

To demonstrate how to prove theorems from the definition of **says**, let us prove SAYS-I2, i.e., that a principal believes anything that is globally true. The proof follows directly from the three rules embodied in the definition of **says**, and is shown in Figure 3.3. Note that applications of β rules are omitted in the proof.

The proof demonstrates that the premise, F, can be used to derive the body of the definition of the **says** operator, which is definitionally equivalent (line 9) to deriving A **says** F. The parameters of **says** are therefore in scope throughout the body of the proof (lines 1–8). The outermost layer of the definition of **says** is the universally quantified variable that represents says-like relations. By proving that the inner layer can be derived with respect to any says-like relation (lines 2–7), we demonstrate that it holds for all such relations (line 8). This inner layer demonstrates that the rules 3.1–3.3, with the variables universally quantified, imply $S'(A,F)$. This is proven by instantiating the quantified variables with A and F, the arguments of the definition of **says** (line 4). The third variable, G, is unimportant, so we instantiate it with $\bot$ (false). After instantiating the variables, we discard the second two rules, since they are not relevant (line 5) and use the first rule to prove the subgoal $S'(A,F)$ (line 6).

We can use the same technique to prove theorems SAYS-I and SAYS-I3.

In addition to directly obeying each the rules in its definition, the **says** operator also acts in accordance to what the rules as a whole imply. For example, we can prove that A **says** $(A$ **says** $F)$ implies A **says** F. This is consistent with our intuitive idea of belief—barring complicated metaphysical ideas of reality, if Alice believes that she believes a

Figure 3.3: Proof of theorem SAYS-I2: principals believe tautologies.

1	F	premise
2	$[S'_0]$	
3	$\left[\begin{array}{l} \forall A' \forall F' \forall G' \forall K.(F' \to S'_0(A', F')) \wedge \\ \quad ((S'_0(A', F') \ \wedge \ S'_0(A', (F' \to G'))) \to S'_0(A', G')) \wedge \\ \quad (K \ \textbf{signed}\ F' \to S'_0(\textbf{name}\, K, F')) \end{array}\right]$	assumption
4	$(F \to S'_0(A, F) \wedge$	
	$((S'_0(A, F) \ \wedge \ S'_0(A, (F \to \bot))) \to (S'_0(A, \bot))) \wedge$	
	$(K \ \textbf{signed}\ F \to S'_0(\textbf{name}\, K, F)))$	$\forall_{A'F'G'K}$ e 3
5	$F \to S'_0(A, F)$	$\wedge$ e 4
6	$S'_0(A, F)$	$\to$ e 5, 1
7	$\forall A' \forall F' \forall G' \forall K.((F' \to S'_0(A', F')) \wedge$	
	$((S'_0(A', F') \ \wedge \ S'_0(A', (F' \to G'))) \to S'_0(A', G')) \wedge$	
	$(K \ \textbf{signed}\ F' \to S'_0(\textbf{name}\, K, F'))) \to S'_0(A, F)$	$\to$ i 3–6
8	$\forall S' \forall A' \forall F' \forall G' \forall K.((F' \to S'(A', F')) \wedge$	
	$((S'(A', F') \ \wedge \ S'(A', (F' \to G'))) \to S'(A', G')) \wedge$	
	$(K \ \textbf{signed}\ F' \to S'(\textbf{name}\, K, F'))) \to S'(A, F)$	$\forall_{S'}$ i 2–7
9	$A \ \textbf{says}\ F$	$\overset{\text{def}}{=}$ i 8

formula F, then she actually does believe F. We show this proof in Figure 3.4 and call the theorem SAYS-TAUT.

$$\frac{A \textbf{ says } (A \textbf{ says } F)}{A \textbf{ says } F} \qquad (\text{SAYS-TAUT})$$

3.2.2 Local Names

Our application-specific logic has two kinds of principals: principals that are created from public keys, and local name spaces that belong to those principals. The former kind, principals created from public keys, maps cleanly onto the PCA logic. As in the application-specific logic, Alice can be described by the term $\textbf{name}(\texttt{pubkey}_{\texttt{Alice}})$, where **name** is the PCA-logic constructor and the whole term has the (PCA-logic) type `worldview`.

Expressing local names (like *Registrar*.`CS101`) is less straightforward—we do not have the option of using a built-in constructor. As with the **says** operator, before devising a definition for local names in the PCA logic we need to clarify their meaning.

The key notion in the application-specific logic is belief; we are interested in principals as entities that hold beliefs. The beliefs of local names are described only in one rule, SPEAKSFOR-E, which states that if the owner of a local name delegates away its privileges (e.g., A **says** (B **speaksfor** $A.S$)), then the local name believes at least as many formulas as the recipient of the delegation (e.g., if B **says** F then $A.S$ **says** F). This rule is a particular instance of a more general principle—local names believe whatever the principals that own them decide they believe. $A.S$ believes, in other words, whatever A believes that $A.S$ believes. In addition, the prefix of $A.S$ that identifies the principal to

Figure 3.4: Proof of theorem SAYS-TAUT: Alice believes her own beliefs.

1	$A\ \textbf{says}\ (A\ \textbf{says}\ F)$	premise
2	$[\neg(A\ \textbf{says}\ F)]$	assumption
3	$[A\ \textbf{says}\ F]$	assumption
4	$[\neg(A\ \textbf{says}\ F)]$	assumption
5	$\bot$	$\neg$ e 4,3
6	$(A\ \textbf{says}\ F) \rightarrow (\neg(A\ \textbf{says}\ F)) \rightarrow \bot$	$\rightarrow_2$ i 3–5
7	$A\ \textbf{says}\ ((A\ \textbf{says}\ F) \rightarrow (\neg(A\ \textbf{says}\ F)) \rightarrow \bot)$	SAYS-I2 6
8	$A\ \textbf{says}\ ((\neg(A\ \textbf{says}\ F)) \rightarrow \bot)$	SAYS-I3 7,1
9	$A\ \textbf{says}\ \bot$	SAYS-I3 8,2
10	$[\bot]$	assumption
11	F	$\bot$ e 10
12	$\bot \rightarrow F$	$\rightarrow$ i 10–11
13	$A\ \textbf{says}\ (\bot \rightarrow F)$	SAYS-I2 12
14	$A\ \textbf{says}\ F$	SAYS-I3 13,9
15	$\bot$	$\neg$ e 2,14
16	$A\ \textbf{says}\ F$	RAA 2–15

Figure 3.5: Proof of theorem SAYS-I2’: local names believe tautologies.

1	F	premise
2	$S\ \textbf{says}\ F$	SAYS-I2 1
3	$A\ \textbf{says}\ (S\ \textbf{says}\ F)$	SAYS-I2 2

whom the name *S* belongs is needed only for someone like *C* to distinguish between *A.S* and *B.S*. From *A*’s perspective, *A.S* is uniquely identified by the name *S*. Hence, the set of ideas that are held by *A.S* is in fact the set of ideas that *A* believes *S* believes. We need local names only to reason about their beliefs, so instead of defining local names as terms in the PCA logic we can just reason about the beliefs that the local names’ owners believe they hold. *A.S* thus becomes just an abbreviation.

$$A.S\ \textbf{says}\ F \equiv A\ \textbf{says}\ ((\textbf{name}\ S)\ \textbf{says}\ F)$$

We formally defined the notion of belief after deciding that all principals’ beliefs should be consistent according to a particular set of rules. After adopting this abbreviation for local names, we should be able to show that the beliefs of local names are consistent in the same way. We therefore prove theorems about the beliefs of local names analogous to the theorems we proved about the beliefs of simple principals. For illustration we show the SAYS-I2’ theorem, an analogue to SAYS-I2; its proof is in Figure 3.5.

$$\frac{F}{A.S\ \textbf{says}\ F} \qquad (\textsc{says-i2'})$$

Treating local names as an abbreviation is convenient in that it absolves us from having to define another operator, which makes the semantics of our application-specific logic simpler and easier to reason about. It is inconvenient, however, in that local names are no longer a term in the logic, so they cannot be passed to an operator as a single argument. For example, since $A.S$ is not a single term, we cannot write the formula B **speaksfor** $A.S$. Instead, we will have to define an additional version of speaksfor and write $\mathbf{speaksfor}(B,A,S)$. It is possible to maintain the semantics of local names as abbreviations and still allow all principals to be represented as single terms; defining this in the PCA logic is complicated, however, so we defer it to Section 5.3.

3.2.3 General Delegation

Delegation is closely linked to belief. If Alice believes she is delegating her own authority to Bob, then her authority really is being delegated. If Alice believes she is delegating Bob's authority, however, then that belief has no meaning to anyone but Alice.

We want to define a delegation statement to be the particular formula that Alice believes if she wants to delegate her authority to Bob. To delegate authority means to allow Bob's beliefs to influence her own. If Alice has delegated her authority to Bob, then Bob's belief that it is OK to access a certain URL (Bob **says** $(\mathbf{goal}(\mathit{URL},\mathit{nonce}))$) is sufficient to cause Alice to believe this too (Alice **says** $(\mathbf{goal}(\mathit{URL},\mathit{nonce}))$). Therefore, the delegation formula that Alice believes has to make it possible to conclude that Bob's beliefs imply Alice's. It is straightforward to express this in higher-order logic.

$$A \ \mathbf{speaksfor}\ B \stackrel{\text{def}}{=} \forall U \forall N.(A\ \mathbf{says}\ (\mathbf{goal}(U,N))) \to (B\ \mathbf{says}\ (\mathbf{goal}(U,N)))$$

The next step is to prove the SPEAKSFOR-E rule as a theorem. The proof makes use of the SAYS-I3 theorem to conclude that A believes that A believes the goal. The SAYS-TAUT theorem allows us to conclude from this that A believes the goal. The full proof is shown in Figure 3.6.

Figure 3.6: Proof of the SPEAKSFOR-E theorem.

1	A **says** $(B$ **speaksfor** $A)$	premise
2	B **says** $(\mathbf{goal}(U,N))$	premise
3	$[\forall U' \forall N'.B$ **says** $(\mathbf{goal}(U',N')) \rightarrow A$ **says** $(\mathbf{goal}(U',N'))]$	assumption
4	B **says** $(\mathbf{goal}(U,N)) \rightarrow A$ **says** $(\mathbf{goal}(U,N))$	$\forall_{U,N}$ e 3
5	$(\forall U' \forall N'.B$ **says** $(\mathbf{goal}(U',N')) \rightarrow A$ **says** $(\mathbf{goal}(U',N')))$	
	$\rightarrow (B$ **says** $(\mathbf{goal}(U,N)) \rightarrow A$ **says** $(\mathbf{goal}(U,N)))$	$\rightarrow$ i 3–4
6	A **says** $\big((\forall U' \forall N'.B$ **says** $(\mathbf{goal}(U',N')) \rightarrow A$ **says** $(\mathbf{goal}(U',N')))$	
	$\rightarrow (B$ **says** $(\mathbf{goal}(U,N)) \rightarrow A$ **says** $(\mathbf{goal}(U,N)))\big)$	SAYS-I2 5
7	A **says** $(\forall U' \forall N'.B$ **says** $(\mathbf{goal}(U',N')) \rightarrow A$ **says** $(\mathbf{goal}(U',N')))$	$\overset{\text{def}}{=}$ e 1
8	A **says** $(B$ **says** $(\mathbf{goal}(U,N)) \rightarrow A$ **says** $(\mathbf{goal}(U,N)))$	SAYS-I3 6, 7
9	A **says** $(B$ **says** $(\mathbf{goal}(U,N)))$	SAYS-I2 2
10	A **says** $(A$ **says** $(\mathbf{goal}(U,N)))$	SAYS-I3 8, 9
11	A **says** $(\mathbf{goal}(U,N))$	SAYS-TAUT 10

To allow delegation to local names like $B.S$, we have to formulate another version of the **speaksfor** operator. The formulation follows directly from the definitions of

speaksfor and our abbreviation for local names.

$$\begin{aligned} A\,\textbf{speaksfor}'\,B.S \overset{\text{def}}{=} {} & \forall U \forall N.(A\,\textbf{says}\,(\textbf{goal}(U,N))) \\ & \rightarrow (B\,\textbf{says}\,((\textbf{name}\,S)\,\textbf{says}\,(\textbf{goal}(U,N)))) \end{aligned}$$

We modify SPEAKSFOR-E2 to take into account the alternative delegation operator, **speaksfor′**, and then we prove the rule as a theorem. The proof is analogous to the proof of SPEAKSFOR-E.

$$\frac{A\,\textbf{says}\,(B\,\textbf{speaksfor}'\,A.S) \qquad B\,\textbf{says}\,(\textbf{goal}(URL, nonce))}{A.S\,\textbf{says}\,(\textbf{goal}(URL, nonce))} \qquad (\text{SPEAKSFOR-E2})$$

3.2.4 Specific Delegation

The only difference between specific and general delegation is that specific delegation shares authority regarding only a particular URL, rather than all URLs. The **delegate** operator, therefore, is very similar to the **speaksfor** operator; it takes the additional URL as a parameter, and its definition does not quantify over all URLs.

$$\textbf{delegate}\,(A,\ B,\ U) \overset{\text{def}}{=} \forall N.(B\,\textbf{says}\,(\textbf{goal}(U,N))) \rightarrow (A\,\textbf{says}\,(\textbf{goal}(U,N)))$$

An analogous operator is used to delegate authority to local principals.

$$\begin{aligned} \textbf{delegate}'\,(A,\ B.S,\ U) \overset{\text{def}}{=} {} & \forall N.(B\,\textbf{says}\,((\textbf{name}\,S)\,\textbf{says}\,(\textbf{goal}(U,N)))) \\ & \rightarrow (A\,\textbf{says}\,(\textbf{goal}(U,N))) \end{aligned}$$

3.2.5 Time

The **after** operator is similar to the delegation operators: if a principal believes a formula constructed with such an operator, and if some external condition is true, then we can conclude that the principal has certain other beliefs. This suggests that the **after** operator can be defined as an implication. In the case of delegation the external condition is the belief of the recipient of the delegation, whereas in the case of the **after** operator it is a judgment about the current time.

$$\textbf{after}\ (N,\ F) \overset{\text{def}}{=} (\textbf{localtime} > N) \to F$$

The PCA logic describes the current time with the constant **localtime**, which makes it possible to replace the magical side condition on the AFTER-E rule with an additional premise.

$$\frac{A\ \textbf{says}\ (\textbf{after}\ (N,\ F)) \qquad \textbf{localtime} > N}{A\ \textbf{says}\ F} \quad (\text{AFTER-E})$$

The proof is straightforward; the key, again, is to use SAYS-I3 (modus ponens inside **says**) to discharge the proof obligation in the definition of **after**.

3.3 Soundness

As discussed in Section 2.5, before we decide to rely on a particular security logic, we would like to have certain guarantees about its behavior.

One desired behavior is consistency: it should not be possible to prove both some formula and its negation. If it were possible, one could prove $\perp$, and $\perp$ could be used to prove any arbitrary formula. A logic in which one can prove arbitrary formulas is not of much use, so we certainly want our application-specific logic to be consistent. A related, but strictly more powerful, property is soundness. In a sound logic it is impossible to erroneously prove any formula that is not supported by the initial facts. A sound logic is necessarily consistent, so a proof of soundness also guarantees consistency.

For any particular application-specific logic, we might like to have even stronger guarantees. It would be useful to prove, for example, that principals' beliefs obey a property akin to consistency. We would be more inclined to trust our logic if we knew, for example, that if the Registrar signed only certain kinds of statements then his beliefs would never include $\bot$. Any such property would require the overall logic to be sound, or at least consistent; in this section we address these more general guarantees.

The discussion of soundness in Section 2.5 was brief and highly informal, since soundness is proven with respect to a model, and in that chapter we presented no model for our logic. In this chapter, we have provided the logic with a semantics—each of the operators and rules of our application-specific logic has a definition in the PCA logic. Any set of formulas that has a model in higher-order logic is sound [4, Thm. 5402]; if the application-specific logic were embedded directly in higher-order logic, this theorem would be sufficient to prove the soundness of our application-specific logic. The PCA logic differs somewhat from the particular formalization of higher-order logic to which the theorem refers [4, Chapter 5], but we will argue that the PCA logic itself can be based on the same formalization. We address each significant difference in turn.

- The sets of operators and inference rules of the PCA logic are not the same as those of higher-order logic. However, each of the operators of the PCA logic can be defined from the operators of higher-order logic. Similarly, each of the inference rules can be derived from the axioms and inference rule of higher-order logic. For example, the PCA logic has an operator for existential quantification (∃), whereas higher-order logic has universal quantification (∀). Existential quantification, however, can be defined as an abbreviation: $\exists x.A \overset{\text{def}}{=} \neg(\forall x.\neg A)$.

- The PCA logic has the type of principals. This type is defined, however, as a string, and strings can be modeled by the natural numbers in a conventional presentation of higher-order logic.

- Constructors like **signed** and **name** have no analogue in higher-order logic. Higher-order logic permits arbitrary constants of various types, fortunately, so the addition of our constants merely gives results to a particular formulation of higher-order logic [4, p. 162].

We have established that the PCA logic can be expressed in a particular formulation of higher-order logic. This is sufficient to show that the set of definitions and theorems that corresponds to our application-specific logic is sound.

Proving access using our application-specific logic introduces one more variable which could potentially lead to unsoundness: each proof of access is based on a set of premises (of the form A **signed** F) which are temporarily added to the logic as postulates; if the premises themselves make the higher-order logic inconsistent, we cannot guarantee anything about the application-specific logic. For example, if both F and $\neg F$ are added, one could derive $\perp$, and therefore prove *Bob* **says goal**$(\ldots)$ even if Bob does not wish to grant access.

Fortunately, in our case the premises cannot lead to such a situation. More formally, the property we would like to hold is that if we derive a formula ϕ in the logic expanded with our premises, and the **signed** constructor does not occur in ϕ, then ϕ is also derivable in the unexpanded logic. The **signed** constructor is specific to our formulation of higher-order logic, so it does not appear in any standard axioms or inference rules; the standard axioms and inference rules could potentially affect only the parameters, A and F. To convert a proof of ϕ derived in the enhanced logic to a proof in the original logic, we replace all the instances of the **signed** constructor in the derivation of the proof with $\lambda A \lambda F.\top$. All the premises are similarly replaced and evaluate to $\top$, leaving a derivation in the original logic, with no mention of the **signed** operator. Since no inference rules of higher-order logic mention **signed**, any inference rules that were used in the proof with **signed** will still work in the proof with $\lambda A \lambda F.\top$. The property described above hence holds; therefore, adding premises of the form A **signed** F does not interfere with the soundness of the original logic [6].

Chapter 4

System Implementation

To make use of our access-control logic we have developed a system with automated provers, proof checkers, and a protocol that allows hosts to transmit to each other statements made in our logic. The system we have built consists of clients, servers, and fact servers. Clients (e.g., Alice) attempt to access web pages and have to construct proofs; servers (e.g., Bob) guard access to the web pages published on them; and fact servers (e.g., the Registrar) publish and, optionally, protect pieces of the security policy. The functionality required of fact servers is a subset of what is required of ordinary servers, so this chapter will not discuss them in detail.

One of the design requirements of our system was that it be convenient to retrofit onto existing web infrastructure. This dictates the architecture of our system—we provide extensions to standard web browsers and standard web servers which enable them to carry out the PCA protocol. Figure 4.1 shows the architecture of our system.

The bulk of the client part is a web browser. The rest—the proxy server and the prover—are components that enable the web browser to use the PCA protocol. The

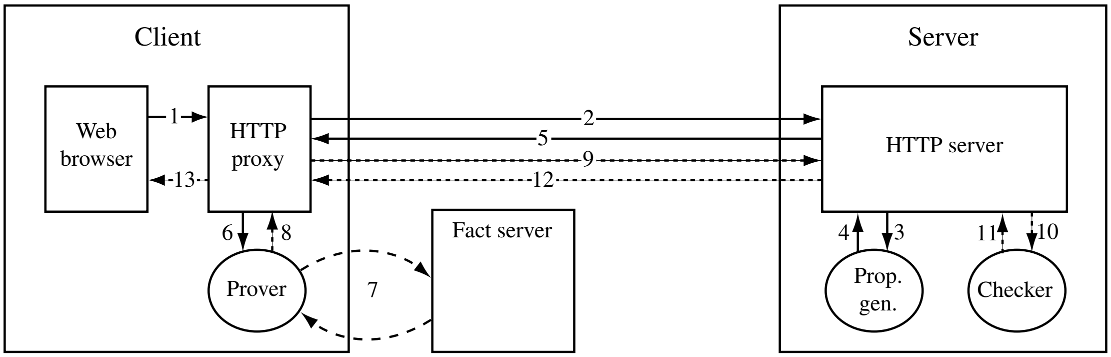

Figure 4.1: The components of our system.

browser itself remains unmodified, and our system does not use any features that are unique to a particular browser version.

The server part of our PCA system is built around an unmodified web server. The web server is PCA-enabled through the use of a servlet which intercepts and handles all PCA-related requests. The two basic tasks that take place on the server's side during a PCA transaction are generating the proposition that needs to be proved and verifying that the proof provided by the client is correct. Each is performed by a separate component, the proposition generator and the checker, respectively.

This chapter will describe in detail the individual parts of the client and the server. An extended example will illustrate how they are all used during the course of a PCA transaction.

4.1 Client

To access a PCA-protected web page, the client will have to generate proofs of one or more challenges provided by the server.

4.1.1 Proxy Server

The job of the proxy server is to be the intermediary between a web browser that has no knowledge of the PCA protocol and a web server that is PCA-enabled. An attempt by the browser to access a PCA-protected web page results in a dialogue between the proxy and the server that houses the page. The dialogue is conducted through PCA-enhanced HTTP, that is, HTTP augmented with headers that allow it to convey information needed for authorization using the PCA protocol. The browser is completely unaware of this dialogue; it sees only the web page returned at the end.

The proxy is meant to be a substitute for a browser plugin. We decided to use a proxy instead of a plugin because this lets our system be completely browser independent. A production implementation would probably replace the proxy with a plugin. Like a plugin, our proxy is meant to be tightly coupled with the web browser. Unlike traditional web proxies, it is meant to serve a single client, not a set of them. This is because the proxy needs to speak on behalf of a client, perhaps signing statements with the client's private key or identifying itself with the client's public key. If a shared proxy were to be used for this purpose, its ability to access the private information of several clients would be a concern. Also, it would have to authenticate client-side connections so that it would know which client's data, or identity, to use for PCA transactions. Such authentication would be at cross purposes with one of the goals of our system—authorization uncoupled from authentication.

Using a local, single-user proxy does not solve the problem of authenticating the user to the proxy. On a well-configured, single-user personal computer, however, logging in to the computer itself is sufficient to guarantee that only the client whose private key the proxy holds will be able to use the proxy.

4.1.2 Secure Transmission and Session Identifiers

In a PCA system a client obtains access to a resource by presenting a proof—the proof acts as a capability that allows access to the resource. It's important, therefore, to guard the proof against misuse, both by an attacker who intercepts it and a malicious client who might try to reuse a old proof. To that end, the server requires that each proof goal contains a session identifier, or nonce. In addition, proofs and nonces are transmitted only over a secure channel.

The session identifier is a server-generated secret shared by the client and server. A single authorization transaction might require the client to generate multiple proofs. The session identifier is used in challenges and proofs (including in digitally signed formulas within the proofs) to make them specific to a single authorization session. The session identifier not only allows a server to require a client to generate a fresh proof, but also allows the server to cache proofs and to let clients present the session identifier as a token that demonstrates that they have already provided the server with a proof of access. Since they may be time-dependent, the server will still verify that the premises of a cached proof are valid.

Since the session identifier may be sufficient to gain access to a resource, stealing a session identifier, akin to stealing a proof in a system where goals are not tagged with nonces, compromises the security of the system. In order to keep the session identifier secret, communication between the client and server uses the secure protocol HTTPS instead of normal HTTP in all cases where a session identifier is sent. If the client attempts to make a standard HTTP request for a PCA-protected page, the server replies with a special "Authorization Required" message which directs the client to switch to HTTPS and retry the request.

4.1.3 Prover

In the course of a PCA conversation with a server, the proxy needs to generate proofs that will demonstrate to the server that the client should be allowed access to a particular file. This task is independent enough from the rest of the authorization process that it is convenient to abstract it into a separate component. During a PCA conversation the client may need to prove multiple statements; the process of proving each is left to the prover.

The core of the prover in our system is the Twelf logical framework [45]. Proofs are generated automatically by a logic program that uses tactics. The goal that must be proven is encoded as the statement of a theorem. Facts that are likely to be helpful in proving the theorem are added as assumptions. The logic program generates a derivation of the theorem; this is the “proof” that the proxy sends to the server.

The tactics that define the prover (see Appendix A.2) roughly correspond to the inference rules of the application-specific logic. Together with the algorithm that uses them, the tactics comprise a decision procedure that generates proofs. As described in Section 2.6, the algorithm implemented by the prover is a depth-first search without marking.

As part of generating the proof of a goal given to it by the proxy, the prover’s job is to find all the assumptions that are required by the proof. Assumptions needed to generate a proof might include statements made by the server about who is allowed to access a particular file, guesses about time, statements by which principals delegate authority to other principals, or statements of goal. While some of these might be known to the proxy, and would therefore have been provided to the prover, others might need to be obtained from web pages.

4.1.4 Gathering Facts and Iterative Proving

As part of generating the proof of a goal given to it by the proxy, the prover's job is to find all the assumptions that are required by the proof.

The client may not always be able to generate a proof of a challenge on its first try. It may need to obtain additional information, such as signed delegations or other facts, before the proof can be completed. The process of fetching additional information and then retrying the proof process is called *iterative proving*. The process does not affect the server, and terminates when a proof is successfully generated.

Proof generation can be divided into two phases. In the first phase, facts are gathered. In the second phase, a prover attempts to assemble the facts into a proof of the challenge. If it is successful, the proof is returned. Otherwise, the phases are repeated, first gathering additional facts and then reproving, until either a proof is successfully generated, or until no new facts can be found.

For the purpose of specifying the order in which they are gathered, we group facts into four general categories.

Goal-oriented facts are those that describe who has access to a particular URL—this is the piece of the security policy that typically resides on the server. Goal-oriented facts represent the last delegation in the delegation chain that gives a client access to a resource, so if a client is looking to accumulate facts, this is the first sort of fact that the client gathers.

Key-oriented facts describe delegation statements made by principals other than the holder of the resource. We call them key-oriented because a client learns them by seeking information about public keys that might be mentioned in goal-oriented facts. Key-oriented facts are the second type of fact to be gathered.

Self-signed facts are the assertions made by a client that access should be granted. Each proof contains one self-signed assertion. If the server delegates authority directly to the client, then a goal-oriented and a self-signed fact could be sufficient to generate a proof. In the hope that it might be able to generating a proof from facts that it has already learned, a client creates a self-signed fact immediately after receiving a challenge.

Assertions about time are the last kind of fact. To demonstrate that a time-dependent delegation statement holds, the client makes an assertion about the current time on the server's machine. These assertions are made whenever the client encounters a time-dependent delegation, and they are verified by the server before the proof is accepted.

Since fetching facts from the web is a relatively time-consuming process (hundreds of milliseconds is a long time for a single step of an interactive authorization that should be transparent to the user), the prover caches the facts for future use. The prover also periodically discards facts which have not been recently used in successful proofs.

4.1.5 Client Control Flow

The proxy's behavior while trying to prove access to a PCA-protected web page is summarized in Figure 4.2. We assume that the proxy has is communicating with the server over a secure channel (HTTPS). The proxy starts out by requesting the URL from the server. If the server's answer does not include a challenge, the proxy's job is done; this is either because the server has returned the desired web page, or because an error occurred on the server.

If the server's answer includes a challenge, the proxy must check whether this same challenge has been proved before. Receiving a challenge that it has already proven most likely indicates that the proxy's proof was not accepted. This should normally happen only if the proxy's guesses about current time were incorrect; the solution, then, is to

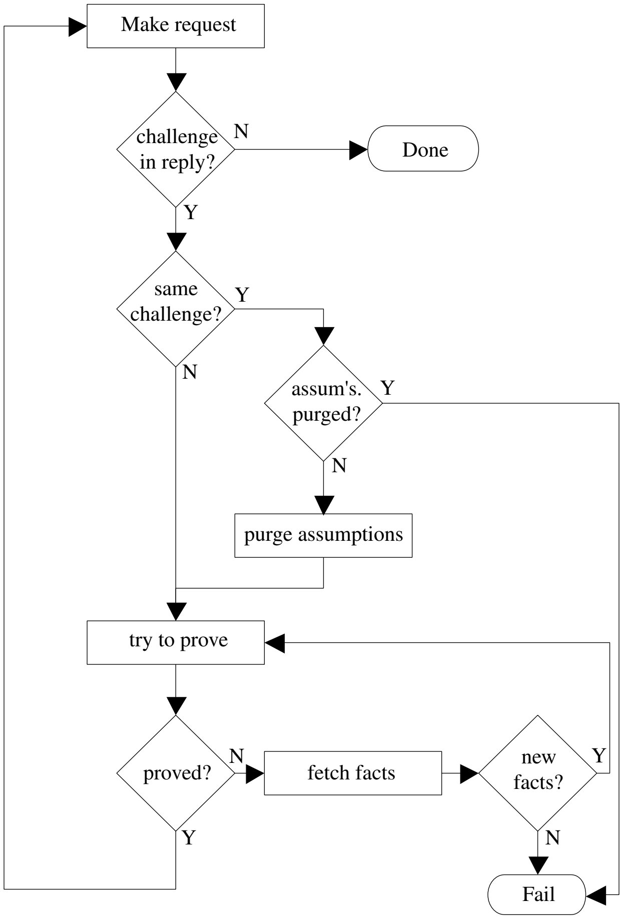

Figure 4.2: Client flowchart.

discard those guesses and attempt to generate the proof anew. If the server continues to respond with the same challenge, the proxy has no recourse but to give up.

When the proxy receives a new challenge, it enters the iterative proving loop, during which it repeatedly fetches facts and tries to construct a proof of the challenge. Iterative proving stops when the challenge has been proven or when the proxy is unable to fetch any new facts.

4.2 Server

4.2.1 Proposition Generator and Iterative Authorization

When a client attempts to access a PCA-protected web page, the server replies with a statement of the theorem that it wants the client to prove before granting it access. This statement, or proposition, depends only on the pathname of the file that the client is trying to access and on the syntax of the logic in which it is to be encoded; it is generated by the server's *proposition generator*, a module independent from the rest of the server.

The proposition generator provides the server with a list of propositions. The server returns to the client the first unproven proposition. If the client successfully proves that proposition in a subsequent request, then the server will reply with the next unproven proposition as the challenge. This process of proving and then receiving the next challenge from a list of unproven propositions is called *iterative authorization*, and is illustrated in Figure 4.3.

The process of iterative authorization terminates when either the client gives up (i.e., cannot prove one of the propositions) or has successfully proven all of the propositions, in which case access is allowed. If the client presents a proof which fails when the server checks it, it is simply discarded. In this case, the same challenge will be returned to the client twice.

Our system generates a proposition for each directory level of the URL specified in the client's request. This ensures that the client has permission to access the full path (as in the standard access control for a hierarchical file system). Since the server returns identical challenges regardless of whether the requested object exists, returning a challenge reveals no information about the existence of objects on the server.

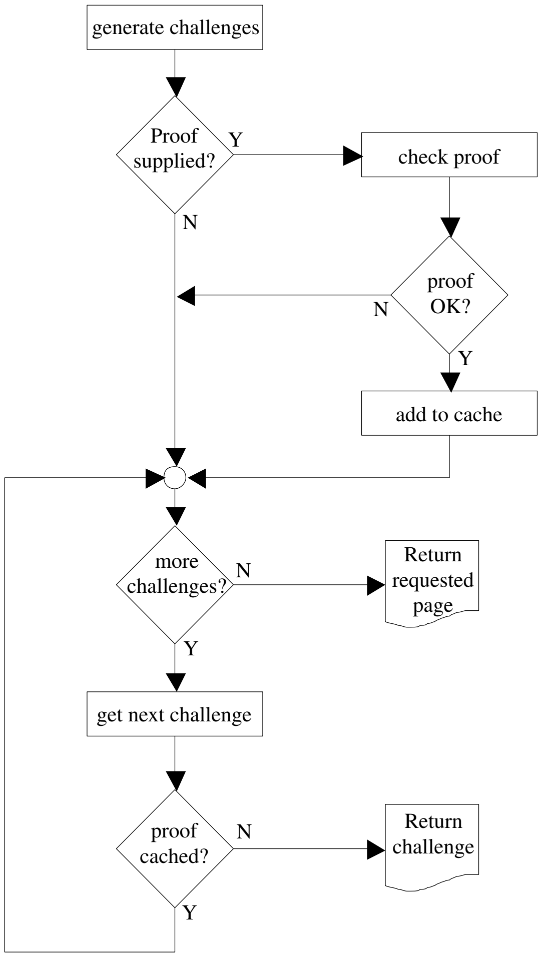

Figure 4.3: Server flowchart.

Isolating the proposition generator from the rest of the server makes it easy to adapt the server for other applications of PCA (protecting a file system, for example, or web-based resources that do not have a hierarchical naming scheme); using it for another application may require nothing more than changing the proposition generator.

A benefit of iterative authorization is that it allows parts of the security policy to be hidden from unauthorized clients. Only when a challenge has been proven will the client be able to access the facts that it needs to prove the next challenge. In the context of our application this means, for example, that a client must prove that it is allowed to access a directory before it can even find out what goal it must prove (and therefore what facts it must gather) to gain access to a particular file in that directory.

4.2.2 Transmitting Challenges and Proofs

For each authorization request, the server's proposition generator generates a list of propositions which must be proven before access is granted. Each proposition contains a URL and a session identifier. The server checks each proposition to see if it was previously proven by the client by checking a cache of previously proven challenges. If all of the propositions have been proven, access is allowed immediately. Otherwise, the first unproven proposition is returned to the client as a challenge. Any other unproven propositions are discarded.

The server constructs a reply with a status code of "Unauthorized." This is a standard HTTP response code (401) [26]. The response includes the required HTTP header field "WWW-Authenticate" with an authentication scheme of "PCA" and the unproven proposition as its single parameter.

Once the client has constructed a proof of the challenge, it makes another HTTPS request (this can be done with the same TCP connection if allowed by keep-alive) containing the challenge and the proof. The challenge is included in an "Authorization" request-header field, and the proof is included in a series of "X-PCA-Proof" request-header fields. The server checks that the proof proves the supplied challenge, adds the challenge to its cache of proven propositions, and then begins the checking process for the next proposition.

4.2.3 Proof Checking

The Theory

After it learns which proposition it must prove, the client generates a proof and sends it to the server. If the proof is correct, the server allows the client to access the requested web page. Proofs are checked using Twelf. The proof provided by the client is encoded as an LF term [31]. The type (in the programming languages sense) of the term is the statement of the theorem that must be proven; the body of the term is the proof itself. Checking that the derivation is correct amounts to type checking the term that represents the proof. If the term is well typed, the client has succeeded in proving the proposition.

As is the case for building the proof, using Twelf for proof checking is overkill, since only the type-checking algorithm is used. The proof checker is part of the trusted computing base of the system. To minimize the likelihood that it contains bugs that could compromise security, it should be as small and simple as possible. Several minimal LF type checkers have been implemented [9, 43]; one of these could serve as the proof checker for our system.

The natural way to efficiently represent LF proofs is in the form of directed acyclic graphs (DAGs). This makes it easy to reuse the many shared structures, particularly type information, that occur in the proof. Twelf accepts as input and provides output in human-readable ASCII, so instead of representing proofs as DAGs we have to represent them in a more linear fashion, as trees, with all the shared structures replicated wherever they are used. The replication causes an exponential increase in the size of our proofs. Relying on Twelf's implicit type reconstruction algorithm would allow us to compress the proofs to a manageable size. Unfortunately the algorithm is undecidable, which could cause correct proofs not to be accepted or the server to be tied up by a complicated proof. The best solution would be to represent our proofs in the more efficient, machine-friendly format and use one of the previously mentioned minimal checkers. For now, however, we will use the inefficient, explicitly typed form; our proofs are sufficiently small to remain tractable despite the blowup in size.

The Practice

Verifying that proofs are valid is slightly more involved than just checking that Twelf will accept them. Twelf allows the user to input both axioms and theorems. If we allowed a client's proof to contain axioms, he could simply assert a proposition instead of proving it. In addition, the server must verify any digital signatures and guesses about time that are sent with the proof.

We developed a preprocessor that resolves these issues before the proof is sent to Twelf. The preprocessor first makes sure that all of the terms that make up the proof have both a type and a definition; a proof that contains illegal axioms is rejected.

Next, two special types of axioms are inserted into the proof as necessary. The first type is used to make propositions about digital signatures, and the second type is used to make propositions regarding time. These are required since an LF type checker cannot itself check digital signatures or assertions about time. The client inserts into the proof placeholders for the two types of axioms. The server makes sure that each axiom holds, generates an LF declaration that represents it, and then replaces the placeholder with a reference to the declaration.

For digital signatures, the client inserts into the proof a proposition of the special form "#signature *key, formula, sig*". The server checks that *sig* is a valid signature made by the key *key* for the formula *formula*. If so, the #signature statement is replaced by an axiom asserting that *key* signed *formula*.

To make statements about time, the client inserts a proposition of the special form "#localtime $\{<,>\}$ *value*". The preprocessing stage verifies whether the assertion about time is correct. If it is, the preprocessor produces an axiom that describes the current time as being before or after N and replaces the #localtime statement with a reference to that axiom. If the assertion is not correct, the proof is rejected.

Once the proof has been parsed to make sure it contains no axioms and special axioms of these two forms have been reintroduced, the preprocessor verifies that the proof is attempting to prove the correct challenge. (It might be a perfectly valid proof of some other challenge!) If this final check succeeds, then the whole proof is passed to an LF type checker; in our case, this is again Twelf.

If all of these checks succeed and the proof is valid then the challenge is inserted into the server's cache of proven propositions. The server will either allow access to the page (if this was the last challenge in the server's list) or return the next challenge to the client.

4.3 A Sample Transaction

To illustrate how the various parts of the client and server work together to perform a PCA transaction, we will describe in detail what happens when Alice tries to access the midterm results page.

First, Alice types the URL $\texttt{http}:\texttt{//server/midterm.html}$ into her browser. Her browser forms the request and sends it to the local proxy (Figure 4.1, step 1). The proxy server does not yet know that the page Alice has requested is PCA protected, so it forwards the request to Bob without modifying it (step 2). Alice's browser will remain completely unaware of the proxy's actions until the proxy sends the browser either the page that Alice attempted to load or an error message explaining why the page couldn't be loaded.

Bob receives Alice's request. Since the request is for a PCA-protected web page, the request is handled by the servlet, which will handle the server's end of the remainder of the PCA transaction. The servlet notices that the request has been made over HTTP and rejects it with a hint that Alice should try again via HTTPS. Alice's proxy switches to HTTPS and sends the same request again.

After receiving the second, encrypted request, the servlet first generates the session ID, SID. It then passes the request and the ID to the proposition generator (step 3). The proposition generator returns a list of propositions that Alice must prove before she is allowed to see /midterm.html (step 4).

$$(\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}))\ \textbf{says}\ (\textbf{goal}(\texttt{http}://\texttt{server}/, \texttt{SID}))$$
$$(\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}))\ \textbf{says}\ (\textbf{goal}(\texttt{http}://\texttt{server}/\texttt{midterm.html}, \texttt{SID}))$$

The first proposition states that the server believes that it's OK for the session identifier SID to read $\texttt{http}://\texttt{server}/$. The servlet checks whether this proposition has already been proved—it has not—and then constructs an HTTP response that includes this proposition as a challenge and sends it to Alice (Figure 4.1, step 5).

The proxy receives Bob's message and extracts the challenge. Knowing that any proof of the challenge must include Alice's assertion that it is OK to access the URL in question, the proxy generates the corresponding fact.

$$(\textbf{name}(\texttt{pubkey}_{\texttt{Alice}}))\ \textbf{signed}\ (\textbf{goal}(\texttt{http}://\texttt{server}/, \texttt{SID}))$$

Armed with these facts, Alice's proxy uses the prover to try to prove the challenge (step 6). The attempt fails, since Alice's assertion is by itself insufficient to support a proof. Alice's proxy will now iteratively fetch additional facts until a proof can be found (step 7).

The proxy first attempts to fetch from Bob the goal-oriented facts about $\texttt{http}://\texttt{server}/$. The facts about each URL are housed on Bob's machine in a location determined by the name of the URL, so Alice knows where to find them.

Bob receives Alice's request for goal-oriented facts. These facts, like the midterm results web page, are PCA protected. To bootstrap the authorization process, however, the goal-oriented facts that describe who has access to the root directory of the server are freely available.

In addition to the midterm results page, Bob is in the habit of making other class materials available on his server. The root directory on his machine, therefore, is always accessible to any of his students. Bob therefore returns to Alice the following fact:

$$\begin{aligned}
&(\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}))\ \textbf{signed}\\
&\qquad\qquad(\textbf{delegate}\ (\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}),\\
&\qquad\qquad\qquad\qquad(\textbf{name}(\texttt{pubkey}_{\texttt{Registrar}})).\texttt{CS101},\\
&\qquad\qquad\qquad\qquad\texttt{http}://\texttt{server}/))
\end{aligned}$$

Armed with these facts, Alice's proxy uses the prover to try to prove the challenge. The attempt fails, since the facts are insufficient to support a proof. Alice's proxy will now iteratively fetch additional facts until a proof can be found. To learn where to look for additional facts, Alice examines the facts she has already obtained.

Bob's public key and the Registrar's public key are embedded in the facts Alice has collected. In each key is encoded a URL that describes a location at which the owner of that key publishes additional facts. The Registrar's key, heretofore given as $\mathtt{pubkey}_{\mathtt{Registrar}}$ actually has the form $\mathtt{pubkey}_{\mathtt{Registrar}};\mathbf{http}://\mathbf{server}/\mathbf{facts}/$.

Alice visits these URLs in the hope of finding useful facts. From the URL mentioned in the Registrar's public key she learns the following fact:

$$(\mathbf{name}(\mathtt{pubkey}_{\mathtt{Registrar}}))\ \mathbf{signed}\\ ((\mathbf{name}(\mathtt{pubkey}_{\mathtt{Alice}}))\ \mathbf{speaksfor}\ ((\mathbf{name}(\mathtt{pubkey}_{\mathtt{Registrar}})).\mathtt{CS101}))$$

After fetching this fact, Alice again attempts to generate a proof of the first assumption. She now has enough facts to demonstrate that she is authorized to read $\mathtt{http}://\mathtt{server}/$, so she succeeds in constructing the proof. The proxy repeats its request to access $\mathtt{http}://\mathtt{server}/\mathtt{midterm.html}$, this time including in the request the proof that was just generated (step 9).

Upon receiving Alice's request, the servlet again uses the proposition generator to determine which propositions must be satisfied before access is granted. Noticing that Alice's request carries a proof of the first proposition, the servlet send the proof to the checker (step 10). The checker verifies that Alice's proof of the first proposition is valid. The servlet checks its cache to determine whether the second proposition has been proved. It has not, so the servlet again rejects Alice's request, this time including in its response an encoding of the second challenge.

Alice now repeats the process that she used to generate the first proof, with minor variations because she already knows some facts that will be useful in constructing the proof (steps 6–12).

As before, Alice's proxy first creates a self-signed fact.

$$(\mathbf{name}(\mathtt{pubkey}_{\mathtt{Alice}}))\ \mathbf{signed}\ (\mathbf{goal}(\mathtt{http}://\mathtt{server}/\mathtt{midterm.html},\mathtt{SID}))$$

Alice now tries to use this fact, together with the facts that have been gathered during the attempt to generate the first proof, to prove the second challenge. The attempt fails, since Alice has no goal-oriented fact that describes who is allowed to access the midterm results.

Exactly as for the previous challenge, Alice requests from Bob a goal-oriented fact about $\texttt{http://server/midterm.html}$. Before sending her the fact, Bob verifies that Alice had previously proved that she is allowed to access the server's root directory, since only people who can access the root directory are allowed access to the goal-oriented facts about files in the root directory.

Like the root directory, the midterm results file is accessible only by students taking CS101; in addition, no one may access it before 8 P.M.

$$\begin{aligned}
&(\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}))\ \textbf{signed}\\
&\qquad\qquad(\textbf{after}\ (8 P.M., (\textbf{delegate}\ (\textbf{name}(\texttt{pubkey}_{\texttt{Bob}}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad(\textbf{name}(\texttt{pubkey}_{\texttt{Registrar}})).\texttt{CS101},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\texttt{http://server/midterm.html}))))
\end{aligned}$$

Parsing the goal-orient fact, Alice also learns that Bob's delegation is time dependent. Knowing that it's currently 9 P.M. according to her own clock, Alice assumes that Bob's clock shows a time after 8 P.M. She asserts the formula $\textbf{localtime} > 8P.M.$ and adds it to her list of facts.

As before, at this point Alice attempts to generate a proof. Unlike for the first challenge, however , proving succeeds, since Alice has in her cache the Registrar's delegation statement. Alice makes a final request to access $\texttt{http://server/midterm.html}$, this time including in it the proof of the second challenge.

The server receives Alice's request for `midterm.html` and generates a list of propositions that need to be proved before access is granted. The first proposition has been proved, and Alice has included a proof of the second one; the server confirms that her proof is valid. There are no more propositions left to be proved, so Alice has successfully proved that she is authorized to read `http://server/midterm.html`. The server sends the requested page to Alice. Alice's proxy recognizes that the proving process is complete, and returns the page to Alice's browser.

4.4 Performance and Optimizations

4.4.1 Caching and Modularity

Our authorization protocol involves a number of potentially lengthy operations like transferring data over the network and verifying proofs. We use caching on both the client and the server to alleviate the performance penalty of these operations.

Client-side One of the inevitable side-effects of a security policy that is distributed across multiple hosts is that a client has to communicate with each of them. Delegation statements in the security policy may force this communication to happen sequentially, since a client might fetch one piece of data only to discover that it needs another. While there is little than can be done to improve the worst-case scenario of a series of sequential fetches over the network, subsequent fetches of the same facts can be eliminated by caching them on the client. Some facts that reside in the cache may expire; but since it is easy for the client to check whether they are valid, they can be checked and removed from the cache out-of-band from the proof-generation process.

Server-side To avoid rechecking proofs, all correctly proven propositions are cached. Some of them may use time-dependent or otherwise expirable premises—they could be correct when first checked but false later. If such proofs, instead of being retransmitted and rechecked, are found in the cache, their premises must still be checked before authorization is accepted. The proofs are kept cached as long as the session ID with which they are associated is kept alive.

Since all proofs are based on a sparse and basic core logic, they are likely to need many lemmas and definitions for expressing proofs in a concise way. Many clients will use these same lemmas in their proofs; most proofs, in fact, are likely to include the same basic set of lemmas. We have added to the proof language a simple module system that allows us to abstract these lemmas from individual proofs. Instead of having to include all the lemmas in each proof, the module system allows them to be imported with a statement like `basiclem = #include http://server/lemmas.elf`. If the lemma `speaksfor_e1`, for example, resides in the `basiclem` module, it can now be referenced from the body of the proof as `basiclem.speaksfor_e`. Instead of being managed individually by each client, abstracting the lemmas into modules allows them to be maintained and published by a third party. A company, for instance, can maintain a single set of lemmas that all its employees can import when trying to prove that they are allowed to access their payroll records.

To make the examples in the previous section more understandable, we have omitted from them references to modules. In reality, each proof sent by a client to a server would be prefixed by a `#include` statement for a module that contained the definitions of, for example, `says`, `speaksfor`, `delegate` and the lemmas that manipulate them, as well as more basic lemmas.

Aside from the administrative advantages, an important practical benefit of abstracting lemmas into modules is increased efficiency, both in bandwidth consumed during proof transmission and in resources expended for proof checking. Instead of transmitting with each proof several thousands of lines of lemmas, a client merely inserts a `#include` declaration which tells the checker the URL at which the module containing the lemmas can be found. Before the proof is transmitted from the client to the server, the label under which the module is imported is modified so that it contains the hash of the semantic content (that is, a hash that is somewhat independent of variable names and formatting) of the imported module. This way the checker knows not only where to find the module, but can also verify that the prover and the checker agree on its contents.

When the checker is processing a proof and encounters a `#include` statement, it first checks whether a module with that URL has already been imported. If it has been, and the hash of the previously imported module matches the hash in the proof, then proof checking continues normally and the proof can readily reference lemmas declared in the imported module. If the hashes do not match or the module has not been imported, the checker accesses the URL and fetches the module. A module being imported is validated by the checker in the same way that a proof would be. Since they are identified with content hashes, multiple versions of a module with the same URL can coexist in the checker's cache.

The checker takes appropriate precautions to guard itself against proofs that may contain modules that endlessly import other modules, cyclical import statements, and other similar attacks.

4.4.2 Speculative Proving

In our running example the web proxy waited for the server's challenge before it began the process of constructing a proof. In practice, our proxy keeps track of visited web pages that have been protected using PCA. Based on this log, the proxy tries to guess, even before it sends out any data, whether the page that the user is trying to access is PCA protected, and if it is, what the server's challenge is likely to be. In that case, it can try to prove the challenge even before the server makes it (we call this *prove-ahead* or speculative proving). The proof can then be sent to the server as part of the original request. If the client guessed correctly, the server will accept the proof without first sending a challenge to the client. If the web proxy already has all the facts necessary for constructing a proof, this will reduce the amount of communication on the network to a single round trip from the client to the server. This single round trip is necessary in any case, just to fetch the requested web page; in other words, the proof is piggybacked on top of the fetch message.

4.4.3 Performance Numbers

protocol stage	*ms*
fetch URL attempt without HTTPS	198
fetch URL attempt with no proof	723
failed proof attempt	184
fetch file fact + failed proof attempt	216
fetch key fact + successful proof attempt	497
fetch URL attempt (empty server cache)	592
failed proof attempt	184
fetch file fact + successful proof attempt	295
fetch URL attempt (server cached module)	330
total	3219

Figure 4.4: Worst-case performance.

protocol stage	*ms*
fetch URL attempt with no proof	180
failed proof attempt	184
fetch file fact + successful proof attempt	295
fetch URL attempt (server cached module)	330
total	989

Figure 4.5: Typical performance.

protocol stage	*ms*
construct proof from cached facts	270
fetch URL attempt (server cached module)	330
total	600

Figure 4.6: Fully-cached performance.

protocol stage	*ms*
fetch URL attempt (already authorized)	175
total	175

Figure 4.7: Performance with valid session ID.

As one might expect, the performance of our system varies greatly depending on how much information is cached by the proxy and by the server. The relevant metric is the amount of time it takes to fetch a protected web page. We evaluated our system using the example of Alice trying to access `midterm.html` (see Figures 4.4–4.7). For comparison, Figure 4.8 shows the length of time to fetch a page that is not protected. The actual example from which we obtained the performance data was based on a slightly different logic that did not include facts about time.

The slowest scenario, detailed in Figure 4.4, is when all the caches are empty and the first attempt to fetch the protected page incurs initialization overhead on the server (this is why the first attempt to fetch the URL takes so long even though a proof is not included).

protocol stage	*ms*
fetch URL attempt (page not protected)	50
total	50

Figure 4.8: Access control turned off.

In this case, it takes 3.2 seconds for the proxy to fetch the necessary facts, construct a proof, and fetch the desired page.

A more typical situation is that a user attempts to access a protected page on a previously visited site (Figure 4.5). In this case, the user is already likely to have proven to the server that she is allowed access to the server and the directory, and must prove only that she is also allowed to access the requested page. In this case she probably needs to fetch only a single (file or goal) fact, and the whole process takes 1 second. Speculative proving would likely eliminate the overhead of an attempted fetch of a protected page without a proof, saving about .2 seconds. If the client already knows the file fact (Figure 4.6), that length of the access is cut to about .6 seconds.

When a user wants to access a page that she has already accessed and the session identifier used during the previous, successful attempt is still valid, access is granted based on just the possession of the identifier—this takes about 175 milliseconds.

Alice's proof might have to be more complicated than in our example; it could, for example, contain a chain of delegations. For each link of the chain Alice would first have to discover that she could not construct the proof, then she would have to fetch the relevant fact and attempt to construct the proof again—which in our system would currently take about .6 seconds.

The performance results show that, even when all the facts are assembled, generating proofs is slow (at least 200 ms) and grows slower as the user learns more facts. While this is a fundamental bottleneck, the performance of our prover is over an order of magnitude slower than it need be. For comparison purposes, we implemented a similar tactical prover in SICStus Prolog [33]. Thought the performance of the two provers cannot be compared directly because they were implemented on different platforms, the Prolog prover was roughly two orders of magnitude faster, with access decisions typically reached in around 5 milliseconds.

If this were a production-strength implementation, we would likely have implemented the theorem prover in Java. The capabilities of Twelf are far greater than what we need and impose a severe performance penalty; a custom-made theorem prover that had only the required functionality would be more lightweight. This also impacts the proof-checking performance; a specialized checker [9] would be much faster.

Chapter 5

Extending the Logic

The application-specific logic we have defined is able to express many useful security policies. To be useful in practice, however, a security logic needs to be able to describe several additional common ideas. In this chapter we describe extensions to our application-specific logic that capture expiration, revocation, a more flexible kind of delegation, and a more general and user-friendly notion of principal.

5.1 Expiration

In Chapter 2 we introduced an operator that allowed delegation statements to become valid only after a certain time. Equally useful is the **before** operator, which can be used to indicate that delegations expire.

$$\textbf{before}\,(N,\,F) \overset{\text{def}}{=} (\textbf{localtime} < N) \rightarrow F$$

From this definition we can prove a theorem that shows that **before** behaves similarly to **after**.

$$\frac{A\ \textbf{says}\ (\textbf{before}\ (N,\ F))\quad \textbf{localtime} < N}{A\ \textbf{says}\ F}\qquad (\text{BEFORE-E})$$

5.2 Key Management and Revocation

In the scenario of Chapter 2, the principals knew each other by their public keys. The Registrar, for example, listed Alice in the roster for CS101 by delegating to her key the authority to speak on behalf of the class. The Registrar implicitly knew, perhaps through having verified it himself, that the person who held Alice's private key was Alice.

In a more realistic scenario, the binding between Alice's identity and her key would be managed by a certification authority (CA). The CA might export a local name that represented Alice, which it would bind to the holder of Alice's private key.

$$\mathit{CA}\ \textbf{signed}\ ((\textbf{name}\ \texttt{pubkey}_{\texttt{Alice}})\ \textbf{speaksfor}\ \mathit{CA}.\texttt{Alice})$$

The Registrar could now make Alice a member of CS101 by referring to $\mathit{CA}.\texttt{Alice}$ instead of to Alice's key.

Alice's private key could become compromised. In this case, the CA would want to renege on its assertion that Alice's identity was bound to the corresponding public key. One way to implement this would be to issue name-to-key bindings with a limited lifespan.

$$\mathit{CA}\ \textbf{signed}\ (\textbf{before}\ (\mathit{tomorrow},\ ((\textbf{name}\ \texttt{pubkey}_{\texttt{Alice}})\ \textbf{speaksfor}\ \mathit{CA}.\texttt{Alice})))$$

If Alice's key were compromised, the CA simply would not issue a new statement biding Alice's identity to her old public key. This approach is appealing in its simplicity, but also has drawbacks: even if stolen, Alice's public key will continue to be bound to her identity until *tomorrow*. To reduce the likelihood of misuse the interval at which the CA reissued bindings would have to be short, but that would require the CA to be signing and all of the CA's clients constantly to be fetching the new certificates.

A more practical choice—the one made in reality [51]—is to allow the name-to-key bindings to be valid for a long interval and to give the CA a mechanism for revoking bindings that are no longer valid even though they have not reached their expiration date.

The traditional idea of revocation [32] is nonmonotonic—it allows new facts to cause previously known facts to disappear. The new fact is a certificate revocation list; the previously known fact is a certificate. The validity of any particular certificate depends on knowing the most current revocation list. In any large system with many principals and certificates, it is impossible to guarantee with reasonable efficiency that every principal has the most up-to-date revocation list. In the presence of adversaries who might try to deny access to the newest revocation lists, this becomes even bigger problem. In addition, reasoning about nonmonotonic revocation is complicated, because only exhaustive examination of all facts can demonstrate that no revocation list invalidates a particular certificate.

In our logic we implement a monotonic version of revocation [36]—our certificates have meaning only in the presence of a valid revocation list. More specifically, a certificate described by $\mathbf{cert}(A,F,N)$ can be understood to imply that A believes the formula F only in the presence of a valid revocation list that shows that the certificate with serial number N has not been revoked. A revocation list $\mathbf{revlist}(T_1,T_2,L)$ contains a list L of expired certificates and specifies the interval $\langle T_1,T_2]$ during which this list is valid. If the current time is after T_2, a new revocation list must be obtained. We require that the revocation list is issued by the same principal that issued the certificate that is being revoked.

$$\frac{\textbf{cert}(A,F,N) \qquad A\ \textbf{signed}\ (\textbf{revlist}(T_1,T_2,L)) \qquad \textbf{localtime} < T_2 \qquad N \notin L}{A\ \textbf{says}\ F} \tag{CERT-E}$$

As before, we define **cert** and **revlist** in a way that allows the (CERT-E) rule to be proved as a theorem.

A certificate is data that is signed by a principal, so it seems natural to define it using the **signed** operator. The question then becomes: what specific form should the signed formula have? We want to convey the idea that the content of the certificate is valid only under certain circumstances, so most likely we need to use some form of implication which will allow us to conclude that the principal who created the certificate stands behind its content only if the premises of the implication can be discharged. The premises must be derivable from the revocation list, because it is a valid revocation list (i.e., a revocation list that has not expired and that does not revoke a particular certificate) that gives meaning to a certificate. The formulas that are easiest to fabricate are formulas about belief. We have on multiple occasions taken advantage of modus ponens inside says (SAYS-I3), and this is another situation in which it will come in handy. We will define a certificate as a **signed** statement in which the content, F, is expressed as the belief of a fictitious entity. Certificates are revoked by serial number, so the fictitious entity will represent the serial number of the current certificate.

$$\mathbf{cert}(A,F,N) \overset{\text{def}}{=} A\ \mathbf{signed}\ ((\mathbf{name}\ \texttt{SerialNumber})\ \mathbf{says}\ ((\mathbf{name}\ N)\ \mathbf{says}\ F))$$

One way to conclude from a certificate $\mathbf{cert}(A,F,N)$ that the principal A believes the formula F is to demonstrate that A believes that if $\texttt{SerialNumber}.N$ belies F then F is true, i.e., A **says** $((\texttt{SerialNumber}.N\ \mathbf{says}\ F) \rightarrow F)$. A should believe this, of course, only if the certificate has not yet been revoked, that is, if there exists a revocation list that is both current and does not revoke the certificate. This leads us to the definition of a revocation list.

$$\begin{aligned}\mathbf{revlist}(T_1,T_2,L) \overset{\text{def}}{=}\ & \forall N\ \forall F. \\ & \mathbf{localtime} < T_2 \\ & \rightarrow N \notin L \\ & \rightarrow (\mathbf{name}\ \texttt{SerialNumber})\ \mathbf{says}\ ((\mathbf{name}\ N)\ \mathbf{says}\ F) \\ & \rightarrow F\end{aligned}$$

To be relevant, a revocation list needs to be issued by the same principal who issued the certificate whose validity we are confirming. This ensures both that there is no confusion about whose certificate a particular serial number refers to and that a certificate can be revoked only by an authorized entity, the issuer.

A revocation list is a chained implication. To conclude from a certificate that A believes F, we will first need to resolve the implication chain to its last two elements ($(\mathbf{name}\ \texttt{SerialNumber})\ \mathbf{says}\ ((\mathbf{name}\ N)\ \mathbf{says}\ F) \rightarrow F$). To discharge the first two parts of the implication (**localtime** $< T_2$ and $N \notin L$), we need to demonstrate that the revocation list is current and that the list of serial numbers it carries does not include the serial number of our certificate. Discharging the first two parts of the implication yields a formula which can be combined with the certificate (using SAYS-I3) to demonstrate that the issuer of the certificate believes its content. This is formalized by the CERT-E theorem.

5.3 Abstract Principals

Our notion of principals, as defined in Chapters 2 and 3, was simple and sufficiently expressive for our sample scenario. A real security logic, however, should have a more general notion of local names. The semantics of local names as abbreviations (e.g., $\mathit{Alice}\text{'}s\ \mathit{Bob}\ \textbf{says}\ F = \mathit{Alice}\ \textbf{says}\ (\mathit{Bob}\ \textbf{says}\ F)$) seems appropriate and flexible enough, but the interface exposed to the user should be improved: we should not need different operators to delegate authority to a simple principal and to a local name. Ideally, there should be a new type of principals that represents both principals created from a public key and arbitrarily long local names, i.e., the A in $A\ \textbf{says}\ F$ should range not only over principals created from single strings (*Alice*), but also over their local names (*Alice*'*s Bob*), the local names' local names (*Alice*'*s Bob*'*s Charlie*), and so on.

Our scheme of encoding the application-specific logic in higher-order logic makes adding this functionality at least conceptually straightforward.

First, we introduce a new type, `prin`, to represent our more powerful principals. Each principal is specified by one or more strings: *Alice* by the string $\mathtt{pubkey}_{\mathtt{Alice}}$, *Alice*'*s Bob*'*s Charlie* by the strings $\mathtt{pubkey}_{\mathtt{Alice}}$, `Bob`, and `Charlie`. We hence define our new type to be a list of strings.

$$\mathtt{prin} = \mathtt{string\ list}$$

Second, we need to define constructors that will let us create principals of the type `prin`.

$$\textbf{mkprin} : \texttt{string} \rightarrow \texttt{prin}$$

So we can represent them in the PCA logic, the constructors need to have semantics, i.e., definitions. Principals are really just lists of strings, so a constructor that takes a single string and returns a principal can be defined as the cons (::) operator.

$$\textbf{mkprin}\, A \overset{\text{def}}{=} A :: \text{nil}$$

Similarly, we can define a constructor that creates local principals.

$$\textbf{localprin} : \texttt{prin} \rightarrow \texttt{string} \rightarrow \texttt{prin}$$
$$\textbf{localprin}\ (A,S) \overset{\text{def}}{=} S :: A$$

A constructor that creates a local principals several levels deep may also be useful.

$$\textbf{localprin}' : \texttt{prin} \rightarrow \texttt{string}\ \texttt{list} \rightarrow \texttt{prin}$$
$$\textbf{localprin}'\ (A,L) \overset{\text{def}}{=} \textbf{concat}\ (L,A)$$

Using these constructors we can represent principals like *Alice* and *Alice's Bob*.

$$\begin{aligned} \textit{Alice} &= \textbf{mkprin}\ (\texttt{pubkey}_{\texttt{Alice}}) \\ &= \texttt{pubkey}_{\texttt{Alice}} :: \text{nil} \\ \textit{Alice's Bob} &= \textbf{localprin}\ (\textbf{mkprin}\ (\texttt{pubkey}_{\texttt{Alice}}), \texttt{Bob}) \\ &= \texttt{Bob} :: \texttt{pubkey}_{\texttt{Alice}} :: \text{nil} \end{aligned}$$

As before, the definitions of the type `prin` and the corresponding constructors need not concern a user of the logic; they are shown here to aid in explanation. For convenience, we will continue to refer to principals such as *Alice*'*s Bob* as *Alice*.*Bob*; now that abbreviation means $\textbf{localprin}\ (\textbf{mkprin}\ (\texttt{pubkey}_{\texttt{Alice}}), \texttt{Bob})$.

Next comes the harder task of developing operators that take arguments of type `prin`. As with the original semantics, the key definition is that of the **says** operator. The corresponding operator that ranges over arguments of type `prin` we will call $\textbf{says}'$. Since we decided we were satisfied with the initial semantics of local names, and wanted to change only the user interface, we can define $\textbf{says}'$ in terms of **says**, so that, for example, $\textit{Alice's Bob's Charlie}\ \textbf{says}'\ F \equiv \textit{Alice}\ \textbf{says}\ (\textit{Bob}\ \textbf{says}\ (\textit{Charlie}\ \textbf{says}\ F))$.

As a first step toward defining $\textbf{says}'$, let us informally define a function that performs a single step of expanding the abbreviation.

$$\text{unroll}\ (\langle \textit{head} :: \textit{tail},\ F\rangle) \quad = \quad \langle \textit{tail},\ (\textbf{name}\ (\textit{head})\ \textbf{says}\ F)\rangle$$

The unroll function maps pairs to pairs. Its argument is a pair composed of a list of strings (i.e., the principal) and a formula F. The function generates the formula that the head of the list of strings believes F. The function's output is a pair composed of the tail of the input string list and the generated formula.

As an illustration, let us apply the unroll function to the root principal that represents Alice and the formula F.

$$\begin{aligned} \text{unroll}\ (\langle \textbf{mkprin}\ (\texttt{pubkey}_{\texttt{Alice}}),\ F\rangle \quad &= \quad \text{unroll}\ (\langle \texttt{pubkey}_{\texttt{Alice}} :: \text{nil},\ F\rangle) \\ &= \quad \langle \text{nil}, (\textbf{name}(\texttt{pubkey}_{\texttt{Alice}})\ \textbf{says}\ F)\rangle \end{aligned}$$

More to the point, applying unroll twice to the principal that describes Alice's Bob recreates the formula that we are trying to abbreviate.

$$\begin{aligned}
&\text{unroll}\,(\text{unroll}\,(\langle \texttt{Bob} :: \texttt{pubkey}_{\texttt{Alice}} :: \text{nil},\ F\rangle) = \\
&\quad = \text{unroll}\,(\langle \texttt{pubkey}_{\texttt{Alice}} :: \text{nil},\ \textbf{name}(\texttt{Bob})\ \textbf{says}\ F\rangle) \\
&\quad = \langle \text{nil}, \textbf{name}(\texttt{pubkey}_{\texttt{Alice}})\ \textbf{says}\ (\textbf{name}(\texttt{Bob})\ \textbf{says}\ F)\rangle
\end{aligned}$$

It seems apparent that the intended meaning of the $\textbf{says}'$ operator can be achieved by applying unroll the appropriate number of times. This is exactly how we define $\textbf{says}'$.

$$\begin{aligned}
A\ \textbf{says}'\ F \quad \overset{\text{def}}{=} \quad & \forall N\ \forall Y. \\
& \text{length}\ (A, N) \\
& \rightarrow \text{iter}\ (\text{unroll}, N, \langle A, F\rangle, Y) \\
& \rightarrow \text{snd}\ (Y)
\end{aligned}$$

The definition uses relations rather than functions, which makes the definition appear more complicated than it is. Conceptually, however, it is simple: for all numbers N and pairs Y; if N is the length of the principal, apply unroll N times; if Y is the result of applying unroll N times, its second component, $s_1\ \textbf{says} \ldots s_n\ \textbf{says}\ F$, is exactly the intended meaning of $s_1 \ldots s_n\ \textbf{says}'\ F$. We use the relation iter to apply unroll N times; this and many other ideas, including the theory of lists that we use here and in Section 5.2, have been defined in higher-order logic as part of the Foundational Proof-Carrying Code project [7].

The definition of iter is similar in concept to the definition of **says**: iter is the particular relation C that satisfies exactly the rules we specify.

$$\begin{aligned}
\text{iter}\,(F,N,X,Y) \;\overset{\text{def}}{=}\; & \forall C\;. \\
& (\forall Z\;.\;C\;(F,0,Z,Z)) \\
& \rightarrow (\forall N'\;\forall Z'\;\forall Z''\;.\;\text{isNat}(N') \rightarrow N' > 0 \\
& \qquad \rightarrow C\;(F,N'-1,Z',Z'') \rightarrow C\;(F,N',Z',F(Z''))))\\
& \rightarrow C\;(F,N,X,Y)
\end{aligned}$$

To complete the transition from **says** to **says**′, we need to prove some theorems about **says**′. A useful one to start with is the dual of SAYS-I.

$$\frac{S\;\mathbf{signed}\;F}{\mathbf{mkprin}(S)\;\mathbf{says}'\;F} \qquad (\text{SAYS'-I})$$

Since **says**′ is just an abbreviation of the appropriate use of **says**, it seems reasonable that we should be able to prove about **says**′ everything that we can prove about a chain of application of **says**. We can prove, for example, the following theorems.

$$\frac{A\;\mathbf{says}\;(A\;\mathbf{says}\;F)}{A\;\mathbf{says}\;F} \tag{5.1}$$

$$\frac{A\;\mathbf{says}\;(B\;\mathbf{says}\;(A\;\mathbf{says}\;(B\;\mathbf{says}\;F)))}{A\;\mathbf{says}\;(B\;\mathbf{says}\;F)} \tag{5.2}$$

It should come as no surprise, therefore, that we can also prove a similar theorem for **says**′.

$$\frac{A\ \mathbf{says}'\ (A\ \mathbf{says}'\ F)}{A\ \mathbf{says}'\ F} \qquad (\textsc{says'-taut})$$

Once we have proved all the necessary theorems about **says**′, operators like **speaksfor**, whose definitions rely on **says**, can be redefined using **says**′. In this way, uses of **says** are completely confined to the definition of and the proofs of theorems about **says**′. A user of the logic has at his disposal a much more powerful notion of principals than before, and does not need to be aware how these principals are implemented.

We can alter the meanings of some operators to take advantage of our new notion of principals. In our definition of **speaksfor**, for example, we will encompass the idea that if principal *A* speaks for principal *B* then anything believed by a compound principal rooted at *A* is also believed by the corresponding compound principal rooted at *B*.

$$\begin{aligned} A\ \mathbf{speaksfor}'\ B \overset{\text{def}}{=} \forall U \forall N \forall L.((&\mathbf{localprin}'(A,L)\ \mathbf{says}'\ (\mathbf{goal}(U,N))) \\ &\rightarrow (\mathbf{localprin}'(B,L)\ \mathbf{says}'\ (\mathbf{goal}(U,N)))) \end{aligned}$$

There are, of course, other useful notions of what “speaking for” a principal should mean, so we could define a whole family of delegation operators to describe the different ways in which principals might wish to delegate their authority.

5.4 Delegation Across Domains

An important feature for cross-domain authorization is the ability to delegate a privilege under a new name. Consider, for example, a case of two companies that are merging. The employees of company A access accounting information through a web page called `accounting.html`. The employees of company B call their own accounting page `Buchhaltung.html`. After the merger the combined accounting information resides at `accounting.html`. Everyone who was authorized to access `Buchhaltung.html` should also be able to access the new page.

The delegation operators we defined in Chapter 2 can delegate authority to company B's certification authority in such a way that every employee of company B who has access to `accounting.html` can access the same page of the merged company. The operators are not sufficiently flexible, however, to provide the kind of renaming needed in this situation. Hence we add an operator similar to **delegate**. $\mathbf{delegateAs}(A,U_a,B,U_b)$ extends A's authority over the resource U_a to B under the name U_b. If the principal B asserts that it is OK to access U_b this will have the same effect as if A asserted that it is OK to access U_a.

$$\mathbf{delegateAs}(A,U_a,B,U_b) \stackrel{\text{def}}{=} \forall N.(B\ \mathbf{says}\ (\mathbf{goal}(U_b,N))) \rightarrow (A\ \mathbf{says}\ (\mathbf{goal}(U_a,N)))$$

Based on this definition we can now define the behavior of **delegateAs** as a theorem.

$$\frac{A\ \mathbf{says}\ (\mathbf{delegateAs}(A,U_a,B,U_b)) \quad B\ \mathbf{says}\ (\mathbf{goal}(U_b,N))}{A\ \mathbf{says}\ (\mathbf{goal}(U_a,N))} \quad (\text{RENAME-E})$$

5.5 An Extended Example

With the extensions we define in this chapter, our toy logic of Chapters 2 and 3 becomes a full-fledged distributed authorization logic that is able to describe with sufficient precision many realistic scenarios. As an illustration, we revisit the example of Alice, Bob, and the Registrar. Initially we had ignored issues such as key management, expiration, and revocation; this time we will show how to model more realistically the security policy protecting Bob's web page. The detail in which this scenario can be described using our logic is similar to what might be achieved with SPKI [23] or the logic underlying the Taos authorization mechanism [53].

As before, to gain access to the midterm results page, Alice must prove that Bob believes the goal statement.

$$(\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{Bob}}))\ \mathbf{says}\ (\mathbf{goal}\ (\mathtt{midterm.html},\ \mathtt{nonce}))$$

The initial example assumed that the principals knew each other's public keys; this time, a certification authority (CA) binds principals' public keys to their names. Bob's security policy, therefore, will refer to the Registrar by name rather than by key. In addition, the Registrar we want to model is not an individual like Alice or Bob, but a function of Bob's University. We model this indirection by using local names. To refer to the entity that the CA knows as "University," we say $\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ}$. This University's Registrar, then, is $\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ.Reg}$. As is customary, we assume that everyone knows the CA's public key.

In addition to stating his policy using this additional level of indirection, Bob also wants the midterm results web page to be visible only until the end of the semester.

$$\begin{aligned}\mathcal{P}_1 = \mathtt{pubkey}_{\mathtt{Bob}}\ &\mathbf{signed}\ (\mathbf{before}(\mathit{end\text{-}of\text{-}semester},\ \mathbf{after}\ (8\mathit{P.M.},\\ &\quad \mathbf{delegate}\ (\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{Bob}}),\\ &\quad\quad \mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ.Reg.CS101},\ \mathtt{midterm.html}))))\end{aligned}$$

For the principal $\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ.Reg.CS101}$ to be meaningful, both the CA and the University need to create their respective local names. The CA does this in the traditional fashion, via a revocable certificate. Following our implementation of certificates, in order to be valid a certificate must be accompanied by a current revocation list that does not invalidate that certificate. Here, the CA signs a revocation list that revokes no certificates.

$$\begin{aligned}\mathcal{P}_2 = \mathbf{cert}(&\mathtt{pubkey}_{\mathtt{CA}}, \mathbf{before}\ (\textit{next-year},\\ &(\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{Univ}})\ \mathbf{speaksfor}\ (\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ}))),\\ &0001)\\ \mathcal{P}_3 = \mathtt{pubkey}_{\mathtt{CA}}&\ \mathbf{signed}\ (\mathbf{revlist}\ (\textit{yesterday},\ \textit{tomorrow},\ \mathrm{nil}))\end{aligned}$$

The CA has delegated to the holder of the University's private key the right to speak on behalf of the principal $\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ}$.

Unlike the CA, the University eschews certificates in favor of simplicity, but reissues its statements weekly.

$$\begin{aligned}\mathcal{P}_4 = \mathtt{pubkey}_{\mathtt{Univ}}\ \mathbf{signed}\ (&\mathbf{before}(\textit{next-week},\\ &\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{Reg}})\ \mathbf{speaksfor}\ (\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{Univ}}).\mathtt{Reg})))\end{aligned}$$

With this statement, the University bestowed upon the holder of the private key corresponding to $\mathtt{pubkey}_{\mathtt{Reg}}$ the authority to act as the University's Registrar until next week.

Note that although the CA maps the University's public key to a local name, the CA exerts no control over the the University's own name space. The University is free to manage its own local names, and uses a local name to refer to the Registrar. The Registrar is now represented by the principal $\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{Univ}}).\mathtt{Reg}$ as well as by $\mathbf{mkprin}(\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ.Reg}$.

Now that we have established the Registrar's authority, it is time for the Registrar to issue a statement making Alice a member of the CS101 class. This statement is similar to the Registrar's old policy, except that now Alice is referred to by her proper name (and probably her Social Security Number or some other information that will guarantee uniqueness). As with other pieces of the security policy, this one has a limited lifetime.

$$\begin{aligned}\mathcal{P}_5 = \texttt{pubkey}_{\texttt{Reg}}\ &\textbf{signed}\ (\textbf{before}(\textit{next-week},\\ &\textbf{mkprin}(\texttt{pubkey}_{\texttt{CA}}).\texttt{Alice}\ \textbf{speaksfor}\ (\textbf{mkprin}(\texttt{pubkey}_{\texttt{Reg}}).\texttt{CS101})))\end{aligned}$$

Naturally, this statement can be useful to Alice only if the CA issues a certificate binding Alice's name to her key.

$$\begin{aligned}\mathcal{P}_6 = \textbf{cert}(\texttt{pubkey}_{\texttt{CA}}, &\textbf{before}\ (\textit{next-year},\\ &(\textbf{mkprin}(\texttt{pubkey}_{\texttt{Alice}})\ \textbf{speaksfor}\ (\textbf{mkprin}(\texttt{pubkey}_{\texttt{CA}}).\texttt{Alice}))),\\ &0002)\end{aligned}$$

There is no need for the CA to issue another revocation list, since the previous one ($\mathcal{P}_3$) is still valid.

The final piece of the puzzle is provided by Alice.

$$\mathcal{P}_7 = \texttt{pubkey}_{\texttt{Alice}}\ \textbf{signed}\ \textbf{goal}\ (\texttt{midterm.html},\ \texttt{nonce})$$

We will briefly run through the steps of assembling these seven statements into a proof that will give Alice access to the midterm results. We abstract from the details of demonstrating that all time-dependent statements are currently valid and assume that each of the **signed** statements is converted to a **says** statement.

First, Alice's statement ($\mathcal{P}_7$) can be combined with the CA's certificate ($\mathcal{P}_6$) and revocation list ($\mathcal{P}_3$) to conclude that the principal $\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Alice}$ believes that the web page may be accessed. From the Registrar's delegation statement making Alice a member of CS101 ($\mathcal{P}_5$) we can then reason that $\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{Reg}}).\mathtt{CS101}$ thinks the page may be accessed. The University's delegation to the Registrar ($\mathcal{P}_4$) and the CA's certificate binding the University's public key to its name ($\mathcal{P}_2$) and associated revocation list ($\mathcal{P}_3$) can be used to derive that $\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{CA}}).\mathtt{Univ}.\mathtt{Reg}.\mathtt{CS101}$ believes the goal statement. This can be combined with Bob's policy ($\mathcal{P}_1$) to conclude $(\mathbf{mkprin}\ (\mathtt{pubkey}_{\mathtt{Bob}}))\ \mathbf{says}\ (\mathbf{goal}\ (\mathtt{midterm.html},\ \mathtt{nonce}))$, i.e., that Bob believes that it is OK for Alice to gain access to the web page.

Chapter 6

Conclusions and Future Work

A number of logics have been proposed for modeling and solving the problems of distributed authentication and authorization. The logics have two typically incompatible goals: expressivity, the ability to represent as many authorization scenarios as possible; and decidability, the characteristic that answers to access-control questions can always be derived. Proof-carrying authorization is an alternative to the mainstream approach. PCA is a framework for defining security logics that allows logics engineered for solving different problems to easily interoperate. By requiring clients to provide to servers proofs of authorization, PCA ensures that servers will always be able to reach access-control decisions even if clients use arbitrarily complex logics.

6.1 Contributions

This dissertation gives substance to the idea of proof-carrying authorization [8]. We specify a particular higher-order logic that can be used as a substrate for defining application-specific security logics, and explain how PCA can be used to develop a usable distributed authorization framework. We use PCA to develop a particular application-specific logic and design and implement a system for access control on the web. Our presentation of our application-specific logic is suitable for use as a tutorial for using PCA to give semantics to application-specific security logics.

One of the caveats of using security logics has been that they are often not an accurate model of an implemented system. If a logic only approximates an actual authorization protocol then any theorems we may be able to prove about the logic need to be taken with a grain of salt. In security applications perhaps more than in other fields, it is important to bridge the gap between model and implementation. This dissertation demonstrates that formal methods and tools, such as security logics and theorem provers, need no longer be used just to model or explain real systems. These tools have become practical enough, or are just a step away from practical enough, that they are suitable for use as building blocks of real systems.

The particular contributions of this dissertation are:

- A formulation of the PCA logic [8] suitable for implementing practical authorization logics. Our PCA logic contains only the standard axioms of higher-order logic. This ensures that it is completely general, and suitable for encoding more than just security logics with particular characteristics. A prerequisite for trusting any application-specific logic that we might define in PCA is the soundness of the underlying framework; since our formulation of the PCA logic does not contain any axioms other than those of higher-order logic, it is easier to demonstrate and trust that our PCA logic is sound.

- An application-specific security logic, similar in style to previous logics of authentication [10, 15, 30]. Our application-specific logic has a polynomial-time decision procedure, and can describe a wide range of authorization scenarios, including those involving revocation, local name spaces, and delegation with renaming. We give our logic a semantics in the PCA logic, which both guarantees the soundness of the application-specific logic and makes it suitable for use in PCA systems.

- In conjunction with previous work [11], the design and implementation of a system for controlling access to web pages via PCA. Our system is implemented as add-on modules to standard web browsers and web servers and demonstrates that it is feasible to use a proof-carrying authorization framework as a basis for building real systems.

  The system allows pieces of the security policy to be distributed across arbitrary hosts. Through the process of iterative proving the client repeatedly fetches proof components until it is able to construct a proof. This mechanism allows the server's policy to be arbitrarily complex, controlled by a large number of principals, and spread over an arbitrary network of machines in a secure way. Iterative authorization, or allowing the server to repeatedly challenge the client with new challenges during a single authorization transaction, provides a great deal of flexibility in designing security policies.

  Our performance results demonstrate that it is possible to reduce the inherent overhead to a level where a system like ours is efficient enough for real use. To increase performance we make heavy use of caching and add a module system to the proof language.

6.2 Future Work

The work described in this dissertation leaves open many directions for future research.

A particularly interesting problem deals with managing privacy while using automated provers. In our running example, Alice's proxy—acting on behalf of Alice—collects facts that describe which classes Alice is taking and sends them to Bob. In a more general setting it is easy to imagine that other private data, such as social security and credit card numbers, could be part of an authorization decision. In that case, we would not want an automated prover to blindly collect whatever facts were necessary and use them to create a proof. Alice, for example, may not want to send out her credit card number unless she explicitly trusts the party that requested it. In addition, a malicious server could attempt to gain as much information as possible about a client by specifying proof goals that required the particular datums to be used as premises.

One approach to solving this problem could be to use existing bit-commitment or oblivious transfer protocols. The question, then, would be how to integrate these protocols with the PCA protocols we described in Chapter 4. But because a client's prover is not likely to know all of the client's preferences—it would not know, for example, whether Alice trusts a web site that is encountered for the first time—it seems necessary that there should be some interaction between the human client and her automated proving mechanisms. This is one area that I intend to explore.

Another interesting problem space is developing protocols that take advantage of the common underlying framework that PCA provides. All proofs are expressed in the same underlying logic, so partial proofs or lemmas can be communicated among clients. It might be useful, for example, to share lemmas that are time-consuming or otherwise hard to prove. Some lemmas might be provable only by certain clients, because others may not know or be able to find the facts needed for the proofs. Clients may even use different but overlapping application-specific logics, which would allow each of them to prove different sets of lemmas. Mechanisms need to be developed that allow these lemmas to be created, communicated between clients, and used during proving.

Our prototype web access-control system (Chapter 4), despite demonstrating the feasibility of our approach, left much to be desired in terms of performance. Though mostly an engineering challenge, several particular problems need to be answered. It is not too difficult to write fast provers that ignore the semantics of the application-specific logic. Yet to be modular and easy to maintain, to cope with a changing logic, for example, a prover should not be completely oblivious of the semantics of the logic. Fetching facts is crucial to successfully generating a proof. In our examples the number of facts was small so it was simple to collect them all; in a more realistic scenario it is likely that fact fetching, a potentially time-consuming operation, would need to be guided by the prover according to its needs.

Applications of authorization systems, such as physical access control, for example, have traditionally been limited by the requirements of servers. With PCA, however, servers no longer need to collect facts or perform computation-intensive proving. This opens the door to novel authorization-system architectures that take advantage of the potentially much more lightweight servers.

Appendix A

Tactical Prover

A.1 Sample Tactical Prover

A tactical theorem prover, implemented in Twelf, attempts to prove that access should be allowed. The prover is presented with a list of assumptions and a goal, and uses tactics to guide its search for the proof. Each assumption consists of a formula and its proof. Each tactic describes how a particular subgoal might be proven. The **p-signed** tactic, for example, generates a proof of the subgoal A **says** F if a proof can be found for the formula A **signed** F; the proof of A **says** F consists of the application of the SAYS-I rule to the proof of the formula A **signed** F.

Proof search begins by checking whether the first tactic, **init**, applies. **init** checks whether the current subgoal is the first assumption. If it is, the proof of that assumption is returned as the proof of the subgoal; otherwise, the prover will try all the other tactics in the order in which they are stated. The last tactic, **next-fact**, removes the first element from the list of assumptions. The prover then attempts to apply each tactic again; this gives the **init** tactic a chance to check whether each of the assumptions matches a given subgoal. The prover thus implements a depth-first search in which the proof goal is the root and the assumptions are the leaves.

Note that the **p-after** tactic retains the condition, also present in the AFTER-E rule, that the current time must be greater than N, the argument of the **after** statement. A practical implementation is described in Chapter 3 and reflected in the tactics shown in Appendix A.2. For now, we assume that the tactic can be used only if the side condition is met.

init:

$$\frac{\qquad}{\mathit{findproof} \qquad ((A\ \mathrm{by}\ P),\Gamma) \qquad A \qquad P}$$

p-signed:

$$\frac{\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{signed}\ F \qquad P}{\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{says}\ F \qquad (\textsc{says-i}\ P)}$$

p-after:

$$\frac{\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{signed}\ (\mathbf{after}\ (N,\ F)) \qquad P}{\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{says}\ F \qquad (\textsc{after-e}\ (\textsc{says-i}\ P))}\ \ \mathit{time} > N$$

p-speaksfor-e1:

$$\frac{\begin{array}{l}\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{says}\ (B\ \mathbf{speaksfor}\ A) \qquad P_1 \\ \mathit{findproof} \qquad \Gamma \qquad B\ \mathbf{says}\ (\mathbf{goal}(U,N)) \qquad P_2\end{array}}{\mathit{findproof} \qquad \Gamma \qquad A\ \mathbf{says}\ (\mathbf{goal}(U,N)) \qquad (\textsc{speaksfor-e}1\ P_1\ P_2)}$$

p-speaksfor-e2:

$$\frac{\begin{array}{l}\textit{findproof} \quad \Gamma \quad A\ \textbf{says}\ (B\ \textbf{speaksfor}\ A.S) \quad P_1 \\ \textit{findproof} \quad \Gamma \quad B\ \textbf{says}\ (\textbf{goal}(U,N)) \quad P_2\end{array}}{\textit{findproof} \quad \Gamma \quad A.S\ \textbf{says}\ (\textbf{goal}(U,N)) \quad (\textsc{speaksfor-e2}\ P_1\ P_2)}$$

p-delegate-e1:

$$\frac{\begin{array}{l}\textit{findproof} \quad \Gamma \quad A\ \textbf{says}\ (\textbf{delegate}\ (A,\ B,\ \texttt{U})) \quad P_1 \\ \textit{findproof} \quad \Gamma \quad B\ \textbf{says}\ (\textbf{goal}(U,N)) \quad P_2\end{array}}{\textit{findproof} \quad \Gamma \quad A\ \textbf{says}\ (\textbf{goal}(U,N)) \quad (\textsc{delegate-e1}\ P_1\ P_2)}$$

next-fact:

$$\frac{\textit{findproof} \quad \Gamma \quad A \quad P}{\textit{findproof} \quad (X,\Gamma) \quad A \quad P}$$

A.2 Implemented Tactical Prover

While giving our application-specific logic a semantics (Chapter 3), we changed slightly our way of describing local names and the time on each host. The accordingly modified tactics are shown here.

init:

$$\frac{}{\textit{findproof} \quad ((A\ \text{by}\ P),\Gamma) \quad A \quad P}$$

p-signed:

$$\frac{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{signed}\ F\quad P}{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ F\quad (\textsc{says-i}\ P)}$$

p-after:

$$\frac{\begin{array}{l}\mathit{findproof}\quad \Gamma\quad A\ \mathbf{signed}\ (\mathbf{after}\ (N,\ F))\quad P_1\\ \mathit{findproof}\quad \Gamma\quad \mathbf{localtime} > N\quad P_2\end{array}}{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ F\quad (\textsc{after-e}\ (\textsc{says-i}\ P_1)\ P_2)}$$

p-speaksfor-e1:

$$\frac{\begin{array}{l}\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (B\ \mathbf{speaksfor}\ A)\quad P_1\\ \mathit{findproof}\quad \Gamma\quad B\ \mathbf{says}\ (\mathbf{goal}(U,N))\quad P_2\end{array}}{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (\mathbf{goal}(U,N))\quad (\textsc{speaksfor-e}1\ P_1\ P_2)}$$

p-speaksfor-e2:

$$\frac{\begin{array}{l}\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (B\ \mathbf{speaksfor}'\ A.S)\quad P_1\\ \mathit{findproof}\quad \Gamma\quad B\ \mathbf{says}\ (\mathbf{goal}(U,N))\quad P_2\end{array}}{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (S\ \mathbf{says}\ (\mathbf{goal}(U,N)))\quad (\textsc{speaksfor-e}2\ P_1\ P_2)}$$

p-delegate-e1:

$$\frac{\begin{array}{l}\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (\mathbf{delegate}\ (A,\ B,\ \mathtt{U}))\quad P_1\\ \mathit{findproof}\quad \Gamma\quad B\ \mathbf{says}\ (\mathbf{goal}(U,N))\quad P_2\end{array}}{\mathit{findproof}\quad \Gamma\quad A\ \mathbf{says}\ (\mathbf{goal}(U,N))\quad (\textsc{delegate-e}1\ P_1\ P_2)}$$

p-delegate-e2:

$$\frac{\begin{array}{llll} \textit{findproof} & \Gamma & A\ \mathbf{says}\ (\mathbf{delegate}'\ (A,\ B.S,\ \mathtt{U})) & P_1 \\ \textit{findproof} & \Gamma & B\ \mathbf{says}\ (S\ \mathbf{says}\ (\mathbf{goal}(U,N))) & P_2 \end{array}}{\begin{array}{llll} \textit{findproof} & \Gamma & A\ \mathbf{says}\ (\mathbf{goal}(U,N)) & (\textsc{delegate-e}2\ P_1\ P_2) \end{array}}$$

next-fact:

$$\frac{\textit{findproof} \quad \Gamma \quad A \quad P}{\textit{findproof} \quad (X,\Gamma) \quad A \quad P}$$

Index of Authors

Bibliography

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, October 1998. 5, 10

[2] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993. 5, 11

[3] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269. ACM Press, Dec. 1993. 5

[4] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986. 29, 54, 55

[5] P. B. Andrews. Classical type theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 15. Elsevier Science, 2001. 37, 41

[6] P. B. Andrews, Sept. 2003. Personal communication. 56

[7] A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001. 90

[8] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, Nov. 1999. 5, 98, 99

[9] A. W. Appel, N. G. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In S. Autexier and H. Mantel, editors, *Verification Workshop*, volume 02-07 of *DIKU technical reports*, pages 41–52, July 25–26 2002. 68, 81

[10] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *21th IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 2000. 5, 99

[11] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002. 16, 100

[12] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. *The KeyNote trust-management system, version 2*, Sept. 1999. IETF RFC 2704. 13

[13] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999. 13

[14] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust-management system. In *Proceedings of the 2nd Financial Crypto Conference*, volume 1465 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer. 5, 12

[15] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb. 1990. 99

[16] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. In *Sixth International Conference on the World-Wide Web*, Santa Clara, CA, USA, Apr. 1997. 13

[17] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. 5, 14

[18] D. E. Clarke. SPKI/SDSI HTTP server / certificate chain discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, Sept. 2001. 10

[19] D. E. Clarke, J.-E. Elien, C. M. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001. 10

[20] T. Dierks and C. Allen. The TLS protocol version 1.0. Internet Request for Comment RFC 2246, Internet Engineering Task Force, Jan. 1999. Proposed Standard. 9

[21] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998. 5, 10

[22] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. Simple public key certificate. Internet Engineering Task Force Draft IETF, July 1997. 9

[23] C. M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, Sept. 1999. RFC2693. 5, 9, 94

[24] M. Erdos and S. Cantor. Shibboleth architecture draft v04. `http://middleware.internet2.edu/shibboleth/docs/`, Nov. 2001. 3

[25] S. Farrel and R. Housley. The internet attribute certificate profile for authorization. Internet Request for Comment RFC 3281, Internet Engineering Task Force, Apr. 2002. RFC 3281. 9

[26] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. IETF - Network Working Group, The Internet Society, June 1999. RFC 2616. 67

[27] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001. 1

[28] L. Guiri and P. Iglio. Role templates for content-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 153–159, New York, Nov. 6–7 1997. ACM Press. 14

[29] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software—Practice and Experience*, 30(15):1609–1640, Dec. 2000. 13

[30] J. Y. Halpern and R. 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, Mordano, Italy, June 1999. 5, 10, 99

[31] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993. 68

[32] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, Apr. 2002. RFC3280. 2, 5, 9, 84

[33] Intelligent Systems Laboratory. *SICStus Prolog User's Manual, Release 3.10.1*. Swedish Institute of Computer Science, 2003. 81

[34] O. Kornievskaia, P. Honeyman, B. Doster, and K. Coffman. Kerberized credential translation: A solution to web access control. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001. 2

[35] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992. 11

[36] N. Li and J. Feigenbaum. Nonmonotonicity, user interfaces, and risk assessment in certificate revocation. In *FC: International Conference on Financial Cryptography*. LNCS, Springer-Verlag, 2001. 84

[37] N. Li, J. Feigenbaum, and B. Grosof. A logic-based knowledge representation for authorization with delegation. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 162–174, Washington - Brussels - Tokyo, June 1999. IEEE. 14

[38] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2000. 14

[39] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACMTISS: ACM Transactions on Information and System Security*, 6(1):128–171, Feb. 2003. 14

[40] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002. 14

[41] E. C. Lupu and M. Sloman. Reconciling role-based management and role-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 135–142, New York, Nov. 6–7 1997. ACM Press. 14

[42] A. J. Maywah. An implementation of a secure web client using SPKI/SDSI certificates. Master's thesis, Massachusetts Institute of Technology, May 2000. 10

[43] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press. 68

[44] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33-38, Sept. 1994. 2

[45] F. Pfenning and C. Schürmann. System description: Twelf: A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10 1999. Springer. 61

[46] B. Ramsdell, editor. *S/MIME Version 3 Certificate Handling*, June 1999. RFC2632. 9

[47] B. Ramsdell, editor. *S/MIME Version 3 Message Specification*, June 1999. RFC2633. 9

[48] R. L. Rivest and B. Lampson. SDSI—A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, Apr. 1996. 9

[49] V. Samar. Single sign-on using cookies for web applications. In *Proceedings of the 8th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 158–163, Palo Alto, CA, 1999. 2

[50] C. P. Thacker, L. Stewart, and E. H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988. 11

[51] Thawte Consulting Ltd. Certification practice statement. `http://www.thawte.com/cps/`, Mar. 2002. 84

[52] C. Weider and J. Reynolds. *Executive Introduction to Directory Services Using the X.500 Protocol*, Mar. 1992. RFC1308. 9

[53] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994. 11, 94