SPKI/SDSI HTTP Server / Certificate Chain Discovery in SPKI/SDSI

by

Dwaine E. Clarke

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author . . . .
Department of Electrical Engineering and Computer Science
August 21, 2001

Certified by . . . .
Ronald L. Rivest
Andrew and Erna Viterbi Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

SPKI/SDSI HTTP Server /
Certificate Chain Discovery in SPKI/SDSI

by

Dwaine E. Clarke

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The issue of trust is of growing importance as our communities become increasingly interconnected. When resources are shared over an untrusted network, how are decisions on which principals are authorized to perform particular actions determined? SPKI/SDSI, a security infrastructure based on public-keys, is designed to facilitate the development of scalable, secure, distributed computing systems. It provides fine-grained access control, using a local name space hierarchy, and a simple, flexible, trust policy model; these features allow for the ability to create groups and delegate authorizations. Project Geronimo, named after the famous Native-American Apache chief, explores the viability of SPKI/SDSI by using it to provide access control over the Web. The infrastructure was integrated into the Netscape web client and Apache web server, using a previously developed SPKI/SDSI C Library. This thesis focuses on the server implementation. A SPKI/SDSI Apache module was designed and implemented: its principle functions are to protect web objects using SPKI/SDSI ACLs, and to determine whether HTTP client requests should be permitted to perform particular operations on protected objects. An administrative tool was developed to enable ACLs to be created, and updated, securely. The thesis also describes the algorithm for certificate chain discovery in SPKI/SDSI. Finally, the demonstration developed for Project Geronimo is outlined. The demo was successfully shown to our sponsors and various groups within the Laboratory for Computer Science.

Thesis Supervisor: Ronald L. Rivest
Title: Andrew and Erna Viterbi Professor of Electrical Engineering and Computer Science

Acknowledgments

Firstly, Many Thanks and Praise to the Almighty from Whom I have been blessed.

Many Thanks to my family who has always been there for me. Mummy, Daddy, Sylvan, Gerren, Thank you very much; I really appreciate it.

Many Thanks to my thesis advisor, Prof. Ron Rivest. Ron, Thank you for the opportunities and for believing in me more than I believed in myself.

Many Thanks to Carl Ellison, who has forever changed the way I think. My summers at Intel are among my most rewarding academic experiences to date.

Many Thanks to Andrew Maywah. Andrew and I went through so much together: SPKI/SDSI demos, classes, problem sets, class projects, Athena all-nighters ...

Many Thanks to Matt Fredette and Jean-Emile Elien who spent much time with Andrew and me helping us get our project off the ground.

Many Thanks to George Savvides and Matt Burnside (and also to practically everyone listed before) for proof-reading my thesis and for their very helpful comments.

Many Thanks to my friends (and with this statement, you've all made it into the acknowledgements!)

Finally, Many Thanks to NASA for their financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet has facilitated growth in the development of distributed computing systems. As more people use the Internet, the information and resources stored on networked computers become more valuable. This increases the need to develop security schemes which are scalable, flexible, easy to understand, and easy to use.

One approach has been to develop security infrastructures based on public keys. Today, the most common of these infrastructures is the X.509 Public-Key Infrastructure. This thesis describes SPKI/SDSI (Simple Public-Key Infrastructure/Simple Distributed Security Infrastructure), an alternative public-key based security infrastructure. SPKI/SDSI is motivated by the perception that X.509 is too complex and is incomplete. X.509's complexity arises from a dependency on global name spaces and its lack of flexibility. Its incompleteness can be immediately perceived if one tries to define a security policy (such as write an Access Control List (ACL)).

SPKI/SDSI is designed to facilitate the development of scalable, secure, distributed computing systems. It provides fine-grained access control, using a local name space hierarchy, and a simple, flexible, trust policy model. These features allow for the ability to create groups and delegate authorizations.

This thesis explores the viability of SPKI/SDSI by using it to provide access control over the Web. The infrastructure was integrated into the Netscape web client and the Apache web server, using a previously developed SPKI/SDSI C Library. After some consideration, this project was named “Project Geronimo”, after the famous Native-American Apache chief. This thesis focuses on the design and implementation of the SPKI/SDSI web server, and the discovery of certificate chains in SPKI/SDSI.

The principle result of this research has been the demonstration that SPKI/SDSI is a viable approach for providing security in distributed computing systems. The prototype implementation and security protocol developed for Project Geronimo are serving as the basis for some of the security schemes that will be used in the MIT Laboratory for Computer Science’s “Project Oxygen”[34, 2].

1.2 Our Contribution

The first part of Project Geronimo that was developed was the protocol between the client and the server. The protocol allows the server to make an access control decision when a client requests access to a protected resource (resources on the server are either public or protected by SPKI/SDSI ACLs). Our general protocol consists of four messages, and is a typical challenge-response protocol. In the first message, the client requests access to a resource. If the resource is public, the server honors the request; if the resource is protected by an ACL, the server issues a challenge to the client in the protocol’s second message. This challenge contains the requested resource’s ACL. Because of SPKI/SDSI’s ability to define groups, ACLs can be very simple and small, normally consisting of just one group. Using the server’s challenge, the client generates a response consisting of a signed request and a certificate chain; the digital signature provides proof that the request is authentic; the certificate chain provides proof that the request is authorized. The client uses the SPKI/SDSI Certificate Chain Discovery algorithm[3] to generate the certificate chain. A SPKI/SDSI certificate chain is a chain of authorization, typically used to prove that a particular requestor has been authorized to perform a particular request. The client sends its response to the server’s challenge in the protocol’s third message. The server verifies this response and honors the request if it verifies; otherwise it returns an error web page to the client. The requested resource, or the error page, is returned in the protocol's fourth message, from the server to the client.

The SPKI/SDSI HTTP client was designed and implemented by Andrew Maywah for his Master's thesis[33]. The client is implemented as a Netscape plugin, which offers such benefits as portability, simplicity, and ease of development. The plugin generates the responses to the server's challenges. The other important feature of the plugin is that it uses a small Java-based pop-up password box to prompt the user for his password to unlock his private key, and starts a small session window if the key is successfully unlocked. The session window maintains state between client requests during the same Netscape session. It prevents the user from having to re-enter his password every time he accesses a SPKI/SDSI protected document within the same Netscape session. There are no shared secrets between the user and the server, and the user's password is never sent across the network.

This thesis discusses the design and implementation of the SPKI/SDSI HTTP server. The server was implemented as an Apache module, utilizing Apache's modular architecture and well-defined Application Programming Interface (API). Its principle functions are to protect web objects using SPKI/SDSI ACLs, and to determine whether HTTP client requests should be permitted to perform particular operations on protected objects. A separate tool, using CGI (Common Gateway Interface), was also developed to enable an administrator to create, view and update ACLs securely.

Web objects on the server are protected on a per-directory basis. To protect a directory, a .htaccess file is created in it. The file contains the directive "`SPKI/SDSI on`", and pointers to the file with the SPKI/SDSI ACL and the error web page that is returned if the user's credentials do not grant him access. The error page is customizable, and can be used to provide a user with feedback on why his request failed, and information on how he may obtain valid certificates if he is a user who should be authorized to access the protected resource.

To determine whether a request for a protected resource should be honored, the server verifies the request's "*proof of authenticity*" and "*proof of authorization*". The "proof of authenticity" is a signed request, and the "proof of authorization" is a sequence/chain of certificates. The principal that signed the request must be the same principal that the chain of certificates authorizes.

The thesis also describes the new algorithm for Certificate Chain Discovery in SPKI/SDSI. The certificate chain discovery algorithm takes as input an ACL, a request, a public key, a set of signed certificates, and a timestamp. If it exists, the algorithm returns a certificate chain which provides proof, consisting of signed certificates, that the public key is authorized to perform the operation(s) specified in the request on the object protected by the ACL, at the time specified in the timestamp. Previous work on Certificate Chain Discovery in SPKI/SDSI resulted in the algorithm published in Jean-Emile Elien's Masters thesis[7]. Jean-Emile represented signed name and authorization certificates as rewrite rules, and derived new "unsigned authorization certificates" by composing signed certificates. The rules under which certificates were composed were explicitly stated in his thesis, and his algorithm had a worst case running time of $O(n^4 l)$ where "n" is the number of input certificates and "l" is the length of the longest subject in those certificates.

Our revised algorithm takes advantage of a constraint on the SPKI/SDSI certificate structure. Specifically, a certificate that defines a name cannot have an authorization tag: thus, a certificate can only define a name, or delegate an authorization, but cannot do both at the same time. We were able to refine the earlier certificate algorithm and separate the composition of name certificates from the reduction of authorization certificates. The result was a simpler, more intuitive algorithm, with a factor of $O(n)$ improvement over the original algorithm. The worst case running time of the new algorithm is $O(n^3 l)$, and, in practice, we expect it to run in time linear to the number of input certificates and length of the longest subject in those certificates. The new algorithm has been described in a journal paper, "Certificate Chain Discovery in SPKI/SDSI", which has been accepted for publication in the Journal of Computer Security[3].

The principal goal of our work on Project Geronimo was to develop a demo illustrating some of the capabilities and advantages of the SPKI/SDSI Infrastructure. This demo was successfully implemented using the SPKI/SDSI web client and server.

It featured a new user, Alice, who had already generated her SPKI/SDSI key-pair and installed the plugin, going through the process of gaining authorization credentials to view web pages to which she should be permitted to access. The demo featured SPKI/SDSI groups and name certificates, ACL administration, the client-side password box and session window, the server-side customizable error page, certificate chain discovery, and discretionary, fine-grained access control over an untrusted network. The web medium used for the demo made it attractive and more interesting, and it was successfully presented to several representatives from various groups in the MIT Laboratory for Computer Science. This thesis outlines the schematics of the demonstration.

1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 gives a brief overview of the security concepts most relevant to this thesis.
- Chapter 3 discusses the background to public-key infrastructures and SPKI/SDSI. The chapter concludes with a table comparing SPKI/SDSI with X.509 and PGP, another popular public-key security infrastructure.
- Chapter 4 describes the SPKI/SDSI Access Control Protocol, and compares it to other client access control schemes.
- Chapter 5 describes the design and implementation of the SPKI/SDSI HTTP Server. It gives a brief overview of the SPKI/SDSI HTTP Client. The details of the web client are in Andrew Maywah's Master's Thesis [33]. The chapter also describes the demo that was designed and implemented for the project.
- Chapter 6 describes the algorithm for discovering certificate chains in SPKI/SDSI.
- Chapter 7 concludes the thesis.

Chapter 2

Security Overview

2.1 Designing Secure Distributed Systems

Designing secure, yet scalable and easy to use distributed computing systems is a challenging problem. In most cases, the network connecting different computers is *untrusted*, as it is assumed that adversaries can eavesdrop on communication channels, and surreptitiously modify messages sent from one computer to another. The situation is further complicated by the fact that the adversary may not just be an attacker trying to do harm to the system from outside on the network, but can be a user who has valid access to some parts of the system and is trying to gain access to other parts of the system. There are several security questions that should, potentially, be addressed, and the following list summarizes some of them. This list is not exhaustive, but is intended to illustrate the complexity of the problem.

1. What are the security goals of the system? What assumptions does the design make, and are those assumptions explicitly and precisely stated and recorded? What is the system's trusted computing base (TCB)[38, 29], the parts of the system that have to work correctly to make the system secure?

2. What are the principals[1], and how are their messages authenticated?

[1] described in Section 2.4 and defined in Section 2.5

3. How are authorizations specified? With what granularity can they be specified? How should authorizations be granted to valid users? Can a previously-granted authorization be revoked from a user, and, if so, how? Is revocation disruptive to the system? Can an authorization be delegated, or do all users have to go to the same person to be granted the authorization? Can a user pass on an authorization he has received to other users without the knowledge/permission of the person from whom he originally received the authorization?

4. How are access control lists (ACLs) stored, processed, and updated securely?

5. Can groups be created and used on ACLs? How are users added and deleted from groups? How are groups audited?

6. If the system uses cryptographic keys, how are these keys protected, managed, and distributed? What specific functions are these keys intended to perform, and precisely what do the keys represent? Who generates these keys? How does the system recover if a key is compromised, or lost?

7. Is the confidentiality of messages assured, or can an adversary learn sensitive information as messages travel over the network? Is the integrity of messages assured, or can an adversary modify messages as they are travelling over the network (even if he may not be able to read them), and still have them be accepted as authentic?

8. Can an adversary deny service to a valid user, even though the adversary himself may not be able to access the protected resource?

9. What naming architecture does the system use? Is it user-friendly and in harmony with the security design?

10. Which information is logged? How are resources audited? Can users be held accountable for their actions?

11. Is individual privacy protected by the system? If information is logged, who is logging this information, and how are logs stored and protected? Does the

system have mechanisms to prevent or discourage abuse? (Privacy and auditing are competing goals.)

12. Is the integrity of devices assured? Can an adversary insert trojan devices into the system, or discretely tamper with hardware in the system's TCB?

13. Is the system scalable? Does the system scale well as more users and computers are added?

14. Does the system detect intrusions and violations to its protection mechanisms, such as viruses? How does the system react and recover?

15. How much damage can a disgruntled or treacherous user, or a malfunctioning computer, do to the system?

16. Is the system user-friendly? How much does a typical user need to know and understand to use the system in such a way that he/she does not cause security breaches?

17. Who "owns" the system? Who is responsible for maintaining it? How easy is it to support, upgrade and extend?

The goal of secure systems is to protect resources and information. Systems are designed to allow access by authorized users, and prevent access by unauthorized users. The latter requirement is a *negative goal*. It requires checking that all the ways in which an adversary may try to access the resource or information are blocked. Negative goals are difficult to achieve as they require careful reasoning and analysis of all possible scenarios. Furthermore, it is difficult to ascertain if the system achieves the negative goal in practice, as there may be no direct feedback: whereas a valid user will let you know if he cannot access a resource, an adversary may not let you know if he has been able to access the resource. To successfully design and implement secure systems, a paranoid approach, in which the architects and engineers consider everything that can go wrong, must be adopted. This approach, in which all possible scenarios are considered and paranoid decisions are made, is called the "*safety net*"[2] approach.

There is no "silver bullet", no established list of rules which, if adhered to, will guarantee that the system is secure. Systems with security should have security integrated into their designs, instead of being added on after most of the design is completed. To aid in the development of secure systems, several design principles have been established. Examples of these principles are described in detail in [38], and are summarized below:

1. Keep the design as *simple* and as *small* as possible. This makes it easier to check if the goals of the design have been accomplished.

2. *Make all assumptions explicit.* This encourages careful analysis of the protection mechanisms in the system, and makes it easier to ascertain where the faults are if the system fails.

3. The system should provide immediate *feedback* when assumptions are violated or errors occur. Architects should then review their design, and reiterate the design and implementation to fix the problems. The design process should facilitate reiterations.

4. Design protection mechanisms to *default to lack of access.* Thus, if a situation which was not anticipated arises, the controls should deny access to the user/adversary. The alternative philosophy of granting access to everyone, then determining why a user should be denied access is the wrong approach and can result in loopholes and backdoors in the system.

5. Every program and user should operate with the *least set of privileges* required to do their job. This limits the damage that any particular user/program can do. It also minimizes the violations, intended and unintended, that can result from sharing between programs and users.

[2] Jerome H. Saltzer, M. Frans Kaashoek. Topics in the Engineering of Computer Systems. M.I.T. 6.033 class notes, draft release 1.10. 6 February 2001, page 6-13

6. *Minimize the number of mechanisms which are shared* between programs and users. Each shared mechanism represents a potential information path from one user to another through which information can flow from a user who was allowed to access the information to a user who was not allowed to access that particular information.

7. *Clearly define the requirements, and try to minimize and simplify the TCB.* The TCB, trusted computing base[38, 29], consists of the parts of the system that have to work correctly to make the system secure. By clearly specifying the TCB, designers will know the attacks their system is protected against, the attacks it is vulnerable to, and whether their assumptions are reasonable. If a component outside of the TCB fails, the system might deny access it should have granted, but it must not grant access it should have denied. Using digitally signed certificates is an example of minimizing the TCB. Certificates are tamper-resistant documents and can be stored in untrusted places; if a certificate has been tampered with, it will not verify, and will be ignored when access control[3] decisions are made.

8. *Completely mediate* every request for access to a protected resource and check it for authority. "It implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated."[4]

9. Make the system *easy for a typical user to use and understand.* If the system is complex to use and understand, users will make mistakes which could result in security breaches.

10. The design should be *open.* The security of the design should not depend on

[3] described in Section 2.5

[4] Jerome H. Saltzer, M. Frans Kaashoek. Topics in the Engineering of Computer Systems. M.I.T. 6.033 class notes, draft release 1.10. 6 February 2001, page 6-15

the secrecy of the design, and should depend only on the secrecy of keys and passwords. It is much easier to keep keys and passwords secret than it is to keep a design secret, especially if the implementation is going to have widespread use. Furthermore, if the design is open, it is easier for many reviewers, and skeptical users, to evaluate the design to determine if it meets its goals. An open design is less susceptible to errors, as more people have the opportunity to analyze the design and give feedback on it. Of course, the major problem with open designs is that once an open design has been deemed to be secure, competitors can then use it to secure their own resources. Thus, with rare exception, companies and government agencies do not make the designs of their security systems public.

2.1.1 SPKI/SDSI Contribution

SPKI/SDSI is designed to facilitate the development of scalable, secure, distributed computing systems. It provides fine-grained access control, using a local name space architecture, and a simple, flexible, trust policy model. In particular, it addresses the specific issues regarding:

1. specifying, granting, delegating and revoking authorizations
2. creating, maintaining and auditing groups
3. naming
4. facilitating scalability
5. designing simple, user-friendly systems.

2.2 Confidentiality

The problem of security usually requires the examination of the following three issues:

Authentication Verifying i) the sender of the message and, ii) whether the message that was received is the same as the message that was sent (i.e. detecting whether the message has been altered as it travelled from the sender to the recipient). In this publication , the sender of a message refers to the message's *originator.*

Authorization Determining if a particular principal is allowed to perform the operation it is requesting to perform.

Confidentiality Sending messages sealed in such a way that only the intended recipient, and sender, can read the messages.

Confidentiality is described in this section, and authentication and authorization are described in the following two sections. It should be noted that, of the three, authorization is the main focus of this thesis. Moreover, this thesis does not specifically discuss methods and protocols for protecting privacy in a secure system. Privacy is defined as the "ability of an individual (or organization) to decide whether, when, and to whom personal (or organizational) information is released."[5]

If two parties want to communicate *privately* over an untrusted network, they can encrypt their messages before sending them, transforming *plaintexts*, which are messages anyone can read, into *ciphertexts*, which are messages no one can read. The intended recipients of the ciphertexts would then decrypt them back into plaintexts. Operations used for encrypting and decrypting use cryptographic keys. The results of encrypting and decrypting are dependent on the input message and the key used. The same plaintext encrypts to different ciphertext under different keys. A cryptographic system (cryptosystem) in which the keys that are used for encryption and decryption must be kept secret is called a *symmetric-key cryptosystem.* A cryptosystem in which the key that is used for decryption must be kept secret, but the key that is used for encryption can be public, is called a *public-key cryptosystem* or *asymmetric-key cryptosystem.*

Symmetric-key cryptosystems are the conventional cryptographic systems. The message's sender and recipient use the same key which both keep secret. The message

[5] Jerome H. Saltzer, M. Frans Kaashoek. Topics in the Engineering of Computer Systems. M.I.T. 6.033 class notes, draft release 1.10. 6 February 2001, page 6-8

is encrypted with the key, the ciphertext is sent over an untrusted network to the recipient, and the recipient decrypts the ciphertext with the same key to retrieve the original message.

Public-key cryptosystems were invented in the late 1970s. In these cryptosystems, each person generates two keys: a public key and a private key. These keys have the following properties:

1. Each user generates his own, distinct pair of keys. One of these keys, K_E, say, is a "public" key, which would be distributed to other users. The other key, K_D, say, is a "private" key, which the user keeps secret, and which only he would use. The user that generates the public-private key pair is referred to as the *keyholder* of those keys.

2. K_E is used to encrypt a message. K_D is used to decrypt the corresponding ciphertext. Anyone can encrypt a message. Since only the keyholder has access to K_D, only the keyholder can decrypt the corresponding ciphertext.

3. K_D is used to digitally sign a message. K_E is used to verify the corresponding digital signature. Only the keyholder can form a specific digital signature using K_D. Anyone can verify the signature.

4. Given K_E, it is very difficult to determine K_D. Thus, K_E, the public key, can be distributed without fear of compromise of K_D, the private key.

To send an encrypted message to a user, Alice, say, one would obtain a copy of her public key, K_{E_A}, say, encrypt the message with it, and send it to her. Anyone can obtain a copy of Alice's public key, thus, anyone can send an encrypted message to Alice. As Alice possesses the only key that can decrypt the message, K_{D_A} (her private key), she is the only one who will be able to decode it. Anyone else who tries to read or decrypt the message will get something unintelligible.

2.3 Authentication

The problem of authentication over untrusted networks has two parts:

i. Authenticating the *sender (originator)* of the message: is the person who is claiming to have sent the message actually the person who sent it?

ii. Authenticating the *integrity* of the contents of the message (data integrity): is the message that was received the same as the message that was sent?

To authenticate messages, the sender computes an authentication tag on the message, and appends it onto the message. This authentication tag must be both dependent on the key that was used to create it, and the message from which it was formed. If the tag is not dependent on the sender's key, it cannot be used to authenticate the sender of the message. If the tag is not dependent on the message, it cannot be used to verify the integrity of the message: an adversary can take the tag on one message, and append it to another message.

In symmetric-key cryptosystems, the message's tag is computed using the sender's symmetric key. As the recipient shares the same key, the recipient recomputes the tag using the message and the recipient's key, then compares the tag on the message he received with the recomputed tag. If the two tags are the same, the message is authentic; otherwise, the message is not authentic. An authentication tag computed with a symmetric key is called a *message authentication code* or *MAC*.

In public-key cryptosystems, the message's tag is computed using the sender's private key. The recipient does not have a copy of the private key, and thus cannot recompute the tag. Instead, the recipient uses a copy of the sender's public key to check the authenticity of the message. He runs a verification procedure on the message, the appended tag, and the sender's public key. If the procedure returns true, the message is authentic; otherwise, it is not authentic. An authentication tag computed with a private key is called a *digital signature*.

Digital signatures can be checked by anyone, as the verification key, the sender's public key, is publicly available. MACs can only be checked by a person who has a copy of the sender's symmetric, secret key: only the message's sender and the recipient are supposed to have copies of this key. *Assuming no one but the sender controls his private key*, the sender of a message authenticated with a digital signature cannot repudiate (disown) the message, because only he controls the key that created the signature: digital signatures provide *non-repudiation*. On the other hand, the sender of a message authenticated with a MAC can repudiate message, and claim that it originated from the receiving party, who also has a copy of the secret key that created the MAC.

2.3.1 Confidentiality vs. Authentication

Confidentiality and authentication are *orthogonal*. Confidentiality ensures the privacy of the message's contents. Authentication verifies the message's sender and the integrity of the message's contents. In practice, people either send messages that are both encrypted and authenticated, or just authenticated if privacy of the message's contents is not a priority. There are several examples where the authenticity of the message is important, but its confidentiality is not. For example, if I get an email from my class's professor stating that the test that I have tomorrow has been cancelled, I especially care about the authenticity of the email, but may not care if others are able to read it. Confidentiality without authentication is not really useful: it does not really matter if no one was able to read the message if one cannot really be sure of the authenticity of the message.

If a user wants to send an authentic, private message, typically, he would first compute the authentication tag on the message, append it onto the plaintext message, then encrypt the message and appended tag. The result of the encryption is then transmitted over the wire. (This is akin to a person signing a paper letter, then putting it into a thick paper envelope, and mailing this to the letter's recipient). If he did the cryptographic operations in the other order, and encrypted the message first, then computed the authentication tag on the ciphertext and appended it onto the ciphertext, anyone would be able to verify the signature on the message (and, of course, only the person with the appropriate key would be able to decrypt and read the message).

Note that the techniques used to ensure confidentiality and the techniques used to ensure authentication have different requirements, and rely on different assumptions. For example, for a particular public-key cryptosystem:

i. the requirements on the operations used to encrypt and decrypt could be:

   a. Given a message, its ciphertext, and the public key, it is difficult to find the private key

   b. Given a ciphertext, and the public key, it is difficult to find any part of the original message.

ii. the requirements on the operations used for signing and verifying could be:

   a. Given a message, its authentication tag, and the public key, it is difficult to find the private key

   b. Given a message and the public key, it is difficult to construct an authentication tag such that the verification procedure returns true.

These requirements are necessary for any public-key cryptosystem, but are not sufficient for every public-key cryptosystem[6]. Now, consider the following proposal for an authentication scheme:

1. Alice wants to send an authenticated message to Bob. She begins the message with the phrase 'Hi Bob', and encrypts the message with her private key. She sends the encrypted message to Bob.

2. Bob verifies the authenticity of the message he receives by decrypting the message with an authentic copy of Alice's public key, and checking to see if the result of the decryption begins with 'Hi Bob'. If it does, the message is authentic. If it does not, it is not authentic.

[6] There are stricter notions of security, such as semantic security, described in [25].

This scheme does not work. The problem with it is that an adversary who knows the design of this authentication scheme, and knows the design of the cryptosystem, may be able to alter one or more bits of the ciphertext such that the result of the decryption still begins with ‘Hi Bob’, but is not the same message that Alice originally sent. Depending on the cryptosystem, the adversary may simply have to flip bits beyond the beginning of the message to successfully attack the authentication scheme. When Bob decrypts the message, it will still begin with ‘Hi Bob’ and thus ‘authenticate’, but the message Bob receives is not the same message Alice sent. From the requirements stated previously, the operations used for encrypting and decrypting do not guarantee that, if any bit in the transmitted bits is altered, the message will not authenticate. However, this is a requirement of the signing/verifying operations (it is part of requirement ii.b), and thus, if those operations had been used, the attack would not have worked.

2.4 Authorization

The protection model that SPKI/SDSI provides is called the *list-oriented guard model*[38, 29, 30]. In the basic *guard model*, there is an impenetrable room with a single door. The room contains the *object* (i.e. information or resource) that is being protected. There is a *guardian* at the door which checks the credentials of each *principal* requesting access to the object. The guardian is responsible for protecting the object and allows a principal to enter the room only if it has determined that the principal's particular request to access the object should be honored.

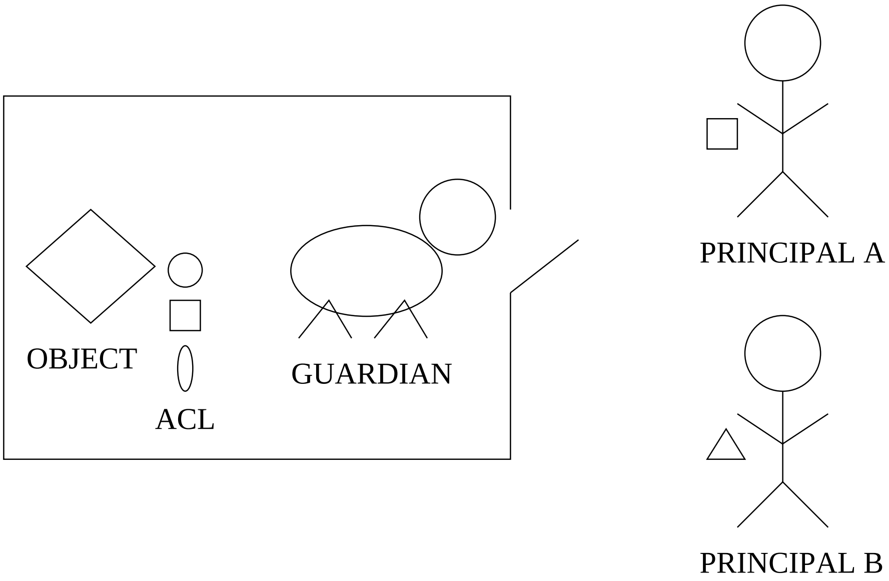
Authorization addresses the question “is this principal allowed to perform the operation it is requesting to perform?” To answer this question, the guardian compares some token of information that it has with information that the requestor has or knows. There are three key aspects to authorization: i) *granting* authorization to a principal, ii) *revoking* a previously-issued authorization from a principal, and iii) *checking* whether a principal has valid authorization when it requests access to the object.

Continuing with the *list-oriented guard model (list model)*, in this model, each principal has a distinct token. The guardian maintains a list of tokens of those who are allowed to enter the room. This list is called an *access control list (ACL)*, and is usually attached to the object. When a principal requests access to the object, it presents its token to the guardian. The guardian checks that the token is on the object's ACL, and, if it is, allows the principal to enter the room. If the token is not on the ACL, the principal's request is rejected. Revocation is done by the guard removing the relevant principal's token from the ACL. Groups consisting of multiple principals can also be formed, and specified on ACLs. If a group appears on an ACL, all principals who are members of that group are allowed to access the object. When groups are used, revoking a principal's authorization entails removing the principal from the group. The list-oriented guard model is illustrated in Figure 2-1 on page 34.

An important issue in designing secure systems is determining how principals are granted their authorizations. In SPKI/SDSI, each principal's authorization to access an object originates directly from the guardian of the object. The principal is granted authorizations on the basis of its qualities, such as being a bona fide member of a particular group. The infrastructure's authorization model also features a clean model for delegation of authority, in which one principal can delegate to another some authority that the first principal has. With delegation, the principal, or set of principals, most suitable for determining if a principal should be authorized to access a particular protected object can be easily, and securely, granted the right to do so.

For the sake of comparison with the list model, the *ticket-oriented guard model (ticket model)*[38], will be briefly described. In the ticket model, each guard has one token (ticket/capability). Each principal has a list of tickets, a ticket for each different object it is authorized to access. To determine if a principal's request to access the protected object should be honored, the guard checks that the ticket the principal presents is the same as the ticket it has. If it is, the principal is allowed to enter the room; if it is not, the principal's request is rejected.

Compared to the list model, the access check in the ticket model is easier and faster, as the guardian does not have to iterate through a list of tokens. However,



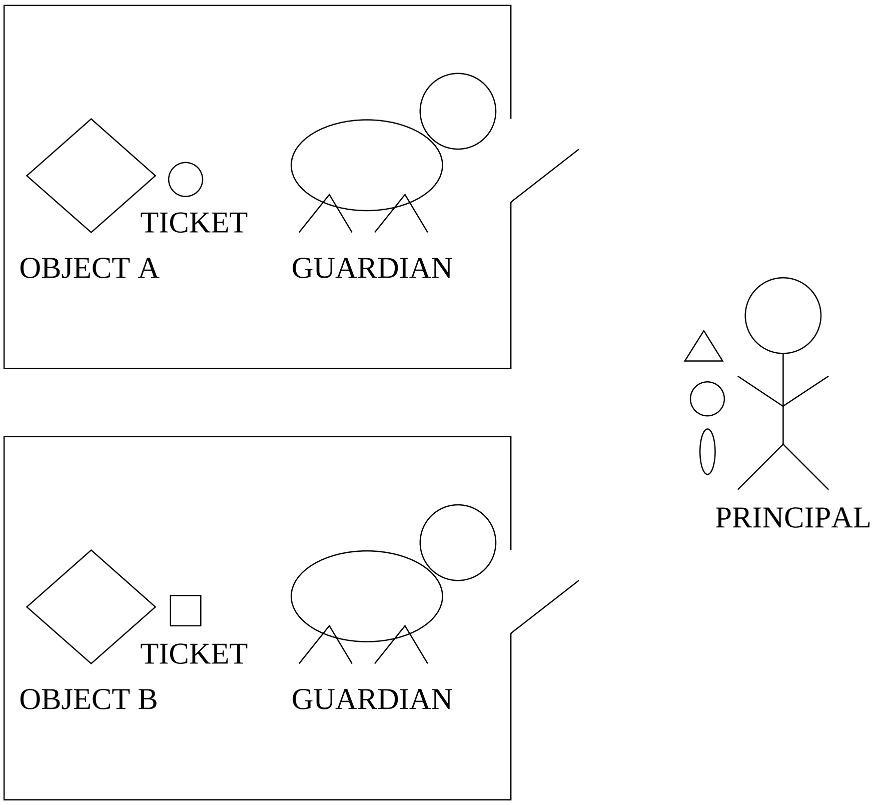
Principal A is authorized to access the object.
Principal B is not authorized to access the object.

Figure 2-1: List-oriented Guard Model

revocation is more difficult. To revoke a principal's authorization in the ticket model, the system must either hunt down the principal and take away his ticket, or change the guard's ticket and reissue new tickets to all the principals who are still authorized. Both methods are disruptive. Common examples of systems that use the ticket model are locked doors that open with specific keys, and the access control system at movie theaters. The ticket-oriented guard model is illustrated in Figure 2-2 on page 35. There are other protection models besides the list and tickets models, and some of the more commons ones are described in the course notes by Prof. Saltzer and Prof. Kaashoek [38].

2.5 Access Control

In the protection models described in the previous section, it is the responsibility of the guardian to provide *access control* to the object it is protecting. The guardian is



The principal is authorized to access object A, but is not authorized to access object B.

Figure 2-2: Ticket-oriented Guard Model

supposed to completely mediate every request before honoring it. This entails first authenticating the principal, then determining if the principal making the request is authorized to perform the requested action. The two operations are linked in that the principal used in the authorization decision must be the same principal that is adjudged to have made the request. In SPKI/SDSI, principals are public keys, and they are authenticated via digital signatures. SPKI/SDSI signatures contain the public key corresponding to the private key that created the signature. Client requests are signed and are accompanied with SPKI/SDSI certificate chains. SPKI/SDSI ACLs are kept on the objects they are protecting. The guardian first determines the authenticity of the request by verifying the signature on the request, then determines if the certificate chain provides a valid proof of authorization from the guardian, via the particular object's ACL, to the public key (principal) that was used to verify the request's signature (the object's ACL could have the public key directly on it, in which case, a certificate chain need not be presented).

2.6 Definitions

Terminology used in this thesis, that has not yet been defined, is defined in this section:

- *user*: a physical human; the actual user
- *client*: the computer/software from which the user generates requests.
- *principal*: the unit of accountability in a computer system. A principal is a participant in a protocol, and is the entity to which authorizations are granted. Principals are often referred to as "Alice", "Bob", "Carol", etc. In theoretical protocol analyses, a principal is usually an abstraction for both a user and the computer the user is using.
- *secure communication channel:* A user meeting another user face-to-face, and giving him a floppy disk with sensitive data is an example of one user communicating with another over a secure channel. A secure communication channel between two users is a channel which provides confidentiality and authentication, and over which there is protection from impersonation attacks, replay attacks, and other attacks against security protocols.
- *"public key that signs ..."*: Public keys are not used to create digital signatures. This phrase is shorthand for "public key corresponding to the private key that signs ... ".
- $\{M\}_{K_D}$: refers to signing the message M with private key K_D. (This notation is used in Figure 4-1 on page 88.)

Chapter 3

Background

3.1 Public-Key Cryptography

3.1.1 Whitfield Diffie and Martin Hellman

In 1976, Whitfield Diffie and Martin Hellman, a researcher and a professor at Stanford respectively, developed the theory of the public-key cryptosystem[6]. This development was significant, as it facilitated secure communication across insecure, untrusted, computer-controlled networks.

Before public-key cryptography, for two parties, Alice and Bob, say, to communicate securely, they had to exchange a secret key in advance via some trusted secure communication channel. Typically, the two parties would meet in person to exchange the key, or use a trusted courier to transfer the secret key from one party to another. Alice and Bob would both use this key to encrypt, decrypt, MAC, and verify MACs on transmissions between them. These systems are described as *symmetric-key cryptosystems*, as the two parties share a single key (symmetric key) for their secure communication.

The problems with using symmetric-key cryptosystems to secure transmissions in computer networks are severe enough to render the model impractical for such systems. One major problem is that of securely distributing the symmetric keys. In order to communicate privately and authentically, two parties must establish a secret key between them, which can be difficult and inconvenient if the distance between them is significant.

Another major problem in symmetric-key cryptosystems is that proving the true authenticity of a message may be impossible. If both Alice and Bob share the same symmetric key, K_{AB}, say, then messages received MAC'd with K_{AB} could have originated either from Alice or Bob. The authenticity of the messages is bilateral, and the *threat of dispute* exists. When Alice sends a message MAC'd with K_{AB}, Alice could deny that the message originated from her and claim that it is Bob's message. Bob could forge a MAC'd message and claim it originated from Alice. Also, if Alice wants to protect the authenticity and privacy of her messages to Bob, she has to be concerned about the threat of key compromise on her system, as well as Bob's system, over which she has no control. The secret key which Alice uses to secure her communication with Bob is kept on two computers, and thus, there are two points from which an adversary can attack the secret key. If Bob's system is surreptitiously compromised by a malicious player, Mallory, say, Mallory can send messages to Bob pretending to be Alice. Of course, Bob has similar issues if he is using MACs to authenticate his messages to Alice. The fundamental issue is that, even though Alice may be able to convince herself that a particular message originated from Bob, she is not able to prove to a third party, like a judge, that the message is Bob's message, and that she did not forge it herself. These problems render symmetric-key cryptosystems undesirable for use in systems using legally-binding documents such as contracts, bills and receipts. Such systems require digital authentication mechanisms that can produce unforgeable, message-dependent signatures. It must be easy for anyone to verify that a signature is authentic, but computationally infeasible for anyone other than the legitimate signer to produce it. Since bits can be copied precisely, digital signatures must be message-dependent, otherwise one could take a user's signature on one message and claim that it is the signature on a different message. Finally, MACs can only be verified by a user who has a copy of the sender's symmetric, secret key: only two users, the message's sender and the message's recipient, are supposed to have copies of this key. To provide a model which is scalable, the authentication scheme should facilitate the verification of a message's authenticity by anyone.

In 1976, Diffie and Hellman proposed the idea of developing a cryptosystem which used a pair of keys, instead of one key, to encrypt, decrypt and authenticate messages. These keys would have the following properties:

1. Each user would generate his own, distinct pair of keys. One of these keys, K_E, say, would be a "public" key, and this would be distributed to other users. The other key, K_D, say, would be a "private" key, which the user would keep secret, and which only he would use. The user that generates the public-private key pair is referred to as the *keyholder* of those keys.

2. K_E is used to encrypt a message. K_D is used to decrypt the corresponding ciphertext. Anyone can encrypt a message. Since only the keyholder has access to K_D, only the keyholder can decrypt the corresponding ciphertext.

3. K_D is used to digitally sign a message. K_E is used to verify the corresponding digital signature. Only the keyholder can form a specific digital signature using K_D. Anyone can verify the signature.

4. Given K_E, it is very difficult to determine K_D.

Because of the 4^{th} property, K_E, the public key can be distributed without fear of compromise of K_D, the private key. For example, a user could publish his public key in his public directory on his filesystem, with world-readable read permissions and user-only write permissions; anyone will be able to read the public key, but not be able to modify it. The private key would be kept in a separate directory that can be accessed by only the user. Because each user uses a pair of keys, a public one and a private one, these systems are described as *asymmetric* or *public-key cryptosystems.*

To send an encrypted message to user Alice, one would obtain a copy of her public key, K_{E_A}, say, encrypt the message with it, and send it to her. Anyone can obtain a copy of Alice's public key, thus, anyone can send an encrypted message to Alice. As Alice possesses the only key that can decrypt the message, K_{D_A} (her private key), she is the only one who will be able to decode it. Anyone else who tries to read or decrypt the message will get something unintelligible.

Furthermore, Alice can also digitally sign a message to authenticate it as coming from her alone. She would create a digital signature using K_{D_A} and the message, and transmit the original message, with the digital signature appended, to her correspondent, Bob, say. (She can further encrypt this message-signature pair with Bob's public key, K_{E_B}, if she wanted to keep the message private.) Bob can verify that the message originated from Alice by obtaining a copy of her public key, (decrypting the message with K_{D_B} if it was encrypted) and verifying the attached signature with K_{E_A}. If the signature successfully verifies, and Bob is sure that he used the correct public key, he can be sure that the message originated from Alice. Furthermore, he can convince a third party of this as well. Alice is the only principal who possesses the private key that created the digital signature. If the message was not authentic, and had been forged or originated from someone else, when the signature is verified with Alice's public key, the verification procedure would fail. If Bob, or anyone else, wanted to forge a message from Alice, they would have to somehow obtain a copy of her private key, to which only she has access. Thus, a digital signature provides non-repudiation, *assuming no one but Alice controls Alice's private key.* Also, Alice's digital signatures could be checked by anyone, as anyone can obtain an authentic copy of Alice's public key. If Alice and Bob were using a symmetric-key cryptosystem, only Bob could check the authenticity of Alice's messages, and vice-versa.

Note that in a symmetric-key cryptosystem, if n people wanted to establish pairwise confidential communication channels, $\binom{n}{2} = (n^2 - n)/2$ distinct keys would be needed. In a public-key cryptosystem, n distinct key-pairs would be needed for n people to establish pairwise confidential and authentic communication channels. (If the n people exchange their keys by meeting face-to-face, say, $\binom{n}{2}$ meetings would need to take place, irrespective of whether the people are using symmetric keys or public keys.)

Public-key cryptosystems addressed the key-distribution problem. Each user would generate his/her own key-pair and distribute the public key. No separate private, authentic communication is needed to distribute the keys: they can be distributed over public, authentic communication channels. The channels still need to be authentic as, before Alice can begin communicating securely with Bob, she needs to be sure that she has obtained and is using the public key belonging to Bob (thus, instead of trusted couriers with leather briefcases being used to distribute keys, as is the case in symmetric-key cryptography, in public-key cryptography trusted couriers with clear plastic briefcases can be used to distribute keys). Public-key cryptosystems also addressed the authentication problem. It was possible, theoretically, to prove and verify the source of messages. The system also scales well: each user has to generate only one key-pair which enables several other users to encrypt/authenticate his messages: if n people want to communicate privately, n key-pairs need to be generated. Finally, assuming each user only keeps one copy of his private key, which he should, there is only one point (computer) from which a user's secret key may be attacked.

Diffie and Hellman made a ground-breaking contribution with their proposal of the concept of a public-key cryptosystem. In their paper, they also proposed an elegant algorithm, based on the difficulty of calculating discrete logarithms in a finite field, that could be used for key agreement (two parties could use it to establish a secret key). However, it could not be used to encrypt/decrypt or authenticate messages, and their paper left implementing a complete public-key cryptosystem as an open problem. The algorithm they presented is referred to as the Diffie-Hellman Key-Exchange Algorithm.

3.1.2 RSA Algorithm

In 1977, Ron Rivest, Adi Shamir, and Len Adleman, professors at MIT, developed the RSA public-key cryptosystem[43]. They were motivated by the Diffie-Hellman paper, and wanted to develop a complete public-key cryptosystem that achieved the goals of being able to securely distribute keys, encrypt/decrypt messages, and authenticate messages. Their cryptosystem is based on the assumption that finding large prime numbers is easy, while factoring the product of two large primes is very difficult. The RSA algorithm is also very simple and elegant, and is the most popular public-key cryptosystem in use today.[1]

The RSA cryptosystem is conjectured to be a “trap-door one-way permutation”. It is a bijection, i.e. a function that is both “one-to-one” and “onto”. The system is called “one-way” because it is easy to compute in the forward direction, but very difficult to invert. It is called “trap-door” as inverting the function is actually easy to do once, and only when, certain private “trap-door” information is known; without this information, the function remains difficult to invert. The “trap-door” and “one-way” properties are necessary for encryption. If Bob wanted to send a private message to Alice, he would use Alice’s ‘function’ to encrypt the message by computing the function on the message; only Alice would be able to read the message as only she would possess the “trap-door” information needed to decrypt the ciphertext. Furthermore, Alice’s ‘function’ can be publicly distributed without fear of compromise of her private trap-door information. The RSA system is a permutation as a message can be encrypted with a public key then decrypted with a private key to retrieve the original message, and vice-versa: “every message is the ciphertext for some other message and every ciphertext is itself a permissible message.”[2] The permutation property is useful to also be able to use the cryptosystem for authentication. In practice, RSA cannot be used by itself to guarantee confidentiality or authentication. It is augmented with other operations to produce procedures that can be used to achieve either confidentiality or authentication.

One common use of a public-key cryptosystem is to bootstrap into a symmetric-key cryptosystem. The public-key cryptosystem is used to set up a secure communication channel over which symmetric keys can be established. The shared symmetric-key is then used for future communications between the two parties. This is useful because, whereas key distribution and authentication are simpler with public-key systems, encryption and decryption with symmetric-key systems are much faster than encryption

[1] [8, 22] contain good anecdotes of the history behind and the development of RSA.

[2] R. L. Rivest, A. Shamir, and L. Adleman. A Method For Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2), Feb. 1978, page 121. See `http://theory.lcs.mit.edu/~rivest/rsapaper.pdf`. Last visited 08/07/2001.

and decryption with public-key systems[3].

3.2 Digital Certificates

In 1978, Loren Kohnfelder, an MIT student supervised by Len Adleman, published his bachelor's thesis entitled "Towards a Practical Public-Key Cryptosystem"[27]. He studied approaches that could be used to effectively adopt a public-key cryptosystem, in particular, the RSA cryptosystem. One problem he focused on was that public keys must be obtained authentically before secure communication can begin. When Bob sends Alice his public key over a potentially insecure channel, how can Alice trust that the public key that she receives is actually Bob's? Mallory, acting as a *person-in-the-middle*[4], could have intercepted this transaction, and replaced Bob's public key with her own. This problem, theoretically, did not exist in traditional cryptography because, in traditional cryptography, keys are transmitted over private, authentic channels. Thus, though in public-key cryptography "the enemy may eavesdrop on the key transmission channel, the key must be sent via a channel in such a way that the originator of the transmission is reliably known."[5] Since, in public-key cryptography, "keys are typically transmitted over public channels (if secure channels were available, traditional cryptography methods could be used), the origin of key transmission is often questionable."[6] Symmetric keys need to be distributed in a way that is both secret and authentic; public keys need to be distributed in a way that is authentic, and do not need to be kept secret.

Diffie and Hellman approached the problem of authenticating public keys by proposing a "Public File". The Public File would act as a trusted third party, a central authority that would serve as a dynamic directory for all public keys in a given system. When a new user is added to the system, he would register once, securely, with the Public File before beginning communications within the system. As

[3] Symmetric-key encryption is about 1,000 times faster than public-key encryption[36].

[4] defined in Section 4.2

[5] Loren M Kohnfelder. Towards a Practical Public-key Cryptosystem. Bachelor's thesis, EECS Dept., Massachusetts Institute of Technology, May, 1978, page 15

[6] Ibid., page 39

an example, a new user could be required to register in person with a 'central registry service', bringing with him some form of trusted identification, before he could begin communicating with other users in the system. The new user would give the Public File his name and public key. In return, and at the time of registration, the user would receive the Public File's public key, which he would use to authenticate transmissions from the Public File. When Alice, say, would like to communicate with Bob, say, she would send a request, possibly signed, to the Public File, asking for Bob's public key. The Public File would return, in a signed reply, the requested information. This information can easily be authenticated, as Alice would have a copy of the Public File's public key, which was reliably obtained.

A central Public File attempts to solve the key authentication problem, but, as Kohnfelder noted, "it is a great potential threat to system security."[7] An adversary that somehow learns the Public File's private key can do damage by authoritatively passing out bogus public keys. Even if an enemy cannot access the File's private key, damage can be done by tampering with the File's records, which may be easier to break into. A Public File would be a large, expensive, complex system to implement in applications where the rate of updates, requests and responses is very large. The File could well end up being the performance bottleneck in such applications. A Public File would also not work in very high security applications, since it may not be trusted: "Consider a Public File coordinating all diplomatic communications in the world; who could reliably operate such an authority?"[8]

Kohnfelder proposed an alternative approach to dealing with the problem of reliably authenticating public keys. He developed the concept of a *digital certificate*, a file that would be issued by a trusted third party, like the Public File, that would certify public keys. A certificate would essentially be a signed message from the trusted third party that would bind the name of the user to a particular public key. It would be an ordered triple containing the public key, a plaintext name, and an "authenticator".

[7] Loren M Kohnfelder. Towards a Practical Public-key Cryptosystem. Bachelor's thesis, EECS Dept., Massachusetts Institute of Technology, May, 1978, page 16

[8] Ibid., page 16

The “authenticator” is essentially the trusted third party’s signature on all of the other information in the certificate. Only the Public File could create authenticators as only it controlled the private key to do so; any principal could verify authenticators as everyone had a copy of the Public File’s public key.

Thus, the Public File had a new role. In its original design, the File was contacted during registration of a new user, and before communication could be initiated between two parties who had never communicated before. Now, only the first contact was necessary. During registration, the user would present his public key and name, and, after verifying that the name was unique within the system, the Public File would create a certificate for the user, and give it to him. (Certificates are tamper-resistant, and do not have to be distributed securely; a new user could register with the Public File one day, and receive his certificate the next day via email; he would verify his certificate himself, before using it.) During registration, the new user would still be given a copy of the Public File’s public key. After registration, no other communication between a user and the Public File was ever necessary. Before communication, two parties would exchange their certificates over public, unauthenticated channels. They would verify each others certificates using the Public File’s public key, then secure their transmissions using the keys in the certificates. Certificates need not be stored in a secure location, nor be securely distributed: if the certificate is a forgery, it would not verify with the Public File’s public key. In this system, users are trusting the Public File, the trusted third party, to reliably authenticate public keys before issuing certificates.

Kohnfelder noted that, though using certificates does not introduce any new weaknesses into the system, one disadvantage is that it is more difficult to recover when a user’s private key is compromised, that is, discovered by an adversary. When a Public File alone is being used, as soon as a user-key compromise is discovered, the File can be contacted and prevented from distributing that key. Users who had a copy of the key would have to be contacted and told that they should stop using it. This could be facilitated if the Public File kept track of all the users to whom it had distributed the key. In a certificate-based system, certificates are public information, and can be copied and distributed freely from one user to another. Tracking down all instances of a certificate can be difficult. Noting that no public-key system can, in any sense, recover well, from key compromise, Kohnfelder presented three proposals to deal with the problem when certificates are being used. In one solution, all the users in the system are informed when a particular key is compromised. This is adequate for a system which is not too large, as compromised private keys should, in general, be very rare. A second solution is to put expiration dates on certificates. The expiration date must also be included in the information which is signed by the trusted third party. Using expiration dates would mean that new certificates would have to be issued periodically by the Public File, increasing the amount of communication with the Public File. In the third approach, an online check is facilitated as part of the certificate verification process. When a user discovers that his key has been compromised, he immediately notifies the Public File. The Public File would maintain a list of all the certificates which have been cancelled due to security leaks. In the case of suspicious communication, the Public File could be consulted to check if the associated certificate had been revoked. (An alternative to this approach is for the Public File to periodically issue signed copies of this list to users, reducing their communication with the File. These periodically issued lists are called Certificate Revocation Lists (CRLs).)

3.3 Public-Key Infrastructures

Public-key infrastructures (PKIs) consist of the services that are needed to deploy and support technologies based on public keys. They address issues related to certificate authorities, certificate formats, the revocation of certificates, and the security policies under which a public key may be trusted. A fundamental point to note when designing technologies that use public-key cryptography is that the security of private keys is vital, and currently, users typically do not protect their private keys as securely as they should. Private keys are capable of being compromised, and the weakest link in most secure systems is usually between the user and his key. Keys are usually stored on computers and there are several security vulnerabilities to which computers are susceptible, including trojan horses, covert channels, viruses, and account violations. Decisions on whether to trust a particular public key, and whether to trust that a particular public key is authorized to perform the operation it is requesting to perform, should involve the guardian of the protected resource, the entity responsible for protecting the resource.

The claim that digital signatures provide non-repudiation differs in practice from in theory. A common misunderstanding is that a message signed with a private key, accompanied by a certificate binding the corresponding public key to the name Alice, say, provides the assurance that 'Alice signed this message'. This assumption may not necessarily be true, however. In particular, the signed message and certificate provide no assurance about whether:

- the private key invocation that gave rise to the signed message was performed by Alice. The message could have been signed by Mallory who compromised Alice's key.
- the private key invocation that gave rise to the signed message was performed with Alice's free and informed consent.[4]

In addition, depending on the issuer's guidelines and procedures for issuing certificates, the verifier of the signature and certificate must also take care to ensure that the 'Alice' referred to in the certificate is the name of the user that it thinks of as Alice.

In practice, a digital signature and a certificate do not undeniably prove that the certificate's subject (Alice) signed and sent the message. *They only attest that, assuming that neither the private key of 'Alice' nor the private key of the certificate's issuer has been compromised, the original message was signed by the private key of a user, the issuer of the certificate in the past, at the time the certificate was issued, had reason for believing had the name 'Alice'.* There are four reasons that the general statement 'that the message originated from Alice' may not be true:

- The certificate has expired, and thus, is no longer valid. This is the most common case, and usually, Alice simply has to contact the issuer again to renew the certificate.

- The issuer of the certificate incorrectly issued the certificate to a user other than Alice. This can occur if the issuer's methods for issuing certificates are flawed and the adversary successfully convinces the issuer that he/she is Alice. (Certificates can also be used for other reasons besides binding a person's name to his key. For example, as will be seen later, certificates can also be used to define groups, and grant authorizations. With these certificates, it may be the case that a certificate was issued correctly by the issuer, but that information in it suddenly becomes invalid, for example, if Alice's membership in the group has suddenly been revoked, or Alice's employment at a company has suddenly been terminated.)

- The private key of the message's sender, Alice, may have been compromised or lost. This case, and the next, are the most serious. In this case, it means that a guardian will not be able to trust *any* of the certificates that have ever been issued to Alice. Carl Ellison summarizes the issue precisely when he states "if the bond between key and person is broken, no layer of certificates will strengthen it. On the contrary, in this case certificates merely provide a false sense of security to the [recipient]"[9].

- The private key of the issuer may have been compromised or lost. Similar to the previous case, this means that a guardian will not be able to trust *any* of the certificates that have ever been issued by the issuer.

On the positive side, public-key infrastructures are of value in applications where the cost of attacking a user's private key is much more than the value obtained by compromising that key. Similarly, the cost of impersonating another user with

[9] Carl Ellison. Establishing Identity Without Certification Authorities. *6th USENIX Security Symposium*, July 1996. See `http://world.std.com/~cme/usenix.html`. Last visited 08/07/2001.

the intent of being issued a false certificate should be much more than the value obtained by a successful impersonation. For example, certificates issued by typical PC users who protect their private keys by encrypting them with high-entropy (hard-to-guess) passphrases and storing them in private directories on their machines, and who certify users face-to-face, may be useful in providing low-value access control in small communities.

Because of the possibility of key compromise, digital signatures achieve non-repudiation in theory, but may not achieve it in practice.

> "Non-repudiation may be impossible, with current hardware and environments and is likely to remain so into the future. The loss of non-repudiation may rob certificates of their value as evidence against a key-holder but not of their value in granting access. That is, a private key and the certificate that empowers it may be viewed as a brass house key. That brass key might allow Joe Smith to enter a building, but knowing that the key was given to Joe Smith does not mean that only Joe Smith could have used it. That brass key has value to Joe and not to someone accusing Joe."[10]

It is important to understand that "certificates aren't like some magic security elixir, where you can just add a drop to your system and it will become secure. Certificates must be used properly if you want security."[11]

3.3.1 X.509

X.509[26] is the conventional Public-Key Infrastructure. It is an ISO (International Organization for Standardization) standard and has evolved through 3 different versions, with the first version developed in the late 80s. It has a hierarchical global

[10] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *RFC 2693: SPKI Certificate Theory.* The Internet Society. September 1999. See ftp://ftp.isi.edu/in-notes/rfc2693.txt. Last visited 08/07/2001.

[11] C. Ellison and B. Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. See http://www.counterpane.com/pki-risks.pdf. Last visited 08/07/2001.

namespace. Each name must be globally unique, and the names are referred to as *distinguished names.* A distinguished name (DN) is assigned logically using a tree structure called the *Directory Information Tree* (DIT). The DIT has a single root. Each vertex (except the root) has a distinguished name which is constructed by joining the distinguished name of its parent in the tree with the entry's *relative distinguished name* (RDN), a name which distinguishes it uniquely from other entries with the same parent vertex. X.509 certificates natively bind distinguished names to public keys. Assuming each user has a single public-private key pair, the name-to-key binding is a single-valued function: each name is bound to exactly one key. A key may be bound to more than one name if the keyholder is issued more than one certificate by different certificate authorities. As a distinguished name is intended to identify a single user, X.509 certificates are commonly referred to as "identity certificates". This term, however, is misleading: an X.509 certificate does not bind a user's identity to a key; it binds a user's name to a key.

X.509 originated from X.500, another ISO standard. X.500 was a global, online, distributed directory service for certificates. Carl Ellison describes the history of X.509 this way: "the X.500 proposal was published [in the late 1980s]. It was to be a global directory of named entities. To tie a public key to some node or sub-directory of that structure, the X.509 certificate was defined. The Subject of such a certificate was a path indicating a node in the X.500 database - a so-called 'Distinguished Name'. The X.500 dream has effectively died but the X.509 certificate has lived on. The distinguished name took the place of a person's name and the certificate was called an 'identity certificate', assumed to bind an identity to a public key."[12]

One issue with global namespaces is that they are inherently unscalable: as the communities get larger, there may be more than one user that could possibly be assigned the same global name; care must be taken to ensure that each user's name is globally unique, and this increases the risk that a user referred to by a particular name

[12] What do you need to know about the person with whom you are doing business? House Science and Technology Subcommittee. Hearing of 28 October 1997: Signatures in a Digital Age. Written testimony of Carl M. Ellison. See `http://world.std.com/~cme/html/congress1.html`. Last visited 08/07/2001.

is not the same user you believe was assigned the name. The user assigned the name 'John Robinson' may be different from the user I associate with 'John Robinson'. If I put the name 'John Robinson' on an ACL, the wrong user is given access to the resource I am trying to protect. Names are defined for human convenience, and, in global name spaces, the requirement that each name must be unique leads to assigning names that eventually end up complex and hard for humans to recognize. This has serious security implications. There are also political and social issues in trying to assign everyone a unique global name.

X.509 certificates are issued by Certificate Authorities (CAs). The certificate authority controls the key-pair that signs the certificate, and is referred to as the certificate's issuer. The public key being bound in the certificate is called the certificate's subject. X.509 CAs usually have strict business rules, documented in their Certification Practices Statements (CPS). Each CA may have a different CPS, stating its legal responsibilities, and the rules under which various certificates are issued.

A standard X.509 certificate includes:

- a public key
- a distinguished name to be associated with the key
- the validity period of the certificate, which indicates when the certificate will expire
- the digital signature of the issuer of the certificate. In the case that this certificate belongs to a top-level CA, i.e. the key specified in the certificate is the CA's key, the issuer signs its own certificate.
- the distinguished name of the issuer of the certificate

The X.509 trust model is *hierarchical.* X.509 communities are built from the top-down, with trust extending from "root" CA keys. The model essentially relies on certificates providing a *chain/flow of authentication* from a trusted CA's key to a user's public key. For example, if Alice receives two valid certificates, one issued by a trusted CA binding the name Bob to key k_1, and the other issued by k_1 binding the name Carol to key k_2, she trusts that k_2 is an authentic copy of Carol's key.

Besides binding names to keys, X.509 certificates can also be used to convey authorization information about the certificate subjects. In Version 3 of the X.509 certificate format, an additional *extensions* field was added to help address some of the problems with deploying Version 1 and 2 of the X.509 standard on a significant scale. The extensions field provides for the addition of any number of optional fields into the certificate. Each field is a triple with the extension type, whether the extension is critical or not, and the value of the extension. The ISO has developed a set of standard extensions, and users may also incorporate their own non-standard extensions. Authorization information can be specified using these fields. For example, the *Subject Directory Attributes*, a standard extension, is used to convey additional information about the subject, such as the subject's position in an organization, to assist a guardian in making a policy decision whether to allow or deny access to the certificate's subject. However, there are reasons why caution should be taken in using X.509 certificates to convey authorization information. If authorization information is specified in an X.509 certificate, the certificate would simultaneously be binding a name to a public key, and conveying authorization information about that key. Often, the entity who is most appropriate for certifying the identity of a keyholder is not appropriate, and is not responsible, for certifying if the keyholder should be authorized to access a particular resource or perform a particular action. "For example, the corporate security department may be the appropriate authority for certifying the identities of persons holding public keys, but the corporate finance office may be the only appropriate authority for certifying permission to sign on behalf of the corporation."[13] Furthermore, the dynamics of the two types of certification may not be compatible. Whereas a user may require only one certificate binding his name to his key, he may be required to be granted a number of different, separate authorizations. Authorization information may only be valid for a short period, like a few months,

[13] Warwick Ford, and Michael S. Baum. *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption.* Prentice Hall PTR, 1997, page 251.

or a few days, or even shorter. The name-to-key binding is typically valid for much longer.

The X.509 standard also describes the concept of Certificate Revocation Lists (CRLs), for invalidating/revoking certificates prior to their expiration dates. This may be a useful feature in the case of key compromise. It is also useful in the case where the information regarding the subject of the certificate is suddenly invalid, for example, when the subject's employment at a company has suddenly been terminated. A revoked certificate is much more suspect than an expired one as it carries the threat of key compromise. Only a certificate's issuer (CA) can revoke an X.509 certificate. A CA's CRL is a periodically-issued, time-stamped, digitally-signed list of revoked, unexpired certificates that have been issued by that CA. The CRL also specifies when the next CRL will be issued. (Once certificates expire, they are removed from the next CRL.) Like certificates, CRLs may be distributed over public channels, as they are signed. CRLs unfortunately do not solve the problems they were designed to solve. There may be considerable, variable, delay between a CA being notified that a certificate needs to be revoked, and the reflection of this need in clients and servers. In any case, the major X.509 security application today, SSL/TLS[42], does not check revocation lists – thus, in actuality, CRLs are near to useless[24].

Perhaps the most patent problem with X.509 is that the standard is difficult to understand and adopt. The data formats are not human-readable, and are expressed in the notation called Abstract Syntax Notation One (ASN.1), an Open Systems Interconnection (OSI) standard. ASN.1 is very powerful, but also very complex. The X.509 PKI is thus, complex to use.

3.3.2 PGP

Pretty Good Privacy (PGP)[35] was developed and first released by Phil Zimmermann in the early 1990s. PGP has two parts: certification and encryption. This discussion will focus exclusively on the certification aspects of PGP.

PGP has the following characteristics:

- It has an egalitarian design. Each public key can issue certificates on the same basis as any other public key. There is no mandatory hierarchical infrastructure, as in X.509, as every public key is a certificate authority.

- PGP implements a fault tolerance mechanism, called the *Web of Trust*, that is designed to compensate for the fact that issuers are not specially protected nor professional.

PGP certificates also bind global names to keys. Assuming each user has a single public-private key pair, the name-to-key binding is a single-valued function: each name is bound to exactly one key. A PGP certificate includes:

- a PGP public key

- a name to be associated with the key. The PGP name space is global, with names generally consisting of users' email addresses.

- the validity period of the certificate, which indicates when the certificate will expire

- the digital signatures of one or more PGP public keys, attesting to the authenticity of the key-user id binding. The first signature on the certificate is that of the private key corresponding to the public key in the certificate. Thus, the user controlling the public key in the certificate first creates and self-signs his own certificate; he is referred to as the certificate's *owner*. Different keys belonging to other users may sign the certificate. Each user that signs the certificate attests that the public key in the certificate belongs to and is controlled by the user with the name in the certificate (the certificate owner). In summary, PGP certificates are always self-signed and may have more than one signature.

(To be more precise, a PGP certificate can specify multiple names/user ids to be bound to a single key, with each of these bindings having its own set of signatures attesting to its validity. Examples of names/user ids include email addresses, official names, and photographs.)

Instead of relying on authentication paths that must end in a trusted CA, PGP users can build authentication paths arbitrarily through the entire worldwide community of PGP users. If Alice decides that she has an authentic copy of Bob's public key, and that she trusts Bob to authenticate other public keys, then, from the set of certificates Bob issues, Alice can increase the number of PGP users with whom she can securely communicate. Similarly, users who have an authentic copy of Alice's key, and trust her to authenticate other public keys can use certificates she has signed to build their individual communities. PGP communities are, thus, built from the bottom-up, in a distributed manner. This trust model is called the *web of trust*.

As certificates are not issued by CAs that have strict business rules and legal responsibilities, and, instead are issued by normal people, they could be more fallible than certificates issued by a professional CA. Under the web of trust, multiple different keyholders sign each certificate. The assumption is that these different keyholders are independent so that even if one of them makes a bad judgement, they won't all do so.

A PGP user stores his public key and the public keys of others in a file on his disk; this file is referred to as his *public keyring*. Stored with each public key are two variables indicating:

- whether the user considers the key to be authentic
- the level of trust that the user places in that particular key for the purpose authenticating other public keys, that is, to act as a "*trusted introducer*"[36].

If the user labels a new key as unauthentic, it is automatically not trusted to act as a trusted introducer. If the user decides that the new key is authentic and labels it as such, he then makes a separate decision as to the level of trust he places in the key to act as a trusted introducer.

To specify the level of trust the user places in a key to act as a *trusted introducer*, a variable specifying the following four levels is used:

- Yes: PGP will automatically accept and use a certificate that is signed by this public key. The public key in the certificate, the one being authenticated, is placed on the public keyring and is automatically assigned the specifier indicating that it is authentic. When adding this new key to the keyring, the PGP software prompts the user to specify the level of trust he places in the key to act as a trusted introducer.

- *Don't know*: the PGP software will prompt the user each time it needs to use this key to authenticate another public key.
- *No*: The PGP software will not use this key to authenticate another public key.
- *Usually*: The user 'marginally' trusts this key to authenticate other keys. The PGP software defaults to requiring two marginally-trusted signatures to authenticate another key.

The PGP *web of trust* model allows any user to act as a certification authority, and issue certificates for any other user. Whether the recipient of the certificate accepts it depends on the level of trust he has in the signing key.

A chain of PGP certificates may be presented to authenticate a key, and, as certificates may be signed by multiple keys, there may be multiple certification paths within the certificate chain via which the particular key may be authenticated. X.509 uses similar authentication chains, but, as each X.509 certificate has one signer, a chain of certificates typically represents one certification path from the CA's key to the key in question.

PGP certificates are revoked when private keys are compromised (or users forget the passwords locking their private keys). In X.509, only the certificate's issuer can revoke a certificate. In PGP, "only the certificate's owner (the holder of its corresponding private key) or someone whom the certificate's owner has designated as a revoker can revoke a PGP certificate. (Designating a revoker is a useful practice, as it's often the loss of the passphrase for the certificate's corresponding private key that leads a PGP user to revoke his or her certificate - a task that is only possible if one has access to the private key.)"[14] As PGP does not use commercial CAs, the

[14] Network Associates, Inc. and its Affiliated Companies. How PGP works. See `http://www.pgpi.org/doc/pgpintro/`. Last visited 08/07/2001.

PKI does not use CRLs for revocation. Typically, to communicate that a certificate has been revoked, a signed note, called a *key revocation certificate*, is posted on PGP certificate servers, and widely distributed to people who have the key on their public keyrings. People wishing to communicate with the affected user, or use the affected key to authenticate other keys, are warned about the hazards of using that public key.

"The *web of trust* model works well for loosely-interacting communities, such as individuals seeking to protect their Internet personal email communications. However, it suffers from the problem of requiring too many individuals to make important decisions regarding trust, leading to a high risk of bad decisions being made in haste or without proper understanding of the consequences."[15] Also, if multiple signatures are required to authenticate one particular key, one should take steps to check the independence of the signers. The different private keys could be controlled by the same user.

3.3.3 SPKI/SDSI

The Simple Distributed Security Infrastructure (SDSI)[46], designed in 1996 by MIT Professors Ron Rivest and Butler Lampson, is a security infrastructure whose principal goal is to facilitate the building of secure, scalable, distributed computing systems. Around the same time, Carl Ellison led an effort to design an infrastructure with a simple, well-defined, flexible authorization model. This infrastructure was named the Simple Public-Key Infrastructure (SPKI)[11, 13]. In 1998, the two designs were merged, and the names joined, to form SPKI/SDSI.

SPKI/SDSI has an egalitarian design. The *principals*[16] *are the public keys* and each public key is a certificate authority. Each principal can issue certificates on the same basis as any other principal. There is no hierarchical global infrastructure. SPKI/SDSI communities can be built from the bottom-up, in a distributed manner,

[15] Warwick Ford, and Michael S. Baum. *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption.* Prentice Hall PTR, 1997. page 277.

[16] described in Section 2.4 and defined in Section 2.5

and do not require a trusted "root". There are two types of certificates in SPKI/SDSI: name certificates and authorization certificates. A name certificate defines a local name in the certificate issuer's local name space, and an authorization certificate grants a specific authorization from the certificate's issuer to the certificate's subject. To help keep the infrastructure simple, a single certificate cannot both define a name and grant an authorization: i.e. each certificate is either strictly a name certificate or an authorization certificate.

Naming

A name certificate consists of four fields: the issuer's key, an identifier, the certificate's subject, and a validity specification. An identifier is a single word over some standard alphabet, such as `Alice`, `Bob`, `Friends`, `A`, `B`. In this document, identifiers will be specified in typewriter font. Following are descriptions of these certificate fields:

issuer the public key that signs the certificate.

identifier the identifier determines the local name that is being defined. The name being defined consists of the issuer's key and this identifier. A name certificate, thus, defines a name that consists of a single key followed by a single word. A name consisting of a single key followed by exactly one identifier is referred to as a "*local name*"[3]. Because a name certificate can only define a local name, each principal can only define names within its own name space.

subject the new meaning of the local name being defined. A subject can be a public key or a name consisting of a single public key followed by one or more identifiers. The public key in the subject does not have to be the issuer's key. (If the subject does not explicitly begin with a key, it is implicit that it is a name in the issuer's name space, and thus begins with the issuer's key.)

validity specification usually, the validity specification is a time period during which a certificate is valid, assuming the signature verifies. Beyond this period, the certificate has expired, and should be renewed. The validity specification takes the form (t1, t2), specifying that the certificate is valid from time t1 to time t2, inclusive. The validity specification can also take the form of an on-line check[17] that is performed to determine if the certificate is valid.

Examples of SPKI/SDSI name certificates are given in the appendix and in the SPKI/SDSI IETF drafts and RFCs[11, 12]. SPKI/SDSI name certificates bind local names to public keys. Assuming each user has a single public-private key pair, the name-to-key binding is a multi-valued function: each name is bound to zero, one or more keys. A single name certificate can define a name in the issuer's local name space to be a public key, another name in his/her local name space, or a name in another principal's local name space. A name certificate that defines the local name "K `A`", where K is the issuer's key, to be the subject, "`S`", can be denoted as "K `A` $\longrightarrow$ `S`".

As each principal can issue name certificates, each principal has its own local name space, consisting of the names it defines. SPKI/SDSI, thus, has a local name space architecture, which helps to make the infrastructure scalable: a user does not have to ensure that the names he defines are unique in a global name space; he can define names which are meaningful to him, which he can easily remember and recognize.

Local name spaces are linked when a principal defines a certificate binding a name in his name space to a name in another principal's name space. Figure 3-1 on page 60 gives an example of linking name spaces. In the example, Bob's mother is named Mary Smith and Ms. Smith's key is K_{MARY_SMITH}. Bob issues a certificate defining the name "K_B `Mother`" in his name space to be his mother's public key (certificate 3.1). Alice *links* her name space with Bob's by issuing a certificate defining the name "`Mary_Smith`" in her name space to be a name which consists of Bob's key and the identifier "`Mother`" (certificate 3.2). "K_A `Mary_Smith`" is now indirectly bound to K_{MARY_SMITH}. One advantage of being able to link name spaces is that, if Bob were to change his definition of 'Mother' (because his mother changed her key-pair, say), the principal Alice refers to as `Mary_Smith` in her name space would also automatically change. Thus, a SPKI/SDSI name provides a layer of indirection. If

[17] described in the discussion on the SPKI/SDSI Certificate Guarantee on page 76

Mary Smith were to change her key-pair, the certificates that refer to the local name "K_B `Mother`" (such as certificate 3.2) do not have to be reissued; only certificates that specify Mary Smith's actual public key, K_{MARY_SMITH}, (such as certificate 3.1) need to be reissued.

$$(3.1) \qquad K_B\ \texttt{Mother} \longrightarrow K_{MARY\_SMITH}$$

$$(3.2) \qquad K_A\ \texttt{Mary\_Smith} \longrightarrow K_B\ \texttt{Mother}$$

Figure 3-1: An example of linking local name spaces

A SPKI/SDSI *group* is typically a set of principals. Each group has a name and a set of members. The name is local to some principal, who is the "owner" of the group, and the group owner is the only one who can change the definition of the group. A group definition may explicitly reference the members of the group, or reference other groups (which may even belong to someone else.) To define a group, a group owner simply issues, to each group member, a name certificate defining the local name of the group in the owner's name space to be that member's key or name. A group owner can also add any principal's group to his group by issuing name certificates binding the name of his group to the name of the group being added. Figure 3-2 on page 61 gives an example of a SPKI/SDSI group. In the example, Alice's friends include Bob (K_B), Carol (K_C), and Derek (K_D); she can add them to her group 'friends' by issuing certificate 3.3 to Bob, certificate 3.4 to Carol, and certificate 3.5 to Derek. (She could also have achieved the same effect by issuing the certificates K_A `friends` $\longrightarrow$ K_A `Bob` and K_A `Bob` $\longrightarrow K_B$, K_A `friends` $\longrightarrow$ K_A `Carol` and K_A `Carol` $\longrightarrow K_C$, K_A `friends` $\longrightarrow$ K_A `Derek` and K_A `Derek` $\longrightarrow K_D$ to Bob, Carol and Derek respectively.) Edward (K_E) has named his key "K_E `Edward`" by issuing the certificate K_E `Edward` $\longrightarrow K_E$. Alice adds Edward to her group 'friends' by issuing certificate 3.6 to him. Alice has a sister, Abby (K_{Abby}), and she considers all of Abby's friends to be her friends. She adds them to her group by issuing certificate 3.7

to Abby's friends. Also, Bob's sister's friends are Alice's friends, and she issues certificate 3.8 to them (note that the name "K_A `B C D`" means K_A's `B`'s `C`'s `D`). In summary, with the certificates in Figure 3-2, "K_A `friends`" in Alice's local name space is bound directly to the keys K_B, K_C, K_D, and indirectly to the keys referenced by "K_E `Edward`", "K_{Abby} `friends`" and "K_B `sister friends`".

$$K_A\ \texttt{friends} \longrightarrow K_B \tag{3.3}$$
$$K_A\ \texttt{friends} \longrightarrow K_C \tag{3.4}$$
$$K_A\ \texttt{friends} \longrightarrow K_D \tag{3.5}$$
$$K_A\ \texttt{friends} \longrightarrow K_E\ \texttt{Edward} \tag{3.6}$$
$$K_A\ \texttt{friends} \longrightarrow K_{Abby}\ \texttt{friends} \tag{3.7}$$
$$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{sister friends} \tag{3.8}$$

Figure 3-2: An example of a SPKI/SDSI group: K_A `friends`

The ability to define groups is one of the principal notions of SPKI/SDSI. One advantage of this feature is that it facilitates the easier management of ACLs. If a set of principals with the same characteristics should be granted access to a number of resources, each protected by a separate ACL, a group definition could be made, and the group name placed on each of the ACLs. The ACLs could be updated once, with an entry containing the group name being added onto each of the ACLs. As new members join the group and are issued the relevant certificates, they will be authorized to access the protected resources without having to update the ACLs again. If groups are used, maintaining and updating ACLs is easier and more efficient, as an explicit list of all the principals does not have to be maintained on each of the ACLs. In the example in Figure 3-2, if the name (group) "K_A `friends`" is specified in an entry on an ACL, K_B, K_C, K_D, and all the keys referenced by "K_E `Edward`", "K_{Abby} `friends`" and "K_B `sister friends`" will automatically be authorized to perform the operation specified in that entry's tag. Furthermore, and perhaps more importantly, there can be a *delayed definition* of the group. A group can be added onto an ACL without knowing beforehand the members of the group. ACLs can be updated with an entry for the group, and, later, at a time that is convenient and appropriate, the owner of the group can issue name certificates adding principals to the group. ACL administrators do not have to know all the members of a group when they are setting up ACLs. Note that the ACL administrator is free to add any principal's group to his ACL. He is not restricted to just adding his own groups (though he could just add his own groups if he so desired). For example, if the user controlling K_A maintains the ACL, he can add groups "K_F `friends`" and "K_G `friends`" on the ACL, say. An example of a SPKI/SDSI ACL is given in the appendix.

The ability to create groups makes security policies easier and more intuitive to define as they can be explicitly specified in terms of groups. As the names of the groups are at the discretion of the owners, groups can have meaningful, intuitive names. This makes the auditing of group definitions and ACLs, important facets of secure systems, simpler. One disadvantage of groups is that revoking the membership of a group member, who is later found to be untrustworthy, is non-trivial. If an explicit list of principals is maintained on the ACLs, the untrustworthy member's privileges can be easily revoked by removing his key from the ACLs. Revocation in SPKI/SDSI is discussed in detail later.

SPKI/SDSI has a clean support for "role-playing". A person may assume different roles, depending on the different jobs he may have. For example, in different capacities, Bob may be acting as an MIT student, or as an Intel summer intern. He may even have a separate key that he uses when travelling. Bob can create different key-pairs for each role he plays. One can then distinguish, from the key Bob uses to sign a particular message, the capacity in which Bob was acting when he signed the message. ACLs and groups can also be defined to include the appropriate key or the appropriate name for the key, if name certificates have been issued defining different names for the different keys. For example, the key Bob uses when he is travelling could have more restrictive access privileges than his regular keys.

There is a second way in which SPKI/SDSI supports roles. A group could be created for each role. The owner of the group is the principal who decides who should occupy the role by issuing a name certificate to the appropriate person(s). For example, the student online administrator for the Boxing Club, a student organization, can implement the role of club President by creating a group with the name "`Boxing-Club-President`". Its (sole) member would be defined to be the key/name of the organization's President. The two methods of creating roles differ in who controls the principal acting in the role.

Note that the second way in which SPKI/SDSI supports roles is also a useful way of defining a group of principals that possess a particular binary attribute. For example, the state of California might define the group "`state-employee`" by issuing certificates to residents with that attribute.

Thus, instead of using a global name space architecture, SPKI/SDSI uses local name spaces, with the ability to link them. A public key followed by zero or more identifiers forms a global label for a set of public keys.

Authorization

An authorization certificate grants a specific authorization from the certificate's issuer to the certificate's subject. A SPKI/SDSI authorization certificate consists of five fields: the issuer's key, the certificate's subject, a delegation bit, a tag, and a validity specification. Following are descriptions of these fields:

issuer The key that signs the certificate. This issuer is the principal granting the specific authorization.

subject The key or group that is receiving the grant of authorization. A subject can be a public key or a name consisting of a single public key followed by one or more identifiers. The public key in the subject does not have to be the issuer's key. (If the subject does not explicitly begin with a key, it is implicit that it is a name in the issuer's name space, and thus begins with the issuer's key.)

tag The tag specifies the specific authorization or authorizations being granted from the issuer to the subject. For example, it may specify the right to access a particular web site, or read and write to a particular set of files, or login to a particular account. An example of a tag will be presented shortly, and more examples are presented in the appendix.

delegation bit If this bit is true, the subject of this certificate is able to grant to other principals any subset of the authorization that it is receiving from the issuer of this certificate. If this bit is false, the issuer is not delegating to the subject the authority it is granting to it.

validity specification This is the same as that for a name cert.

SPKI/SDSI ACLs have a similar syntax to SPKI/SDSI authorization certificates. A SPKI/SDSI ACL consists of a list of entries. The *required* fields for each entry are a subject, a tag, and delegation bit. These fields are the same as that of an authorization certificate. In fact, each entry of an ACL can be considered to be an authorization certificate with the issuer being the owner of the ACL, and the subject, tag and delegation bit being as specified in the entry. The validity specification is as specified in the entry if it is specified; if the validity specification is not specified in the entry, the entry, and thus, the corresponding 'authorization certificate', is assumed to be valid for the time period $-\infty$ to $+\infty$. If ACLs are always going to be stored and processed in secure areas, they do not need issuer fields or signatures. If the owner of the ACL removes entries when they are no longer valid, then the validity specification is also optional. Examples of SPKI/SDSI ACLs and authorization certificates are given in the appendix and in the SPKI/SDSI IETF drafts and RFCs[11, 12].

An authorization certificate in which the issuer, "K", grants the authorization specified in the tag, "T", to the subject, "S", with the delegation bit, "p", and a validity specification, "V", is represented by the 5-tuple, "(K, S, T, p, V)". The value of "p" is either true or false. The validity specification will, generally, not be crucial to discussions in this thesis, as any certificate which fails its validity specification at the time it is being used should be ignored. It is included in this 5-

tuple representation to be consistent with other SPKI/SDSI publications. ACLs are assigned the special issuer "`SELF`", representing the owner of the ACL, and thus, each entry on an ACL has a 5-tuple representation (`SELF`, `S`, `T`, p, `V`). As an example, if K_A is Alice's public key, and K_B is Bob's public key, Alice can issue an authorization certificate, (K_A, K_B, T_1, true, V) granting Bob the authorization specified in T_1 with the permission to delegate this authorization. As another example, (`SELF`, K_B `sister friends`, T_2, false, V) represents an ACL entry with the group "K_B `sister friends`" on it; the members of this group are allowed to perform the operations specified in T_2, but are not allowed to grant this authority to anyone else.

SPKI/SDSI provides a very simple, flexible, authorization model. Authorizations can be specified as precisely or as generally as desired using flexible, user-defined tags. *Intuitively, a tag represents a set of requests.* The exact syntax of tags is clearly defined in the SPKI/SDSI IETF Drafts[11]. An example of a tag that would typically be in an authorization certificate authorizing an HTTP client to access a particular directory is given in Figure 3-3. The tag states explicitly the set of URLs the client is allowed to access, and the HTTP protocol and method that it is allowed to use when accessing the URLs. In this case, the URLs that the client is allowed to access are those that begin with "http://rooster.lcs.mit.edu:8081/demo/ABC/financial/", and it is allowed to perform either HTTP GET or POST requests on those files. More examples of SPKI/SDSI tags are given in the appendix and in the SPKI/SDSI IETF drafts and RFCs[11, 12].

```
(tag
  (http
   (* set GET POST)
   (*
    prefix
    http://rooster.lcs.mit.edu:8081/demo/ABC/financial/)))
```

Figure 3-3: An example of a tag

The ability to specify authorizations in tags in certificates is a powerful notion. Conventional certificates principally bind names to keys. However, a user's name is only one attribute of the user, and is rarely of security interest. A guardian really needs to know whether that user has been granted specific authorization to access the protected resource, and, if so, who granted him that authorization.

Besides groups, another fundamental notion of SPKI/SDSI is the ability to delegate authorization. One way of thinking of an authorization certificate is that it *transfers* or *propagates* a specific authorization from the issuer to the subject. If the delegation bit is set in the certificate, the subject is allowed to continue propagating this authorization, or some subset of it, to other principals, by issuing authorization certificates to them. If the subject, in turn, sets the delegation bit on its certificates to true, the principals to whom it is propagating the authorization will also be able to issue authorization certificates granting new principals the authority, and so on.

When security decisions are made, it is an individual's characteristics that are used to determine whether he should be given access to a protected resource. Sometimes the entity responsible for protecting the resource would prefer to delegate the responsibility for determining if a particular user should be issued access credentials for the resource. For example, in a company, it may be the system administrator (sys-admin) who is responsible for setting up and maintaining ACLs on the company's internal documents. Instead of having every new employee come to him for access credentials, he may want to delegate this responsibility to the employees in the Human Resources (HR) department. In other words, the sys-admin may want to trust employees in the HR department to correctly identify new employees and issue them certificates that will allow them access to the company's internal documents. With SPKI/SDSI, the sys-admin can delegate the authority to determine who is allowed to access the internal documents to the HR department. Figure 3-4 on page 67 gives an example. In the example, the sysadmin ($K_{SYS-ADMIN}$) adds the group "$K_{SYS-ADMIN}$ `Internal`" to ACLs and sets the delegation bits to true. "$T_{INTERNAL\_DOCUMENTS}$" represents the authority to access internal documents. The ACL entry added by the sys-admin is specified in "certificate" 3.9. He then issues a name certificate adding the HR manager's key, $K_{HR\_MANAGER}$, to the group "$K_{SYS-ADMIN}$ `Internal`" (certificate 3.10). The HR manager has, thus, been delegated the authority to decide who should access the internal company documents, as the delegation bit in the ACL entry is true (of course, the sys-admin can still authorize people to view the internal documents as well, but he may decide to leave this job exclusively to HR). The HR manager issues authorization certificates to employees in his department. He also sets the delegation bits in their certificates to true, so that they can also authorize people to view the documents. The HR employees, in turn, authorize new employees after authenticating and validating them. When they issue certificates to these employees, if they are going to be in HR, they will set the delegation bits to true, and if they are not going to be in HR, they set the delegation bits to false. This keeps the power to authorize people to view internal company documents within the Human Resources department (and the sys-admin because he is maintaining the ACLs). Thus, suppose a particular HR employee's key is $K_{HR\_EMPLOYEE}$, and a particular non-HR (financial) employee's key is $K_{FINANCIAL\_EMPLOYEE}$. The HR manager would authorize the HR employee by issuing him certificate 3.11. The HR employee would authorize the financial employee by issuing him certificate 3.12.

(3.9) $(\texttt{SELF},\ K_{SYS-ADMIN}\ \texttt{Internal},\ T_{INTERNAL\_DOCUMENTS},\ true,\ V)$

(3.10) $K_{SYS-ADMIN}\ \texttt{Internal}\ \longrightarrow\ K_{HR\_MANAGER}$

(3.11) $(K_{HR\_MANAGER},\ K_{HR\_EMPLOYEE},\ T_{INTERNAL\_DOCUMENTS},\ true,\ V)$

(3.12) $(K_{HR\_EMPLOYEE},\ K_{FINANCIAL\_EMPLOYEE},\ T_{INTERNAL\_DOCUMENTS},\ false,\ V)$

Figure 3-4: An example of delegation using SPKI/SDSI certificates

Delegation can also be accomplished using just name certificates alone. In the previous example, the sys-admin could have added the group "$K_{SYS-ADMIN}$ `Internal`" onto the ACL, then issued the HR manager the name certificate adding the group "$K_{HR\_MANAGER}$ `HR`" to the group "$K_{SYS-ADMIN}$ `Internal`" ($K_{SYS-ADMIN}\ \texttt{Internal} \longrightarrow K_{HR\_MANAGER}\ \texttt{HR}$). The HR manager can, firstly, now authorize himself to view the internal documents by issuing himself the name certificate $K_{HR\_MANAGER}$ $\texttt{HR} \longrightarrow K_{HR\_MANAGER}$, adding himself to his own HR group. He can give the power of deciding who can access the internal documents to the HR employee by issuing him the name certificate $K_{HR\_MANAGER}$ $\texttt{HR} \longrightarrow K_{HR\_EMPLOYEE}$ $\texttt{HR}$. (Similarly, the HR employee can now give himself the authority to view the internal pages.) The HR employee would authorize the financial employee by issuing him the name certificate $K_{HR\_EMPLOYEE}$ $\texttt{HR} \longrightarrow K_{FINANCIAL\_EMPLOYEE}$. The financial employee will not be able to propagate this authority himself, and thus, the power to authorize people to view internal documents remains in HR. An important distinction between delegation that is achieved using an authorization certificate and delegation achieved using a name certificate is that, in the latter case, the subject is granted, and allowed to transfer, all of the authority that the issuing group name has. That is, the authority delegated from the issuing group name to the subject via a name certificate is all-or-nothing. With an authorization certificate, the issuer may grant, and delegate to the subject, any subset of the authority that the issuer has. This subset is precisely specified in the tag of the certificate. The correct way to think about name certificates, however, is not that they can be used to propagate authorizations, but, instead, that they are used to define names within the issuer's local name space. The fact they can be used to propagate authorizations is a by-product of the properties of the SPKI/SDSI name space architecture.

Of course, if the sys-admin would like to directly authorize every employee himself, he can set up the ACL such that every entry is either a public key, or a group that the sys-admin defines i.e. a group whose name consists of $K_{SYS-ADMIN}$ followed by a single identifier. In this case, each new employee will have to go to the sys-admin to have his key placed on the ACL, or to be issued with his group membership certificate.

Because the term 'trust' is vague, and imprecisely defined, SPKI/SDSI uses the transitive term, 'delegate', instead. This makes the transitivity relations in certificate chains easier to see. If Alice issues an authorization certificate to Bob, granting him some authority, and she delegates that authority to Bob by setting the delegation bit in the certificate to true, and Bob grants Carol some subset of the authority, Alice will have, effectively, granted (that subset of the) authority to Carol. If Alice is presented with a chain of valid certificates, in which she has delegated some authority to Bob, Bob has delegated this authority to Carol, Carol has delegated this authority to Derek, and Derek has granted (delegation bit set to true or false) this authority to Edward, Alice will trust that Edward has been granted the authority from her. If, at any link in the chain, the authorization certificate has its delegation bit set to false, Alice will not trust authority granted beyond that certificate. Alice can control the propagation of authority by setting the delegation bit in her certificate to false, in which case it would not be possible for Bob to grant Carol the authority he is receiving from Alice.

Authorization Flow The SPKI/SDSI Infrastructure is primarily concerned with authorizing principals to perform particular operations on protected resources. To provide access control[18] on a resource, the guardian sets up an access control list (ACL) to protect it. Alice, a typical user, requests to perform a particular operation on the resource. Examples of requests are a request to read a particular file, a request to read a file in a particular directory, a request to login to a particular account, and a request to turn on a particular electrical appliance; in these examples, the 'protected resources' are the file, directory, account and appliance respectively. For the guardian to honor Alice's request, her request must be accompanied with a "*proof of authenticity*", that authenticates the request, and a "*proof of authorization*" that shows that she is authorized to perform the request. The "proof of authenticity" is typically a signed tag, and the "proof of authorization" is typically a sequence/chain of certificates. The principal that signed the tag must be the same principal that the chain of certificates authorizes.

An example scenario follows:

1. Alice requests access to the protected resource. This initial request is not accompanied with a proof of authentication or a proof of authorization.

2. The guardian denies this initial request, because it is not authenticated nor

[18] described in Section 2.5

authorized. It issues a *challenge* to Alice saying ‘you are trying to access a protected file. Using this ACL and tag, prove to me that you have the credentials to perform the action specified in the tag on the file protected by the ACL.’ The ACL returned will be the ACL protecting the object, and the tag will be formed from Alice’s request.

3. Using the SPKI/SDSI *certificate chain discovery algorithm*[3], Alice generates a chain of certificates. The certificate chain discovery algorithm takes as input an ACL, a tag, a public key, a set of signed certificates, and a timestamp. If it exists, the algorithm returns a certificate chain, consisting of signed certificates, which provides proof that the public key (principal) is authorized to perform the operation(s) specified in the tag on the object protected by the ACL, at the time specified in the timestamp. An efficient algorithm for discovering certificate chains in SPKI/SDSI is described in detail in the paper “Certificate Chain Discovery in SPKI/SDSI”[3]. Chapter 6 provides an overview of this algorithm.

4. The principal signs the tag from the guardian. It then sends the certificate chain and signed tag in a second request to the guardian. This second request is the principal’s *response* to the guardian’s challenge. The signed tag provides proof of authenticity, and the certificate chain provides proof of authorization.

5. The guardian verifies the principal’s second request, and if it verifies, allows the principal to access the object. The guardian first verifies the signature on the request, to verify the authenticity of the request, then determines if the certificate chain provides a *chain of authorization* from the guardian to the public key that was used to verify the request’s signature.

An *authorization chain/flow*[13] consists of a chain of valid certificates that authorizes Alice to perform the specific operation she is requesting to perform in the tag she signed. The proof of authorization, i.e. certificate chain, in Alice’s second request provides an authorization chain from the ACL’s issuer to her public key, the key that signed the request. Recall that the ACL's issuer is referred to as "`SELF`", which represents the particular guardian of the resource. When the guardian verifies the certificate chain, it is verifying that there is a chain of authorization originating from it, through the ACL, through zero or more certificates, to Alice's key. If the guardian successfully verifies this authorization chain, Alice is authorized to perform the requested operation on the protected resource. Note that SPKI/SDSI certificate chains which demonstrate flows of authorization can consist of just authorization certificates or both name certificates and authorization certificates.

When Alice is being issued her certificates, she should be given all of the certificates necessary to establish the necessary chain of authorization. In the example described in Figure 3-4 on page 67, if Alice is the HR manager, she would just be given certificate 3.10; if she were the HR employee, she would be given certificates 3.10 and 3.11; if she were the financial employee, she would be given certificates 3.10, 3.11 and 3.12. In each case, she will need to use the certificates to establish the authorization chain from the "$K_{SYS-ADMIN}$ `Internal`" group to her key. As each SPKI/SDSI certificate has one signer, a chain of certificates typically represents one certification path from the guardian to the key in question.

Compare this model to the X.509 hierarchical trust model. In the X.509 model, the trust does not originate from the guardian of the resource. Instead, it originates from a third party over whom the guardian may have no control, but is supposed to trust. The initial decision of trust is removed from the guardian. If there is a *chain of authentication* from the 'trusted' third party to Alice's key, the guardian of the protected resource checks that Alice is on the ACL it is maintaining, and, if she is, authorizes Alice to perform the operation she is requesting if it is a subset of the set of authorizations that the guardian allows Alice to perform.

It is important to note that, in SPKI/SDSI, when a request is made, the user making the request is not authenticated: it is the user's request that is authenticated, using a digital signature (recall that, in SPKI/SDSI, the principals are public keys). The guardian does not necessarily know, or necessarily need to know, the identity of the user making the request: it just needs to know that the particular principal has been authorized to perform the action it is requesting to perform. A SPKI/SDSI certificate chain is relatively silent about the identity of the user using the key-pair, and it is up to the recipient of the certificates (the guardian, in this discussion,) to build up its own image of a principal based on interactions with that principal. For example, the guardian might determine that the signed request it just received was made by the same principal that made a request ten minutes ago, but, beyond knowing that that principal is part of a particular group, say, it may have no idea who the user controlling that principal is. X.509 certificates contain identifying (authentication) information, certified by the 'trusted' third party. The guardian makes access control decisions based on the user's authentication information found in the certificates.

Certificate Result Certificate In SPKI/SDSI, after a guardian verifies that Alice is authorized to perform a particular operation, or set of operations on a protected resource, it can issue an authorization certificate itself to Alice, which summarizes the set of authorizations it has just derived from the certificates Alice presented. This certificate, termed a *Certificate Result Certificate*[13], is a typical authorization certificate, with the following fields:

issuer the guardian's key

subject Alice's key

delegation bit This bit is set to true, if, as determined from the authorization chain used to authorize Alice, Alice is allowed to delegate the authority in this certificate to others.

tag The summary of the set of authorizations that the guardian has determined, from the authorization chain, that Alice is allowed to perform.

validity specification Again, as determined from the authorization chain, the validity specification on the certificate. If all of the specifications in the certificates in the authorization chain are in terms of validity time periods, this validity specification will also be a validity time period that will be no larger than that of any of the certificates in the chain.

This certificate would be signed with the guardian's key, and can be returned to Alice. In the future, when Alice is requesting to access the resource again, she can simply present the *Certificate Result Certificate*, instead of re-deriving and sending the full certificate chain again. The guardian verifies requests accompanied by a Certificate Result Certificate in exactly the same manner as it verifies requests accompanied by the original certificate chain. Certificate chains could, potentially, consist of several certificates: there is no upper bound on the number of certificates that can be in a certificate chain. If Certificate Result Certificates are used, it is easier for the server to verify future requests. Less network bandwidth is utilized as fewer certificates are transmitted. Servers can be also stateless: instead of designing servers to remember the authorizations for a particular client, the server can reply with a Certificate Result Certificate after validating the first request, and the client can send this certificate with future requests. Stateless servers are less vulnerable to denial-of-service attacks. Furthermore, if the server is stateless, it is easier for the server to completely mediate every access to the protected resource. The client can also use the Certificate Result Certificate that it received from one guardian in a certificate chain it presents to another guardian, if the certificate is useful in establishing an authorization chain from that second guardian. This saves both time and space for the second guardian, if it trusts the first one to grant the authority specified in the tag.

If the Certificate Result Certificate will not be used by any verifier other than the one that created it, it can even be MAC'd by a symmetric key private to the verifier, instead of signed with the verifier's private key. This can provide an added efficiency benefit as symmetric-key operations are much faster than public-key operations.

In the example in Figure 3-4 on page 67, the financial employee presents a certificate chain consisting of certificates 3.10, 3.11, and 3.12 to be allowed to view the internal documents. The resulting 5-tuple that the guardian forms after verifying that the certificate chain authorizes the employee is

(SELF, $K_{FINANCIAL\_EMPLOYEE}$, $T_{INTERNAL\_DOCUMENTS}$, false, V). The issuer "SELF" is the issuer of the ACL, and represents the guardian of the resource. However, authorization certificates require actual public keys as issuers. One approach to resolving this issue is to recognize that the ACL administrator is the user responsible for protecting the resource, and replace "SELF" with his public key when forming the Certificate Result Certificate. In this case, the ACL administrator is the sys-admin, and his key is $K_{SYS-ADMIN}$. Thus, the Certificate Result Certificate that would be formed is ($K_{SYS-ADMIN}$, $K_{FINANCIAL\_EMPLOYEE}$, $T_{INTERNAL\_DOCUMENTS}$, false, V). In order for the financial employee to use this certificate, a layer of indirection[19] must be placed on the ACL. The sys-admin can add an entry, (SELF, $K_{SYS-ADMIN}$, (*), true, V), onto the ACL in which he, essentially, grants himself all authority, with the ability to delegate any of the authority that he has. When the financial employee presents his Certificate Result Certificate with an authenticated request to access internal documents, his request will be honored.

Threshold Subjects SPKI/SDSI provides a fault tolerance mechanism by way of threshold subjects. Threshold subjects may only be used in authorization certificates and may not be used in name certificates. They can be used to specify a requirement that "k out of n" keys must sign a request before that request can be honored. For example, an authorization certificate could have a threshold subject that requires that two out of three keys specified must sign a request before a request using the certificate is honored. Threshold subjects can also be placed directly on ACLs (which are, essentially, a list of authorization certificates). Threshold subjects provide fault tolerance because, if one of the keys in the threshold subject is compromised, the adversary is still unable to gain access to restricted resources as he will not be able to convince any of the other principals to sign the request. For example, in a "2 out of 3" threshold, the adversary will need to compromise two keys before being able to convince the guard.

[19] "Any problem in Computer Science can be solved by adding another layer of indirection." - David Wheeler

In terms of flow of authorization, the way to think about threshold subjects is that, if a certificate specifies a threshold subject, at least k out of the n subjects listed must have agreed to sign the request before authorization can be propagated through the certificate. Certificates with threshold subjects may also have their delegation bit set to true, and, in this case, paths must be shown from (at least) k subjects to the signers of the request, before authorization can 'flow' through the certificate with the threshold subject. For threshold subjects specified on ACLs, the concept is the same: "the actual intent is to insure that there are k distinct paths passing permission between the verifier's ACL and the prover's request."[20]

If the delegation bit in a certificate with a threshold subject is set to true, fewer than k keys may be required to sign a valid request if the same principal has been granted some authority originating from multiple different keys in the threshold subject. For example, consider a 2-out-of-3 threshold consisting of the keys $K1$, $K2$, and $K3$, and the certificate with this threshold having the delegation bit set to true; suppose K_1 and K_2 grant Alice (K_A) the permission in the certificate; then Alice's signature alone on a request is good enough to fulfill the requirements of the threshold subject.

A threshold subject can consist of both keys and names/groups. If groups are used in the threshold subject, again, fewer than k keys may be required to sign a valid request if the same principal belongs to different groups. For example, if Alice is on the Tennis team, and the 2-out-of-3 threshold subject consists of the group "Tennis_Team", the group "Basketball_Team", and Alice's public key, then Alice's signature alone on a request is good enough to fulfill the requirements of the threshold subject. Furthermore, if the delegation bit in the certificate is set to true, then the signature of any single principal to whom Alice grants the authorization specified in the certificate is enough to fulfill the requirements of the threshold subject.

If one wants to be certain that k different users must independently agree to sign

[20] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *RFC 2693: SPKI Certificate Theory.* The Internet Society. September 1999. See `ftp://ftp.isi.edu/in-notes/rfc2693.txt`. Last visited 08/07/2001.

a request before it can be honored, then a threshold subject with just keys in them, with the delegation bit set to false, should be specified. One should also take steps to check that the different keys are controlled by different users.

To validate certificate chains, the verifier could make n copies of the certificate with the threshold subject, use a separate copy to handle each different subject that is specified in the threshold, and recursively call its verification procedure to check if there is an authorization chain from each copy to a signer's key. If there are chains from k copies, authorization is allowed to 'flow' through the certificate with the threshold subject.

Threshold subjects add a degree of complexity to the SPKI/SDSI infrastructure. However, they do provide fault tolerance and are very useful. There are many situations where at least two signatures, say, should be required before a request is honored.

Certificate Guarantee

If Certificate Revocation Lists (CRLs) are used for invalidating/revoking certificates, the guarantee the certificate provides is: "This certificate is good until the expiration date. Unless, of course, you hear that is has been revoked."[21] To a guardian protecting a resource, this means that if the certificate has not expired, and is not on the latest CRL that the guardian is using, then it is *probably* valid. In principal, the guardian is always required to check against a CRL to see if the certificate has been revoked. Because CRLs are issued by CAs, there could be significant, variable delay between a CA being notified that a certificate should be revoked, and the guardian learning this knowledge.

The SPKI/SDSI certificate guarantee is at the other end of the spectrum. It is: "This certificate is good until the expiration date. Period."[22] To a guardian

[21] Ronald L. Rivest. Can We Eliminate Certificate Revocation Lists? Proceedings of *Financial Cryptography '98*; Springer Lecture Notes in Computer Science No. 1464 (Rafael Hirschfeld, ed.), February 1998. See `http://theory.lcs.mit.edu/~rivest/Rivest-CanWeEliminateCertificateRevocationLists.ps`. Last visited 08/07/2001.

[22] Ibid.

protecting a resource, this means that if the certificate has not expired, then it is *guaranteed* to be valid. The guardian *never* has to check whether a certificate has been revoked. A certificate from a public-key infrastructure (PKI) which uses CRLs goes through two phases: (1) probably valid (2) expired. A SPKI/SDSI certificate also goes through two phases: (1) definitely valid (2) expired. This is a definite improvement over certificates from PKIs which use CRLs, as they have no "definitely valid" phase.

How does the SPKI/SDSI model work? First, recall the four reasons a certificate could be invalid: (1) the certificate has expired, (2) the requestor's private key has been compromised, (3) the issuer's private key has been compromised (4) the issuer incorrectly issued the certificate. SPKI/SDSI advocates using reasonably short validity periods inside certificates. As with any PKI, issuers must take care when issuing certificates. If a certificate has been incorrectly issued, there would only be a reasonably short period of time in which it could potentially be used. SPKI/SDSI also advocates using "*Certificates of Health*" to deal with the specific issue of key compromise of the requestor's key.

Short validity periods are facilitated in SPKI/SDSI since, in any particular certificate chain authorizing a principal to access a particular resource, the function of each certificate can be easily partitioned among the certificates. Suppose, for example, Alice requires a name certificate binding her name in Bob's local name space to her key (K_B `Alice` $\longrightarrow K_A$), and a name certificate adding "K_B `Alice`" to Charlie's team (K_C `Team` $\longrightarrow K_B$ `Alice`); suppose the group, "K_C `Team`", is on an ACL protecting access to attend Charlie's team meetings; the second certificate can specify a relatively short validity period giving Alice the ability to attend a particular meeting.

One of the reasons for standard PKIs to revoke certificates is because the requestor's key has been compromised. SPKI/SDSI treats key compromise as a different, separate, issue from certificate revocation. It argues "that certificates should not be revoked merely because the key is compromised. Rather, the signer should present separate evidence to the acceptor that the key has *not* been compromised. Since, in this framework, the no-compromise evidence is separate, the ordinary certificates can continue to be 'valid' even though the key has been compromised."[23] SPKI/SDSI suggests using a new kind of agent, called a "key compromise agent" (KCA), or a "suicide bureau" (SB). Multiple SBs could cooperate to serve SPKI/SDSI communities. When Alice creates her key pair, she also signs a personal "suicide note" which she protects in a private place, and also registers her public key with a SB. In the, hopefully unlikely event that her key is compromised or her key is lost, she sends her suicide note to the SB. The SB broadcasts this note on the SB network so that other SBs are aware of the compromised key.

If Alice's key has not been compromised or lost, she can ask an SB for a "Certificate of Health", certifying that she believes that she is the only entity controlling her private key. Now, instead of a guardian needing to check if a requestor's key has been compromised, it can require that the requestor present a "Certificate of Health" along with its request. This certificate will have the time and date that it was issued within the certificate, and the guardian can demand a more recent health certificate before honoring the request. The task of demonstrating that a key has not been compromised is, thus, the responsibility of the user using that key, instead of the responsibility of the guardian. If CRLs are used, the situation is reversed.

Note that, in comparing "Certificates of Health" with CRLs, a "Certificates of Health" is essentially a 'positive statement', whereas a CRL is a negative statement. A "Certificate of Health" states that this key has not be compromised; a CRL states that all keys (for which the issuer has issued certificates), except the ones on this list, have not been compromised. Negative statements are much harder to ascertain and prove as being correct than are positive statements.

In some security systems, using short validity periods may not, by itself, be sufficient. Thus, in the case of high security systems, in which guardians would like to be alerted as soon as possible before they use a certificate that should not have been

[23] Ronald L. Rivest. Can We Eliminate Certificate Revocation Lists? Proceedings of *Financial Cryptography '98*; Springer Lecture Notes in Computer Science No. 1464 (Rafael Hirschfeld, ed.), February 1998. See `http://theory.lcs.mit.edu/~rivest/Rivest-CanWeEliminateCertificateRevocationLists.ps`. Last visited 08/07/2001.

issued, certificates can have online-checks in their validity specification. An online-check is basically a pointer to a server that the guardian must query to see if the certificate has been revoked or to ask for more evidence that it is still valid. If present in a certificate, the guardian must query the server each time before accepting the certificate. Online checks require a network that is reliable and available. SPKI/SDSI also provides specifications for CRLs (a CRL is a blacklist: any certificate on a CRL is invalid) and Revalidation Lists (a Revalidation List is a whitelist: only certificates on a revalidation list are valid).

Sexps

SPKI/SDSI is designed to be simple to understand, adopt and use. To facilitate these goals, the infrastructure's data structures, referred to as "S-expressions" (Sexps)[46], have human-readable ASCII representations, with simple formats. Having data structures which are easy to read is a useful security feature: humans can easily examine a certificate themselves before deciding whether to trust it. Examples of SPKI/SDSI objects are given in the appendix and SPKI/SDSI IETF drafts and RFCs[11, 12].

The easy-to-read ASCII representation of an Sexp is referred to as the Sexp's 'advanced form'. Besides the advanced form, the other standard representations are the canonical (or packed) form and the transport form. The canonical form was designed to be simple to parse by an application, and the transport form was optimized for transmitting Sexps over networks. Programs exist for easily converting from one format to another[45]. There have been also efforts to represent SPKI/SDSI's Sexps using the Extensible Markup Language (XML)[37].

SPKI/SDSI Summary

SPKI/SDSI achieves scalability without compromising security as it is uses public keys and a local name space architecture. The infrastructure provides a clean model for delegating authority, because authorizations can be specified in authorization certificates. It also provides a clean, scalable model for defining groups, because of its local name space architecture, and the property that each name can be bound to zero, one, or more keys. In SPKI/SDSI, the principals are the public keys. A principal's request is honored if the request authenticates, and there is a chain of authorization from the guardian, via the ACL and zero or more certificates, to the principal.

3.3.4 Comparison of X.509, PGP, SPKI/SDSI

Table 3.1 on page 81 compares and summarizes the similarities and differences between X.509, PGP, and SPKI/SDSI.

X.509	**Name Space:**	Global
	Types of Certificates:	Name Certificates
	Name-to-Key binding:	Single-valued function: each global name is bound to exactly one key (assuming each user has a single public-private key pair).
	CA Characteristics:	Global Hierarchy. There are commercial X.509 CAs. X.509 communities are built from the top-down.
	Trust Model:	Hierarchical Trust Model. Trust originates from a 'trusted' CA, over which the guardian may or may not have control. A requestor provides a *chain of authentication* from the 'trusted' CA to the requestor's key.
	Signatures:	Each certificate has one signature, belonging to the issuer of the certificate.
	Certificate Revocation:	Uses CRLs
PGP	**Name Space:**	Global
	Types of Certificates:	Name Certificates
	Name-to-Key binding:	Single-valued function: each global name is bound to exactly one key (assuming each user has a single public-private key pair).
	CA Characteristics:	Egalitarian design. Each key can issue certificates. PGP communities are built from the bottom-up in a distributed manner.
	Trust Model:	*Web of Trust*
	Signatures:	Each certificate can have multiple signatures; the first signature belongs to the issuer of the certificate.
	Certificate Revocation:	A suicide note is posted on PGP certificate servers, and widely distributed to people who have the compromised key on their public keyrings.
SPKI/SDSI	**Name Space:**	Local
	Types of Certificates:	Name Certificates, Authorization Certificates
	Name-to-Key binding:	Multi-valued function: each local name is bound to zero, one or more keys (assuming each user has a single public-private key pair).
	CA Characteristics:	Egalitarian design. The principals are the public keys. Each key can issue certificates. SPKI/SDSI communities are built from the bottom-up in a distributed manner.
	Trust Model:	Trust originates from the guardian. A requestor provides a *chain of authorization* from the guardian to the requestor's key. The infrastructure has a clean, scalable model for defining groups and delegating authority.
	Signatures:	Each certificate has one signature, belonging to the issuer of the certificate.
	Certificate Revocation:	Advocates using short validity periods and *Certificates of Health*.

Table 3.1: Comparison of X.509, PGP, and SPKI/SDSI

Chapter 4

SPKI/SDSI Access Control Protocol

4.1 Protocol

Project Geronimo, the project described in this thesis, explores the viability of SPKI/SDSI by using it to provide access control over the Web. SPKI/SDSI was integrated into the Netscape web client and Apache web server. This section describes the protocol that the client and server use to communicate. The goal of this protocol is to allow the server to make an access control decision when a client requests access to a protected resource. The protocol facilitates complete mediation of client requests.

The protocol implemented by the client and server consists of four messages. This protocol is outlined in Figure 4-1 on page 88, and following is its description:

1. The client sends a standard HTTP request, unauthenticated and unauthorized, to the server.

2. The server has directories which are public (not protected), and directories which are protected using SPKI/SDSI access control lists (ACLs). If the client's request is for a file in a protected directory, the server sends a response to the client containing the ACL protecting the directory and the SPKI/SDSI

tag formed from the client's HTTP request[1]. The response's content-type is "application/x-spki-sdsi". The server returns this response whenever the client sends an unauthenticated request for a file in a directory that is protected by a SPKI/SDSI ACL. If the requested file is in a public directory, it is returned in a standard HTTP reply with HTTP status code "200 OK"[21].

3. (a) If the plugin has not already been started, the content-type "application/x-spki-sdsi" will trigger the SPKI/SDSI Netscape plugin. During initialization, the plugin prompts the user for his password to unlock his private key using a small Java-based pop-up password box. The password is used by the client alone and is never transmitted across the network. If the private key is successfully unlocked, a small session window appears next to the Netscape browser. This session window maintains state between client requests during the same Netscape session. This state is created and maintained by the client only, and is never transmitted across the network. It prevents the user from having to re-enter his password every time he accesses a SPKI/SDSI protected document within the same Netscape session.

   (b) Using the ACL and tag from the server, and the user's public key and certificate cache (which stores all of the user's certificates), the plugin generates a sequence of certificates using the SPKI/SDSI *certificate chain discovery algorithm*[3]. This certificate sequence provides a *chain of authorization* from the ACL 'issuer' to the user's key and provides proof that the client is authorized to perform the operation specified in the tag. If the algorithm is unable to generate a sequence, because the user does not have the necessary certificates, or if the algorithm does not need to generate a sequence, because the user's key is directly on the ACL, the algorithm

[1] Theoretically, the ACL itself, could be a protected resource, protected by another ACL. In this case, the server would return the ACL protecting the ACL; the client will need to demonstrate that the user's key is on this ACL, either directly or via certificates, before gaining access to the ACL protecting the object to which access was originally requested.

returns an empty sequence. (The plugin could return an error to the user if it is not able to generate a certificate sequence and the user's key is not directly on the ACL. In our design and implementation, we chose not to return an error at this point, but to let the plugin send an empty certificate sequence to the server. This way, when the request is not accepted, the plugin can be sent the server-side customizable error page[2], which lets the user know where he should go to get valid certificates.)

(c) The plugin generates a timestamp using the client's local clock. It creates a SPKI/SDSI sequence consisting of the tag from the server, and the timestamp it generated. It signs this sequence with the user's private key. A copy of the user's public key is included in the signature. The plugin then sends the tag-timestamp sequence, its signature, and certificate sequence generated in Step 3b to the server.

4. (a) The server verifies the request by first checking the timestamp in the tag-timestamp sequence against the time in the server's local clock to ensure that the request was made recently[3]; recreating the tag from the client's HTTP request and checking that it is the same as the tag in the tag-timestamp sequence; extracting the public key from the signature; verifying the signature on the tag-timestamp sequence using this key; validating the certificates in the certificate sequence; and verifying that there is a chain of authorization from an entry on the ACL to the key from the signature via the certificate sequence presented. The authorization chain must authorize the client to perform the requested operation, as specified by the tag formed from the client's HTTP request. If the request successfully verifies, the server returns the requested object to the client with HTTP status code "200 OK". If the verification fails, a server-side customizable error page

[2] described in step 4a and section 5.1

[3] In our prototype implementation, the server checks that the timestamp in the client's tag-timestamp sequence is within five minutes of the server's local time i.e.
server's local time $-$ timestamp $<$ 5 minutes.

is returned to the client with the HTTP error code “403 Forbidden”[21]. Error code “403 Forbidden” and the error page are returned whenever the client presents an authenticated request that is denied. The server’s admin sets up the error page, and can use it to provide feedback to the user about why his credentials failed, and whom the user should contact to get the correct credentials.

(b) The client displays either the requested document, or the customizable error page, that is returned by the server.

The meaning of each client request depends only on the content of the request itself, and is not dependent on the content of previous requests. The SPKI/SDSI web server implementing this protocol is *stateless*, making it easier to implement, and more resistant to denial-of-service attacks. Because the server is stateless, it is easier for it to completely mediate every client request, and determine if it should be honored. The server evaluates each request based on its own merit, returns a response that depends only on that request, then forgets about the request altogether. The response the server returns is one of the following:

- If the request is for a file in a public directory, the file is returned. This reply is the standard HTTP reply, with HTTP status code “200 OK”.

- If the request is for a file in a protected directory, and the request is unauthenticated, the server responds with the ACL protecting the directory and the SPKI/SDSI tag formed from the client’s HTTP request. The response’s content-type is “application/x-spki-sdsi”.

- If the request is for a file in a protected directory, and the request is authenticated, the server verifies the request using the request’s signature and certificate sequence provided with the request. If no certificate sequence is provided, the server uses an empty certificate sequence. If the request’s verification fails, the server returns a server-side customizable error page with the HTTP error code “403 Forbidden”.

- If the request is for a file in a protected directory, and the request is authenticated, the server verifies the request using the request's signature, and certificate sequence provided with the request. If no certificate sequence is provided, the server uses an empty certificate sequence. If the request successfully verifies, the server returns the requested file with HTTP status code "200 OK".

The protocol can be viewed as a typical challenge-response protocol. The server reply in step 2 of the protocol is a challenge the server issues to the client, saying 'you are trying to access a protected file. Using this ACL and tag, prove to me that you have the credentials to perform the action specified in the tag on the file protected by the ACL.' The client uses the ACL and tag to help it produce a certificate sequence, using the SPKI/SDSI chain discovery algorithm. It then sends the certificate sequence and signed tag-timestamp sequence in a second request to the guardian. The digital signature provides proof of authenticity, and the certificate sequence provides proof of authorization. The server verifies the second request, and, if it verifies, returns the requested file.

The timestamp in the tag-timestamp sequence helps to protect against replay attacks. As an example, suppose the server logs requests. Suppose this log is not disposed of properly. The timestamp prevents an adversary from replaying requests found in the log and gaining access to protected resources[4].

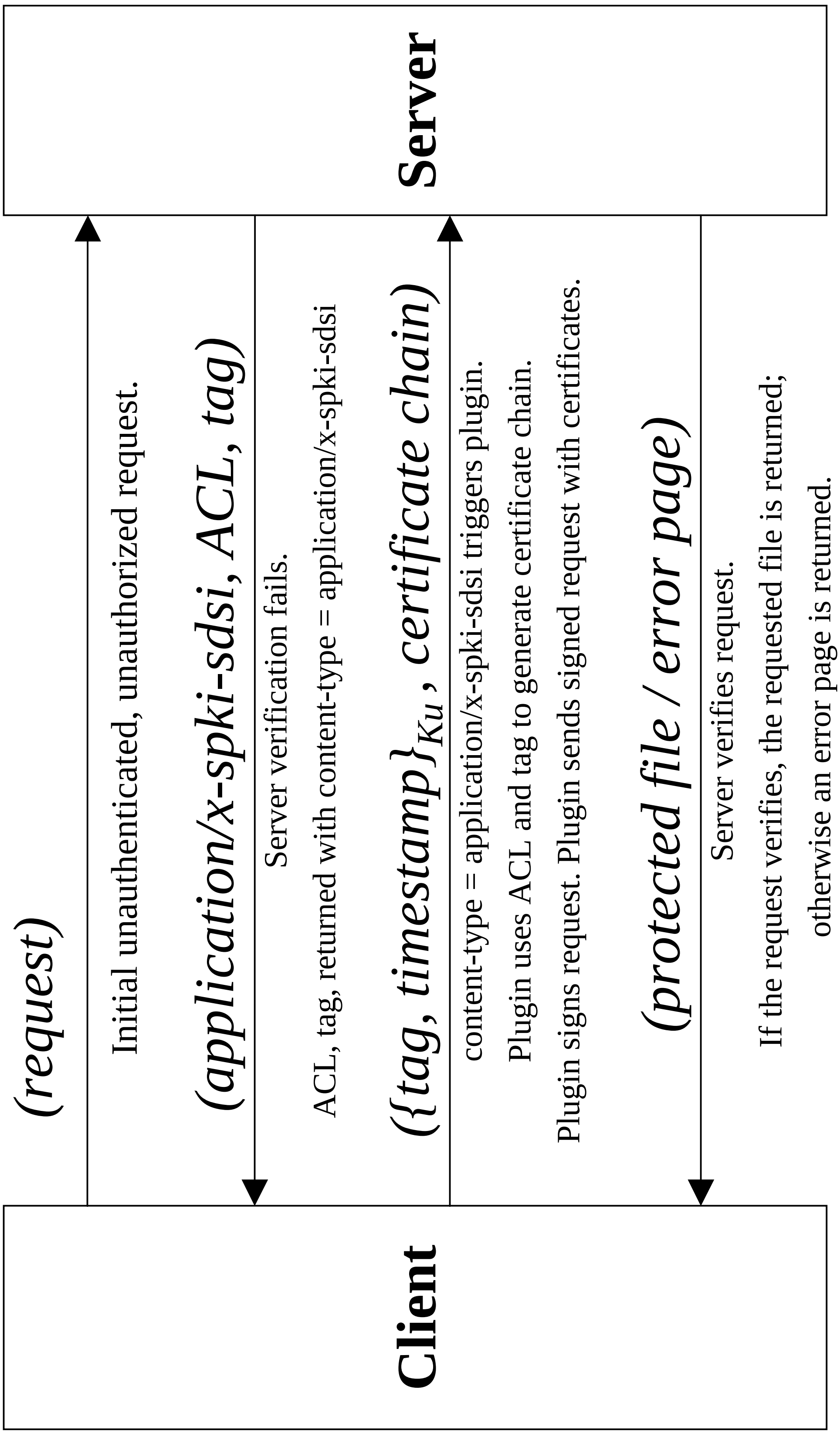
4.1.1 Protocol Variations

Following are three variations on the basic SPKI/SDSI protocol:

First Variation

As the meaning of each request depends only on the request itself, the protocol can be easily adapted to consist of only two messages:

[4] In order to use timestamps, the client's clock and server's clock need to be fairly synchronized; SPKI/SDSI already makes an assumption about fairly synchronized clocks when validity time periods are specified in certificates. An alternative design would be for the server to send a nonce in the protocol's second message, and for the client to replace the timestamp with this nonce in the protocol's third message; of course, in this design, the server is not stateless.


Client
Server
(request)
(application/x-spki-sdsi, ACL, tag)
($\{tag, timestamp\}_{Ku}$, certificate chain)
(protected file / error page)

Initial unauthenticated, unauthorized request.

Server verification fails.

ACL, tag, returned with content-type = application/x-spki-sdsi

content-type = application/x-spki-sdsi triggers plugin.

Plugin uses ACL and tag to generate certificate chain.

Plugin signs request. Plugin sends signed request with certificates.

Server verifies request.

If the request verifies, the requested file is returned;

otherwise an error page is returned.

Figure 4-1: SPKI/SDSI Access Control Protocol

1. The client creates the tag formed from its HTTP request. It sends the tag-timestamp sequence and its signature to the server, accompanied with all of the certificates in the user's cache, in its initial HTTP request.

2. The server performs the certificate chain discovery, and verifies the credentials in the same manner as in the original protocol. The protected object is returned with HTTP status code "200 OK" if the verification is successful. The error page, with HTTP error code "403 Forbidden", is returned if the verification is not successful.

One can imagine favoring this two-stage protocol in a system in which the expected number of certificates in a typical user's cache is small. The server has slightly more work, but, if the number of certificates is small, the extra work will be small. There is a privacy problem in that the server will be receiving all of the certificates the user has, instead of just the ones the user needs to send.

Second Variation

If ACLs typically contain a number of entries of public keys, the client's initial request could always be signed. Thus, if the user's key is directly on the ACL, and the tag in the relevant ACL entry permits the client to perform the operation it is requesting to perform, the server can return the object in its first reply.

The protocol would be:

1. The client creates the tag formed from its HTTP request. It sends the tag-timestamp sequence and its signature to the server in its initial HTTP request.

2. If the request is for a file in a public directory, the file is returned with HTTP status code "200 OK". If the client's request is for a file in a protected directory, the server verifies the client's request. The request is verified in a manner similar to the original protocol except that instead of verifying certificates and verifying certificate chains, the server checks that the key from the signature is directly on the ACL and that the tag in the relevant ACL entry permits the client

to perform the operation it is requesting to perform. If the client's request successfully verifies, the requested object is returned in a standard HTTP reply with HTTP status code "200 OK". Otherwise, the server sends a response to the client containing the ACL and the SPKI/SDSI tag formed from the client's HTTP request. This response's content-type is "application/x-spki-sdsi".

3. The subsequent steps are the same as those in the original protocol.

Third Variation

Certificate sequences could, potentially, consist of several certificates: there is no upper bound on the number of certificates that can be in a certificate sequence. If clients had to transmit these sequences with every request, there would be a significant performance loss. After the server verifies that the client is authorized to perform a particular operation, or set of operations, it can issue a *Certificate Result Certificate*, which summarizes the set of authorizations that have been derived from the certificates presented. For efficiency, the Certificate Result Certificate could be MAC'd with a symmetric key which only the server would know, instead of being signed with the server's private key. If the certificate is MAC'd with a symmetric key, only the server that created it would be able to use it. The server verifies future authenticated requests accompanied with a Certificate Result Certificate in exactly the same manner as it verifies requests accompanied with the original certificate sequence. Certificate Result Certificates are described in more detail in Section 3.3.3.

4.2 Additional Security Considerations

The SPKI/SDSI protocol, as described, addresses the issue of providing client access control. The protocol does not ensure confidentiality, nor authenticate servers, nor provide protection against replay attacks coming from the network.

Security protocols are vulnerable to a number of attacks. To attack a protocol, the adversary does not have to have access to either the client or the server; he can launch attacks using the information he overhears from conversations on the network. There are two types of adversaries: a passive adversary who simply records the conversation and analyzes it, and an active adversary who can do what the passive adversary does, and also modify messages as they are being transmitted over the network and replay messages (modified or unmodified) that were previously transmitted. In this discussion, adversaries are assumed to be active.

A brief description of common attacks on protocols follows:

- *Replay attacks:* the adversary records parts of the conversation and replays them later, hoping that recipient treats the replayed messages as new messages. This could trick the recipient into unintentionally divulging protected information, or performing an unintended action. To prevent replay attacks, steps must be taken to ensure the *freshness* of messages: common techniques include using nonces and/or timestamps.

- *Impersonation attacks:* the adversary impersonates one of the principals in the protocol. The *person-in-the-middle* attack is a common variant of this attack: the adversary sits in the middle of a conversation impersonating the principals to each other. For example, if a principal, Alice, say, were to transmit her public key over an unauthentic channel to another principal, Bob, say, the adversary could replace Alice's key with his own. Bob would then be using the adversary's key to encrypt his messages to Alice. The adversary can intercept Bob's messages, decrypt them, re-encrypt them with Alice's key, and forward these messages to Alice. Neither Alice nor Bob may be able to detect that an adversary can read the encrypted messages.

- *Reflection attacks:* if symmetric keys are used for authentication, the adversary can launch a reflection attack by replaying the message to the principal that originally sent it. This attack is typically foiled by including the names of the sender in the message that is MAC'd, or having the symmetric key used to authenticate one principal be different from the symmetric key used to authenticate another principal[28]. If there are two principals in the protocol, using the latter technique, means that two different symmetric keys are shared by the principals.

The Secure Sockets Layer (SSL) protocol is the most widely used security protocol today. The Transport Layer Security (TLS) protocol is the successor to SSL. Principal goals of SSL/TLS[42, 49] include providing confidentiality and data integrity of traffic between the client and server, and providing authentication of the server. There is support for client authentication, but client authentication is optional.

An SSL session is divided into two phases: the *handshake phase*, in which the server is authenticated and symmetric keys are established, and the *data transfer phase*, in which the application's data is transmitted securely. The handshake phase must be completed before the data transfer phase can begin. Among other features, SSL/TLS has:

- *Authentication of Server:* Server authentication in SSL/TLS is based on public-key cryptography. Before initiating an SSL/TLS session, the client obtains, using authentic channels, the public keys of CAs it is willing to trust. The server has a list of X.509 certificates, each issued by a different CA and each binding the server's hostname (which may be in the certificate's DN name[5] or in a special 'dNSName' extension[6] field) to its public key. Near the beginning of the SSL/TLS handshake, the server sends its list of certificates to the client. The client uses the CA keys it trusts to verify the certificates. If one of the certificates verifies correctly, and the hostname in the certificate matches the name of the host to which the client is connecting, the client continues with the handshake; otherwise, the client notifies the user, or terminates the handshake. The next message the client sends during the handshake encrypts a secret with the public key found in the certificate. Only an authentic server would be able to decrypt this message and use the client's secret to successfully complete the handshake. Without the client's secret, an unauthentic server cannot produce and send valid messages to the client; the client will notice and terminate the

[5] described in Section 3.3.1

[6] described in Section 3.3.1

handshake. Thus, server authentication in SSL/TLS includes the server sending the client a certificate; the client verifying that the certificate contains a name it expects, is signed by a CA it trusts, and has not expired; the client sending the server a challenge consisting of a secret encrypted with the certificate's subject's key; and the server proving to the client that it is in control of the corresponding private key by decrypting the challenge, and using it to generate a response to the client. The client and the server each use the client's secret to generate four symmetric keys in a similar manner, and to finish the handshake. The symmetric keys are used to protect client and server messages in the data transfer phase.

- *Confidentiality:* After authenticating the server, TLS/SSL uses the server's public key to help bootstrap into a symmetric-key system during the handshake phase. Symmetric keys are used to encrypt/decrypt messages between the client and server during the data transfer stage. Encrypting and decrypting with symmetric keys is more efficient than encrypting and decrypting with public keys.

- *Data Integrity*: TLS/SSL MACs the messages sent in the data transfer stage using symmetric keys established in the handshake phase.

- *Protection against Reflection Attacks:* The SSL/TLS handshake generates four different, temporary, shared symmetric-keys: one for the client to use to create MACs, one for the client to use to encrypt its messages, one for the server to use to create MACs, and one for the server to use to encrypt its messages. Using these distinct keys foils reflection attacks, and limits the scope of an attacker if he compromises one of the keys. For example, if an attacker were to compromise the key that the client uses for encryption (because the key was too small, say), he would be able to read messages that the client sends, but not be able to read messages sent by the server; he would also not be able to forge either client or server messages.

- *Protection against Person-in-the-Middle Attacks:* Authentication of the server helps protect against a person-in-the-middle attack. Also, before completing an SSL/TLS handshake, the client and server compare the messages that were sent and received. If both of their transcripts are the same, the handshake has not been tampered with.

- *Protection against Replay Attacks from the Network:* TLS/SSL generates symmetric keys that are temporary: they are different between different communication sessions. TLS/SSL also includes nonces (sequence numbers) in the MAC'd data. Using sequence numbers also prevents against re-ordered or deleted messages. These measures protect against an adversary recording traffic as it is sent over the network, and replaying it.

SSL/TLS is not a panacea, however. In particular, some of the things it does not provide include:

- *Nonrepudiation:* Because SSL/TLS uses symmetric keys for authentication during its data transfer phase, it does not provide nonrepudiation for either client or server messages during this phase.

- It is not possible to establish an SSL/TLS session with a machine to which a network connection cannot be established. For example, if the client is behind a firewall and wants to establish an SSL/TLS connection with a server on the other side, either a hole in the firewall must be created, or the client must establish an SSL/TLS session with the firewall, and let the firewall establish its own SSL session with the server. In the latter case, the firewall can read all the conversation between the client and server.

The SPKI/SDSI Access Control Protocol can be layered over a key-exchange protocol like TLS/SSL to provide additional security. TLS/SSL currently uses the X.509 PKI to authenticate servers, but, it could just as well use the SPKI/SDSI PKI in a similar manner to authenticate servers. With these considerations, the layering of the protocols, with respect to HTTP, is shown in Figure 4-2. A systems designer may be tempted to optimize the SPKI/SDSI Access Control Protocol if it is going to be layered over SSL/TLS; if so, he should state explicitly in his design specifications exactly what he will be trusting the SSL/TLS keys to do.

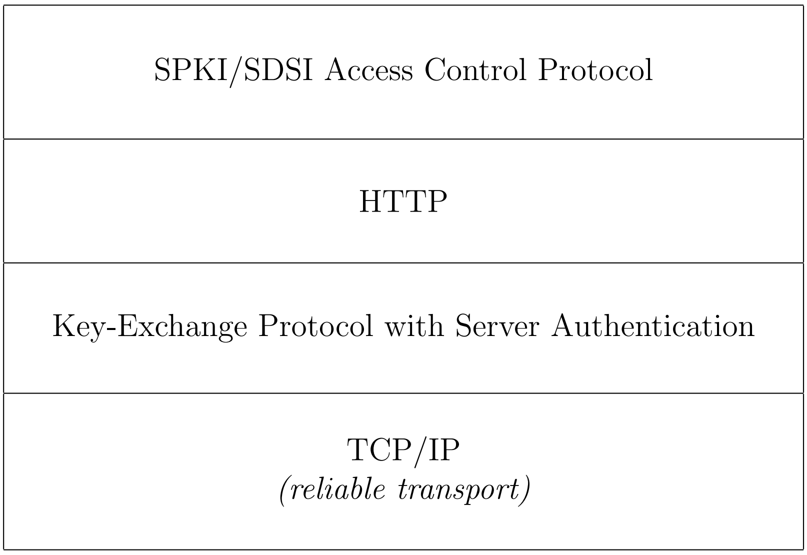

SPKI/SDSI Access Control Protocol
HTTP
Key-Exchange Protocol with Server Authentication
TCP/IP
(reliable transport)

Figure 4-2: Layering of Protocols

Note that the SPKI/SDSI Access Control Protocol is an example of the *end-to-end argument*[39, 31]. The client access control decisions are made in the uppermost layer, involving only the client and the server. Best stated by Saltzer *et al.*, the end-to-end argument is:

> "The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)"[7]

[7] J.H. Saltzer, D.P. Reed and D.D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), Nov. 1984, page 278. See `http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf`. Last visited 08/07/2001.

4.3 Possible alternative protocol design

A possible alternative approach to our protocol design is to incorporate SPKI/SDSI-based client access control into the SSL/TLS protocol. I believe that there are problems with this approach. In SPKI/SDSI, objects are protected with SPKI/SDSI ACLs. In our design, the web server's directories are the objects; each protected directory is protected by an ACL that is referenced in the directory's .htaccess file. When the client makes a request to perform a particular operation in a particular protected directory on the server, the server sends the appropriate ACL to the client so that the client knows which SPKI/SDSI certificates to send in order for its request to be honored.

SSL/TLS provides confidentiality and data integrity of traffic between the client and server, and authenticates the server. If SPKI/SDSI client access control is incorporated into the SSL/TLS protocol, then it seems that one of the following cases applies:

- the SSL/TLS handshake will need to be renegotiated at least each time the client makes a request to access an object in a new directory; the handshake needs to be renegotiated because the client needs to know the ACL protecting the new directory if it is going to avoid sending all of the certificates in the user's cache. This case is clearly infeasible as each renegotiation of SSL/TLS is computationally expensive. Furthermore, why should changing directories on the same server require changing the keys that are used to protect messages to and from that server?

- the client will need to send all of the certificates in the user's cache during SSL/TLS handshake and the server will need to maintain state representing the authorizations belonging to a particular client; with each request from a particular client, the server checks if the request should be honored, using the authorizations the server associates with the client, and the ACL protecting the appropriate directory. In this case, the client is sending, potentially, alot of unnecessary certificates to the server, and the server is doing alot more work as it needs to determine all of the authorizations for each client connecting to it as well as maintain more state, beyond the state it maintains for the SSL/TLS session.

- the server will need to have one ACL protecting access to the server. The server sends this ACL as part of the SSL/TLS handshake, and the client sends a certificate chain proving that it is authorized to access the server. In this case, the protected object is the server, and not an individual directory. SPKI/SDSI features the ability to specify fine-grained authorizations in tags; this feature is all but lost if the server has just one ACL protecting access to it.

In my opinion, the issue is that SPKI/SDSI client access control decisions should be made as close to the top of the protocol stack as possible. This is an instance of the end-to-end argument stated in Saltzer, Reed, and Clark [39] and described in Section 4.2. With reference to the TCP/IP protocol stack[47], if the client access control decisions are made in TLS/SSL, they will be made in the transport layer, the layer in which TLS/SSL provides security. *If ACLs are specified in directories, not all of the information that the client needs to derive its proofs of authorization is necessarily present in the transport layer: at the time the SSL/TLS handshake is being established with the server, the client does not know all of the directories the user will be visiting during that particular SSL/TLS session; therefore, it does not know which particular user certificates to send to the server.* In our design, the client access control decisions are made in the application layer, the topmost layer in the TCP/IP protocol stack. In the application layer, the client will have all of the information it needs to derive its proofs of authorization. The SPKI/SDSI Access Control Protocol is layered on top of the application level protocol (HTTP in this case). HTTP is layered on top of (a protocol like) SSL/TLS. With this layering of protocols, the specific functions of each protocol is clearly defined. SSL/TLS authenticates the server to the client, and provides a confidential channel with data integrity between the client and the server; the SPKI/SDSI Access Control Protocol allows the server to determine if a particular client request should be honored.

In addition, incorporating SPKI/SDSI client access control into SSL/TLS using any one of the approaches previously described in this section seems to be a non-trivial change to the SSL/TLS protocol. It is not clear to me that such a change can be designed and implemented easily and securely.

In summary, I think that SPKI/SDSI may be incorporated into SSL/TLS for the purpose of server authentication. Perhaps, the client's public key may also be authenticated (perhaps in a manner similar to that of SSH, described in Section 4.4.2) within SSL/TLS, so that the client does not have to sign every request. However, SPKI/SDSI client access control decisions should be made in the application layer using the ACL protecting the requested object.

4.4 Comparisons

This section compares SPKI/SDSI-based client access control with three methods: password-based access control on the web, Secure Shell (SSH), and access control based on X.509 certificates.

4.4.1 Password-based access control on the Web

Client access control based on passwords is the most common method in use on the web today. Each user shares a password with the server. Initially, the user sets up an account on the server, and protects access to his account with his password. He logs-in to the server using a username-password pair. The server uses the username to identify the account, and matches the password provided in the login request with the password protecting the account. If they match, the user is allowed access to the account; if they do not match, the user is denied access. SSL/TLS supports password-based access control well, as accounts can be securely established, and users can securely log-in, over SSL/TLS connections[8].

The scheme can be modelled using the ticket-oriented guard model[9]. The object

[8] It should be noted that there are other password-based authentication schemes, such as SRP[48].

[9] described in Section 2.4

is the account, the guard is the login program, the principal is the user, and the token is the password. If the token the user presents is the same as the token the guard has, the user is allowed to access the account; otherwise, access is denied.

One of the problems with password-based access control is that users need to trust the server, and the server administrator, to not disclose or use their passwords. If a user is using the same username-password pair to access multiple online services, the server administrator at one of the services is able to use this information to access the user's account on another service. For example, if Alice has email accounts on Hotmail and Yahoo, and she uses the same username-password pair to access both services, a malicious Hotmail administrator, if he correctly guesses that she has a Yahoo account, can log into her Yahoo account, and vice-versa.

If a user desires protection against this problem, he is forced to remember different username-password pairs, which gets increasingly more difficult as the number of online accounts he uses increases. Besides being an incredible hassle having to remember several different passwords, there are also significant security risks. If the user uses different passwords, he may use passwords which may be easy for him to remember. Such passwords tend to have low entropy (are easy to guess). Furthermore, if he accesses different accounts frequently, he may mistakenly enter the password for one account when trying to login to another account; in the case in which the accounts are on different servers, and the servers are logging login attempts, the first server captures the password used on the second server.

If public-key cryptography is used, there is no shared secret between users and servers. Users can securely use the same public key with many different accounts. A user will still have to enter a password, but in this case, the password is used to unlock his private key on the client so that it can be used to sign requests. The user's password is never transmitted across the network, and the same password can safely be used to access many different accounts. Besides the security benefits, there is the added convenience that if the user wants to change the password he uses to access his accounts, he just has to change the password locking his private key, instead of individually having to change the password protecting each account. Even if the user were using different passwords to enter different accounts, because he is using different key-pairs, say, as passwords remain on the client, submitting an incorrect password cannot result in a breach of security.

Another problem with many password-based implementations is that servers usually save state, in the form of HTTP "cookies", on clients. The server typically generates and sends the cookie to the client when the user successfully logs in. The client then sends this cookie with future requests to the server. This can be used to prevent the user from having to re-type his password each time he wants to access his account within the same client session. The user's encrypted password may be in the cookie, though it does not have to be; for example, instead of the encrypted password, the cookie could contain a randomly-generated shared secret. If the server uses "transient cookies", cookies that are stored in memory on the client, the cookies expire when the web browser is closed, and the user is required to log-in again if he wants to access his account in a new client session. There are a number of security risks with using "cookies". Perhaps the most well-known problem is that cookies can be used to track and record users' web browsing habits, which is usually regarded as a privacy violation. Publications on the security risks of cookies are provided by the World Wide Web Consortium[40], and the MIT Lab for Computer Science's Applied Security Reading Group (ASRG)[20].

The SPKI/SDSI web client also saves state to prevent a user from having to re-enter his password during the same client session. This state is created and stored by the client only, and is never transmitted across the network. The scheme is implemented using a client "session window", and is well documented in Andrew Maywah's Master's thesis[33].

Another significant difference between the two client access control schemes is that, in password-based access control, only the password needs to be compromised; in SPKI/SDSI, both the password and the private key that it protects need to be compromised. As seen from the discussions in the previous paragraphs in this section, there are several ways to attack the password in password-based schemes; in SPKI/SDSI, essentially, the client machine must be compromised, and the private key unlocked, to successfully attack the scheme.

If users just use passwords to protect accounts, creating a group in which each member can access a particular account means either putting each member's username-password pair on the ACL's entry for that account, or having each group member use the same username-password pair. The first case increases the complexity of ACL management, which is a security risk. The second case is patently a security risk; for example, if passwords are shared, accounting becomes difficult because it is difficult to distinguish between different users. The same problems exist with 'delegation', where one user would like to delegate to another user the authority to decide who can access a particular account. SPKI/SDSI provides a clean model for creating groups and delegation without compromise of security. The infrastructure also facilitates easy ACL management. Accounting is also possible as users will always use different signatures to authenticate themselves. These features are described in Chapter 3.

One advantage of password-based schemes are that they are easy to set up, especially over SSL/TLS. When a user sets up an account, he can register his username-password pair with the server securely over an SSL/TLS connection. In SPKI/SDSI-based client access control, users need to be issued certificates; this process must include authentic rendezvous between the certificate issuers and the user, so that the issuers know that the key for which they issue the certificates is authentic.

In summary, as the models are scaled, there is a tradeoff between ease of establishment and security with password-based access control and SPKI/SDSI-based access control. In password-based schemes, each new account the user sets up, or is given access to, means potentially a new password he has to remember. In SPKI/SDSI, each new account the user sets up, or is given access to, means new certificate(s) for the user, but he can still safely use the same password and key-pair. In password-based schemes, security does not scale as more accounts are established. SPKI/SDSI was designed to facilitate scalability without compromise of security. Table 4.1 on page 102 summarizes the comparison of password-based and SPKI/SDSI-based client access control.

Password	**SPKI/SDSI**
Need to compromise a password	Need to compromise a private key and a password
Password sent across network from client to server	Neither private key nor password ever sent across the network
Each new account the user sets up, or is given access to, means potentially a new password he has to remember.	Each new account the user sets up, or is given access to, means new certificate(s) for the user, but he can still safely use the same password and key-pair.
Cannot securely use the same password with different accounts: if the server's admin is untrustworthy/the server is compromised, an adversary can learn the password protecting one account, and use it to access other accounts if they are protected by the same password.	Can securely use the same password with different accounts: if the server's admin is untrustworthy/the server is compromised, the adversary is still unable to access other accounts.
If a user uses different passwords with different accounts, it is a hassle having to remember several different passwords, and there are still security risks: he may choose passwords with less entropy that are easier for him to remember; also, if he mistakenly enters the password for one account when trying to login to another account, the first server captures the password used on the second server if servers are logging login attempts.	Can securely use the same password with different accounts. If different passwords are used, because different key-pairs are used, say, submitting an incorrect password cannot result in a breach of security as passwords remain on the client.
Creating 'Groups' or 'Delegating' means giving others your password, and/or, constantly updating all the relevant ACLs. If passwords are shared, accounting is difficult because it is difficult to distinguish between different users.	Clean model for creating groups and fine-grained delegation without compromise of security; the model facilitates easy ACL management. Accounting is also possible as users will always use different signatures.
Easy to understand. Easy to set up, especially over SSL/TLS.	Principal needs to be issued certificates; this process must include authentic rendezvous between the certificate issuers and the principal, so that the issuers know that the key for which they issue the certificates is authentic.

Bottom Line

Easier to set up	More secure

Table 4.1: Comparison of Password-based and SPKI/SDSI-based Client Access Control

4.4.2 Secure Shell (SSH)

A popular access control system in use on the Internet today is Secure Shell (SSH)[1]. SSH allows a user to securely login over an untrusted network. It creates a secure channel to a shell running on a remote host computer. The user runs a *window*[10] on his local machine which provides a command line interface to the shell on the remote host. He types commands in to the window, and they are sent across the network over the secure channel, and executed in the shell on the remote computer. There are two access control systems at work: SSH protects access control to accounts, and the filesystem restricts the commands a user can execute once the account has been accessed[11].

The SSH protocol is very similar to the SSL/TLS protocol. The SSH client runs on the user's local machine, and the SSH server runs on the remote machine to which the user wishes to login and run the shell. The server is authenticated using public-key cryptography, after which the server's public key is used to establish a secure, symmetric key-based, communication channel between the client and server. In contrast to SSL/TLS, client authentication is required in SSH. SSH is an application, like telnet, ftp, or DNS (Domain Name System); SSL/TLS was designed to be incorporated into applications[12]. SSL/TLS is more transparent than SSH. One advantage of SSL/TLS is that one does not have to be logged in to a user account to set up a secure channel[47].

There are several ways in which a client may authenticate itself to the server in SSH, but this discussion will focus on only two: client authentication using passwords, and client authentication using public keys.

[10] for example, a *terminal window*: `http://www.ssh.com/products/ssh/winhelp22/Terminal_Window.html, last visited 08/07/2001`.

[11] With SSH's *forced commands*[1] feature, the set of programs that a user may execute in an account can also be limited.

[12] Using the TCP (Transmission Control Protocol) protocol stack[47], SSH provides security in the application layer; SSL/TLS provides security in the transport layer, hence its name: Transport Layer Security (TLS).

Client Authentication using Passwords

This is very similar to the preceding discussion in Section 4.4.1 on page 98. The user supplies a username and password to the SSH client, which the client sends to the server over the secure channel. The username is used to identify the account, and the password provided in the login request is matched with the password protecting the account. If they match, the user is allowed access to the account; if they do not match, the user is denied access. In the common case, the SSH server uses the password-authentication system of the host computer to perform its check. The comparison of this access control system with a SPKI/SDSI-based system is the same as that in Section 4.4.1.

Client Authentication using Public Keys

If a user wanted to authenticate himself using his public key, he would first add his key to his account's *~/.ssh/authorized_keys* file on the remote host computer. (If more than one user were to have access to the same account, their public keys would all have to be placed in that account's *~/.ssh/authorized_keys* file.) An overview of the access control protocol follows:

1. The client sends the server a request for public-key authentication. This request contains the user's public key.

2. The server reads the target account's *~/.ssh/authorized_keys* file. If the key presented in the request matches a key in the file, the protocol continues. Otherwise, authentication fails.

3. The server generates a random 256-bit string, encrypts it with the user's public key, and sends this to the client. This is the server's challenge to the client.

4. The client receives the challenge, and decrypts it with the user's private key to retrieve the 256-bit string. It combines the string with the session identifier, a 128-bit string which uniquely identifies the particular SSH session, and hashes

the result. The resulting value is sent to the server as a response to the server's challenge.

5. The server independently computes the response to its challenge. If the client's response matches the value the server computed for the response, authentication of the client succeeds. Otherwise, authentication fails.

This design can be modelled using the list-oriented guard model[13]. The object is the account, the guard is the server's access-check program, the ACL is the *~/.ssh/authorized_keys* file, the principal is the user, and the token is the public key. In comparison with SPKI/SDSI, which is also an example of the list guard model, SPKI/SDSI offers more flexibility allowing a user to be able to easily, and securely, define a group of people who can access a particular account without having to update the account's ACL, and/or delegate to another user the authority to decide who can access the particular account. Because of SPKI/SDSI's *delayed binding*[14] feature, the ACL can be modified once, and a SPKI/SDSI name/group added to it; after it becomes clear which user/users should be authorized to access the account, the relevant certificates are issued to the user/users giving them access without having to update the ACL.

4.4.3 X.509-based access control

An alternative to providing client access control using SPKI/SDSI certificates is using X.509 certificates. Instead of providing a chain of SPKI/SDSI certificates as a *chain of authorization*, the client provides a chain of X.509 certificates as a *chain of authentication*. The principal differences between the two schemes essentially reduce to the differences between the two infrastructures. These differences are described in Chapter 3, and summarized in Table 3.1 on page 81.

[13] described in Section 2.4

[14] described in Section 3.3.3

Chapter 5

Implementation

This section describes, in detail, the implementation of the SPKI/SDSI HTTP server, and gives a brief overview of the client. The details of the client can be found in Andrew Maywah's Master's Thesis [33]. Both the server and client use Matt Fredette's SPKI/SDSI's C library[19] to create and verify SPKI/SDSI Sexps. This section also gives an outline of the demo we designed and developed for Project Geronimo.

5.1 Server Implementation

The Apache web server was extended to handle SPKI/SDSI-based access control. An Apache "module"[41, 32] was used to incorporate SPKI/SDSI into the Apache web server. The Apache web server was chosen because it is open-source, very popular, and facilitates extensibility with a modular architecture and a well-defined Application Programming Interface (API).

Apache provides an API for developing modules that can creatively extend the server's core capabilities. Examples of common modules include those that execute CGI (Common Gateway Interface) scripts, rewrite URLs dynamically, and provide access control based on the client's hostname or IP address. The principle functions of the SPKI/SDSI module are to protect web objects using SPKI/SDSI ACLs, and to determine whether HTTP client requests should be permitted to perform particular operations on protected objects.

Web objects on the server are protected on a per-directory basis. To protect a directory, a .htaccess file is created in it. The file contains the directive "SPKI/SDSI on", and pointers to the file with the SPKI/SDSI ACL and the customizable error page that is returned if the user's credentials do not grant him access. File pointers are relative to the server's document root. An example of a .htaccess file is presented in Figure 5-1. Directories which do not have .htaccess files are public directories.

```
SPKI/SDSI on
ACLFile demo/spki-sdsi-acls/ABC_financial_acl
ErrorFile demo/ABC/public/error.html
```

Figure 5-1: An example of a .htaccess file

The customizable error page is an html page which is set up by the server's administrator. It is specified on a per-directory basis in the .htaccess file. It is useful since, on this web page, the administrator can post information that he/she believes would be most helpful to a valid user trying to obtain the correct credentials. The feedback presented on this page can accelerate the process through which a new user gains valid authorizations. There are six replacement stubs that may be used in the page: #REPLACE_DOCUMENT_URL#, #REPLACE_TAG#, #REPLACE_TAG-TIMESTAMP_SEQUENCE#, #REPLACE_SIGNATURE#, #REPLACE_CERTIFICATE_SEQUENCE#, and #REPLACE_ACL#. Before the server returns the error page to a client in response to a particular denied request, it replaces each instance of #REPLACE_DOCUMENT_URL# with the requested document's URL, #REPLACE_TAG# with the tag the server creates from the client's request, #REPLACE_TAG-TIMESTAMP_SEQUENCE# with the tag-timestamp sequence the client sent with its request, #REPLACE_SIGNATURE# with the signature the client sent with its request, #REPLACE_CERTIFICATE_SEQUENCE# with the certificate sequence the client sent with its request, and #REPLACE_ACL# with the ACL protecting the directory the client is trying to access. If the administrator does not want a particular object to appear on the page returned to the client, he does not put the replacement stub for it in the page on disk. The page can also contain the contact information of persons/offices that would be most helpful in resolving user concerns. The user should be able to print the error page that is returned to him, and take the printout to the relevant office. The information on the printout should be a good starting point in helping to debug the request. The customizable error page facilitates the feedback process of valid users who are being denied access.

The module is implemented in C, using the Apache API. In processing a request, the Apache server traverses through several phases, each corresponding to a different decision it must make. For example, in its "URI translation" phase, the actual file path of the requested file is determined. As another example, in the "MIME type checking phase", the server determines the requested document's content-type, which will be used to indicate to the client which application should be used to handle the document. The SPKI/SDSI module is designed and implemented to interrupt Apache only in its access control phase, and not affect the other functions that may be called to process the request. Thus, the module is an access control Apache module.

As described in Chapter 4, the server is stateless. It interprets each request using only the information presented in the request itself, and not using information in previous requests. The responses the server returns depending on the request it receives are detailed in Chapter 4.

5.1.1 Server Verification Process

To verify a request, the module uses five Sexps:

tag Sexp the module creates the tag from the client's request, using the request's URL, protocol, and method. An example of a tag that would be created from a client's request is given in Figure 5-2.

ACL Sexp the module retrieves the ACL protecting the directory. The ACL is stored in the file pointed to by the AclFile directive in the directory's .htaccess file.

tag-timestamp sequence Sexp the module uses the tag-timestamp sequence Sexp the client sends with its request.

signature Sexp the module uses the signature Sexp the client sends with its request.

certificate sequence Sexp the module uses the certificate sequence Sexp the client sends with its request.

```
(tag
 (http
  GET
  http://rooster.lcs.mit.edu:8081/demo/ABC/financial/budget.html))
```

Figure 5-2: A tag formed from a client's request

The module's verification process is outlined in Figure 5-3 on page 112. The design is completely open, and the client can simulate the entire server verification process itself and determine whether its request will be honored or not. The process consists of the following steps:

1. The module checks the timestamp in the tag-timestamp sequence against the time in the server's local clock to ensure that the request was made recently. In our prototype implementation, the module checks that the timestamp is within five minutes of the server's local time i.e.
server's local time $-$ timestamp $<$ 5 minutes.

2. The module recreates the tag from the client's HTTP request and checks that it is the same as the tag in the tag-timestamp sequence.

3. The SPKI/SDSI signature Sexp contains in it, the public key corresponding to the private key that was used to create it. The module verifies that the signature is a valid signature on the tag-timestamp sequence using the public key in the signature. If the signature verifies, the request is authentic.

4. Each individual certificate in the certificate sequence is validated by checking the signature on the certificate and checking that it does not fail its validity

specification (e.g. the certificate has not expired). If any certificate in the certificate sequence fails its signature verification or fails its validity specification, the entire certificate sequence is rejected and the client is denied access.

5. After validating the individual certificates, the module then verifies that the certificate sequence presented provides a chain of authorization from the ACL's issuer to the key that was used to verify the signature. If the chain of authorization is valid, the request is authorized. The module honors the request after it has determined that it is authorized.

The server, as implemented, does not perform any certificate chain discovery. It just verifies the certificate chain the client presents. If the chain does not verify, the request is not authorized, and is denied; if the chain verifies, the request is authorized, and is honored. The process by which authorization chains are verified in step 5 follows. Briefly, the module first checks that the client request tag, the tag from the tag-timestamp sequence, is a subset[1] of the tag in each of the authorization certificates in the certificate sequence. If there is an authorization certificate whose tag is not a superset of the client request tag, the certificate sequence is invalid, and the verification procedure fails. The module then parses the ACL, parsing each ACL entry into an unsigned authorization certificate with the issuer "`SELF`". ACL entries whose tags are not a superset of the client request tag are ignored. If there is at least one entry whose tag is a superset of the request tag, the procedure continues; otherwise, the request is not authorized, and the procedure fails. The module then parses each name certificate and authorization certificate in the sequence, and each of the remaining ACL entries, into *rewrite rules*[2]. For each ACL entry, rule rewrites are conducted in the order indicated in the certificate sequence. If, for at least one ACL entry, the rule rewrites result in a rule in which the public key used to verify the request's signature is derived from "`SELF`", the request is authorized, and the verification procedure returns 'success'; otherwise, it returns 'failure'. There are specific

[1] Tag intersection is described in the SPKI/SDSI IETF drafts[11].

[2] described in Section 6.1.1

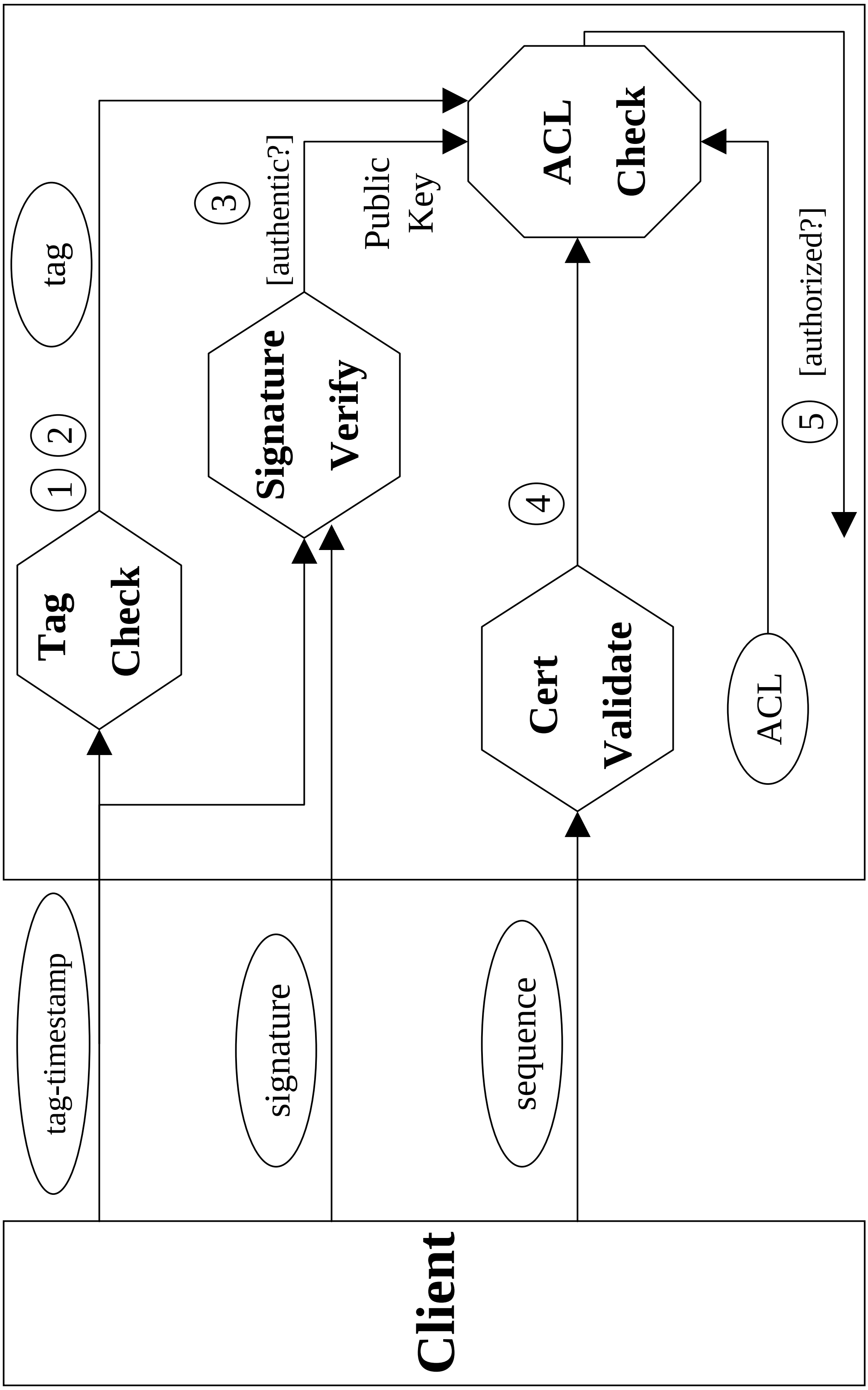


Figure 5-3: Server Verification Process

formats which the client must use to form its “proof of authorization”, so that the server can easily, and deterministically, verify the proof.

5.1.2 ACL Manager

A CGI-based tool was developed to enable an administrator to create, view and update ACLs securely. ACLs are stored in a central directory on the server. The tool enables the administrator to log into it remotely via password-based authentication over a secure SSL connection, view the ACLs already on the server, edit existing ACLs, and create new ACLs. As the server is hosting web objects, the tool’s GUI is HTTP specific making it simple and intuitive to understand and use.

5.2 Client Implementation

The SPKI/SDSI web client was designed and implemented as a Netscape plugin. A plugin is a small shared library of procedures that Netscape Communicator calls when it encounters a document with an appropriate MIME type. Implementing the client as a plugin offered such benefits as portability, simplicity, and ease of development. The plugin is written in C and developed on Unix platforms.

The behavior of the plugin is as follows:

1. First, the user selects a link which sends a regular “GET” request to the server. If the requested document resides in a SPKI/SDSI protected directory, the server rejects the request by sending back a message containing the ACL and tag, with content-type “application/x-spki-sdsi”.

2. Upon receiving the message with content-type “application/x-spki-sdsi”, Netscape initializes the plugin and provides a copy of the message to the plugin.

3. During initialization, the plugin prompts the user for his password to unlock his private key using a small Java-based pop-up password box. The password is used by the client alone and is never transmitted across the network. If the

private key is successfully unlocked, a small session window appears next to the Netscape browser. This session window maintains state between client requests during the same Netscape session. This state is created and maintained by the client only, and is never transmitted across the network. It prevents the user from having to re-enter his password every time he accesses a SPKI/SDSI protected document within the same Netscape session.

4. Using the ACL and tag from the server, and the user's public key and certificate cache, the plugin generates a sequence of certificates using the SPKI/SDSI *certificate chain discovery algorithm.*

5. The plugin generates a timestamp using the client's local clock. It creates a SPKI/SDSI sequence consisting of the tag from the server, and the timestamp it generated. It signs this sequence with the user's private key. A copy of the user's public key is included in the signature. The plugin then sends the tag-timestamp sequence, its signature, and certificate sequence generated in the previous step to the server.

6. If the second request successfully verifies, the server returns the requested document to the client, which it displays. If the second request fails to verify, the server returns an error page to the client, which it displays.

5.3 Demonstration

The principal goal of our work on Project Geronimo was to develop a demo illustrating some of the capabilities and advantages of the SPKI/SDSI Infrastructure. This demo was successfully implemented using the SPKI/SDSI web client and server. It featured a new user, Alice, who had already generated her SPKI/SDSI key-pair and installed the plugin, going through the process of gaining authorization credentials to view web pages to which she should be permitted to access.

The background of the demo is that ABC Corporation, a fictitious firm, is going to be audited by Auditors Inc., another fictitious firm. Alice is among the team of auditors from Auditors Inc. who have been assigned to the task. She needs authorization to view ABC Corp's financial records (web pages). ABC Corp's web pages are hosted on Bob's web server, and the financial ones are protected by the SPKI/SDSI ACL: "ABC_financial_acl".

The demo consisted of the following steps:

1. ABC Corp's financial pages are kept in "demo/ABC/financial/". Bob edits the financial ACL, "ABC_financial_acl", to add an entry with the group "ABC_auditors" with authority to perform 'GET' requests to this directory. He uses the administrative ACL manager to edit the ACL and logs into it using password-based authentication over a secure SSL connection.

2. Alice first tries to access pages in one of ABC Corp's public directories. She is successful.

3. Alice then tries to access the "Budget" page in the financial directory. As this is the first time she is trying to access a document in a SPKI/SDSI protected realm in the current Netscape session, the plugin pops up a password dialog box and asks her for her SPKI/SDSI password. It uses this password to unlock Alice's private key, and, if successful, starts the client-side SPKI/SDSI session window. As Alice does not have the proper credentials for the "Budget" page, she is denied access when she tries to access it. Bob's customizable error page is displayed with instructions to contact him for the necessary certificates.

4. Alice copies her public key to a floppy disk, and takes it to Bob[3].

5. Bob creates a name certificate for Alice's key, binding Alice's public key to Alice's name. He binds Alice's key to a name within his local name space, demonstrating SPKI/SDSI's local name space architecture.

6. Bob creates a name (group membership) certificate binding his Alice to his group "ABC_auditors". This gives Alice access to his "demo/ABC/financial/"

[3] In our demo, Alice took her public key to Bob on a floppy disk; of course, Bob can obtain Alice's key through any authentic channel.

directory. This certificate demonstrates SPKI/SDSI's ability to create groups (and also SPKI/SDSI's local name space architecture).

7. Bob copies these certificates onto Alice's floppy disk.

8. Alice takes the floppy disk, and puts these certificates in her cache.

9. Alice now tries to access the Budget protected document. The Plugin sends a signed request with the new certificates in a chain; she is successful and receives the file.

10. Alice then tries to access another protected document in the financial directory (the "Accounts" page). For this subsequent attempt, the plugin doesn't need to prompt Alice for her password. This allows much faster access, and illustrates the functionality of the client-side session window.

11. Alice, however, is still denied access from the "demo/ABC/minutes/" directory, which is protected with "ABC_board_meeting_minutes_acl". This ACL only has the "ABC_executive_committee" group on it. This demonstrates fine-grained discretionary access control as Bob is able to authorize Alice to view only those documents that she is required to access.

The SPKI/SDSI objects used in the demo are presented in the appendix. The demo featured SPKI/SDSI groups and name certificates, ACL administration, the client-side password box and session window, the server-side customizable error page, certificate chain discovery, and discretionary, fine-grained access control over an untrusted network. The web medium used for the demo made it attractive and more interesting, and it was successfully presented to several representatives from various groups in the MIT Laboratory for Computer Science.

Chapter 6

Certificate Chain Discovery

6.1 Introduction

In Project Geronimo, the certificate chain discovery algorithm is used by the client to derive a *proof of authorization* that demonstrates that the principal is allowed to perform the operation it requests on the object protected by the ACL that the server sends to the client. The client is the prover and the server is the verifier. In a simplified protocol:

1. The prover sends the verifier a request to access a protected object.

2. The verifier replies to the prover with a challenge containing the object's ACL.

3. The prover sends the verifier a second request, with a digital signature providing proof of authenticity of the request, and a certificate chain providing proof that the prover is authorized to access the object. This second request is the prover's response to the verifier's challenge.

4. The verifier verifies the second request. If it verifies, the verifier returns the object to the prover.

Certificate chain discovery is used to derive the proof of authorization in going from step 2 to step 3. The certificate chain discovery algorithm takes as input an ACL, a tag, a public key, a set of signed certificates, and a timestamp. If it exists, the algorithm returns a certificate chain, consisting of signed certificates, which provides proof that the public key (principal) is authorized to perform the operation(s) specified in the tag on the object protected by the ACL, at the time specified in the timestamp. Thus, in going from step 2 to step 3, the prover runs the certificate chain discovery algorithm on an input consisting of the ACL the verifier sent to it, a tag formed from its request (which the verifier may also have sent in the verifier's challenge), its public key, the set of certificates in its cache, and the current time.

This chapter describes the intuition of the algorithm for discovering certificate chains in SPKI/SDSI. The full mathematical description of the algorithm, along with the formal proof of why it works, is detailed in the journal paper "Certificate Chain Discovery in SPKI/SDSI"[3].

Recall that each entry of an ACL can be considered to be an authorization certificate with the issuer being the special designator "`SELF`", representing the owner of the ACL, and the subject, tag and delegation bit being as specified in the entry. The validity specification is as specified in the entry if it is specified; if the validity specification is not specified in the entry, the entry, and thus, the corresponding 'authorization certificate', is assumed to be valid for the time period $-\infty$ to $+\infty$. In this chapter, 'certificates' from ACL entries are referred to as *ACL certificates*: ACL certificates are authorization certificates, and can be distinctly recognized as they have the issuer `SELF`. The input certificates belonging to the user are referred to as *user certificates*: user certificates can be either name certificates or authorization certificates; as the issuer of the certificate is always some public key, user certificates can never have the issuer `SELF`.

The first step in the algorithm is to remove ACL and user certificates that will be useless in deriving the proof of authorization. In particular, the algorithm's first step is to remove

- Each invalid certificate. If a certificate's signature does not verify or the certificate fails its validity specification, it is removed.

- Each authorization certificate whose authorization tag is not equal to, or does not include, the input authorization tag. These certificates are useless in trying to derive the desired certificate chain[1].

For the rest of this chapter, it is assumed that the preceding useless certificates have been removed from the set of input certificates.

6.1.1 Rewrite Rules

We introduce the concept of SPKI/SDSI "rewrite rules"[3]. Rewrite rules are derived from SPKI/SDSI name certificates and authorization certificates. As we further develop our framework in subsequent subsections, it will become clear why we use the term "rewrite rules".

Recall that a name certificate that defines the local name[2] "K `A`", where K is the issuer's key, to be the subject, "`S`", can be denoted as "K `A` $\longrightarrow$ `S`". As an example, if K_A is Alice's public key, and K_B is Bob's public key, Alice can issue a name certificate, K_A `friends` $\longrightarrow K_B$, defining the name "K_A `friends`" to be Bob's key. "K `A` $\longrightarrow$ `S`" represents one of the two types of SPKI/SDSI rewrite rules.

Recall that an authorization certificate in which the issuer, "K", grants the authorization specified in the tag, "`T`", to the subject, "`S`", with the delegation bit, "p", and a validity specification, "`V`", is represented by the 5-tuple, "(K, `S`, `T`, p, `V`)". The value of "p" is either true or false. ACL entries are assigned the special issuer "`SELF`", the principal representing the owner of the ACL. Thus, each entry on an ACL has a 5-tuple representation (`SELF`, `S`, `T`, p, `V`). As an example, if K_A is Alice's public key, and K_B is Bob's public key, Alice can issue an authorization certificate, (K_A, K_B, T_1, true, V) granting Bob the authorization specified in T_1 with the permission to delegate this authorization. As another example, (`SELF`, K_B `friends`, T_2, false, V) represents an ACL entry with the group "K_B `friends`" on it; the members of this group are allowed to perform the operations specified in T_2, but are not allowed to grant this authority to anyone else.

[1] Tag intersection is described in the SPKI/SDSI IETF drafts[11].

[2] described in Section 3.3.3

For the certificate chain discovery algorithm, as we are only considering valid certificates whose authorization tags are a superset of the input tag, the validity specifications and tags in the 5-tuples do not matter.

Definition: *ticket* *The special "ticket"[3] symbols are "□" ("a live ticket") or "■" ("a dead ticket"). The meta-symbol "$\boxed{z}$" may be used to represent a "zombie" ticket that may be either live or dead.*[3]

The ticket "□" is considered to be "live" - it represents a permission obtained with the delegation bit turned on, so it can be further delegated. The ticket "■" is considered to be "dead" - it represents a permission obtained with the delegation bit turned off, so it cannot be delegated further. To represent a ticket that may either be live or dead, we use the meta-symbol "$\boxed{z}$", the "zombie ticket". The zombie ticket doesn't actually appear in rewrite rules, but is used when discussing a rewrite rule having a ticket which may be either live or dead.

We can now represent an authorization certificate, $(K, \texttt{S}, \texttt{T}, p, \texttt{V})$, as a rewrite rule. If the delegation bit, p, is true, allowing propagation, the authorization certificate can be represented as the rewrite rule: "$K\ \square \longrightarrow \texttt{S}\ \square$". If the delegation bit, p, is false, so that delegation is forbidden, the authorization certificate can be represented as the rewrite rule: "$K\ \square \longrightarrow \texttt{S}\ \blacksquare$". If, for theoretical discussion, the delegation bit can be either true or false, the authorization certificate can be represented as the rewrite rule: "$K\ \square \longrightarrow \texttt{S}\ \boxed{z}$". The ticket on the left of a rewrite rule derived from an authorization certificate is always live. The ticket on the right is live if the delegation bit p is true; otherwise it is dead.

In the previous examples, the authorization certificate (K_A, K_B, T_1, true, V) can be represented as $K_A\ \square \longrightarrow K_B\ \square$. Similarly, the ACL entry ($\texttt{SELF}$, K_B $\texttt{friends}$, T_2, false, V) can be represented as $\texttt{SELF}\ \square \longrightarrow K_B\ \texttt{friends}\ \blacksquare$.

[3] *In ASCII text we suggest using "[-]" for the live ticket, "[X]" for the dead ticket, and "[*]" for the zombie ticket. The Certificate Chain Discovery paper uses "$\boxed{1}$" for the live ticket, "$\boxed{0}$" for the dead ticket, and "$\boxed{z}$" for the zombie ticket.*

Definition: *term* *We say that a term[3] is either a key or a SPKI/SDSI name (recall that a SPKI/SDSI name is a key followed by one or more identifiers[4]).* Examples of terms are: K_A and K_B `Alice friends`.

Definition: *string* *We say that a string[3] is either a term or a term followed by a ticket.* Examples of terms are: K_A, K_B $\blacksquare$, K_B `Alice friends`, and K_A `MIT EECS faculty` $\square$.

Thus, to summarize, there are two types of rewrite rules. The rewrite rule's type can be determined from the rule itself:

- Rewrite rules derived from (user) name certificates:
  These rules have no tickets. The string on the left of the rule is always a SPKI/SDSI local name, i.e. a key followed by a single identifier. The string to the right of the rule may be a public key or a name consisting of a key followed by one or more identifiers.
  K_A `friends` $\longrightarrow$ K_B `Carol_Jones Ted` is an example.

- Rewrite rules derived from (user and ACL) authorization certificates:
  The string on the left of the rule is always a public key or `SELF` followed by a "$\square$" ticket. The string to the right of the rule may be a public key or a name consisting of a key followed by one or more identifiers, followed by either a "$\square$" ticket or a "$\blacksquare$" ticket.
  K_A $\square$ $\longrightarrow$ K_B `Carol_Jones Ted` $\square$ is an example.

Rules derived from authorization certificates can be further subdivided into two types:

- Rewrite rules derived from user authorization certificates:
  These rules can never have the issuer `SELF`.
  K_A $\square$ $\longrightarrow$ K_B `Carol_Jones Ted` $\square$ is an example.

[4] *described in Section 3.3.3*

- Rewrite rules derived from ACL authorization certificates:

  These rules always have the issuer SELF.

  SELF □ $\longrightarrow$ K_B Carol_Jones Ted ■ is an example.

6.2 Just Keys

Consider, first, the case in which the input set of user certificates consists of only authorization certificates, and the subjects of all of the authorization certificates and all of the ACL entries are keys. That is, consider the case in which there are no SPKI/SDSI names: all of the terms are keys. Let's denote the input public key (the prover's key) by K_*. The problem of producing the desired certificate chain reduces to doing a depth-first search[5] (DFS) through the authorization certificates, starting from "SELF □" and searching for "K_* $\boxed{\text{z}}$".

The procedure is simple. The strings in all of the rewrite rules each consist of either a key followed by a "□" ticket or a key followed by a "■" ticket.

Just keys algorithm:

1. Set up a directed graph with one vertex for each distinct string. There is an edge from the vertex representing string S_i to the vertex representing string S_j if there is a rewrite rule of the form $S_i \longrightarrow S_j$.

2. Use a depth-first search to determine if there is a path from "SELF □" to "K_* $\boxed{\text{z}}$". If there is a path, return the path; if there is not a path, terminate with failure.

Figure 6-1 gives an example in which the prover, K_A in this case, is authorized to access the object protected by the ACL via user certificates. In the figure, if the rule K_X □ $\longrightarrow$ K_Y □ were removed, say, K_A would not be authorized to access the protected object.

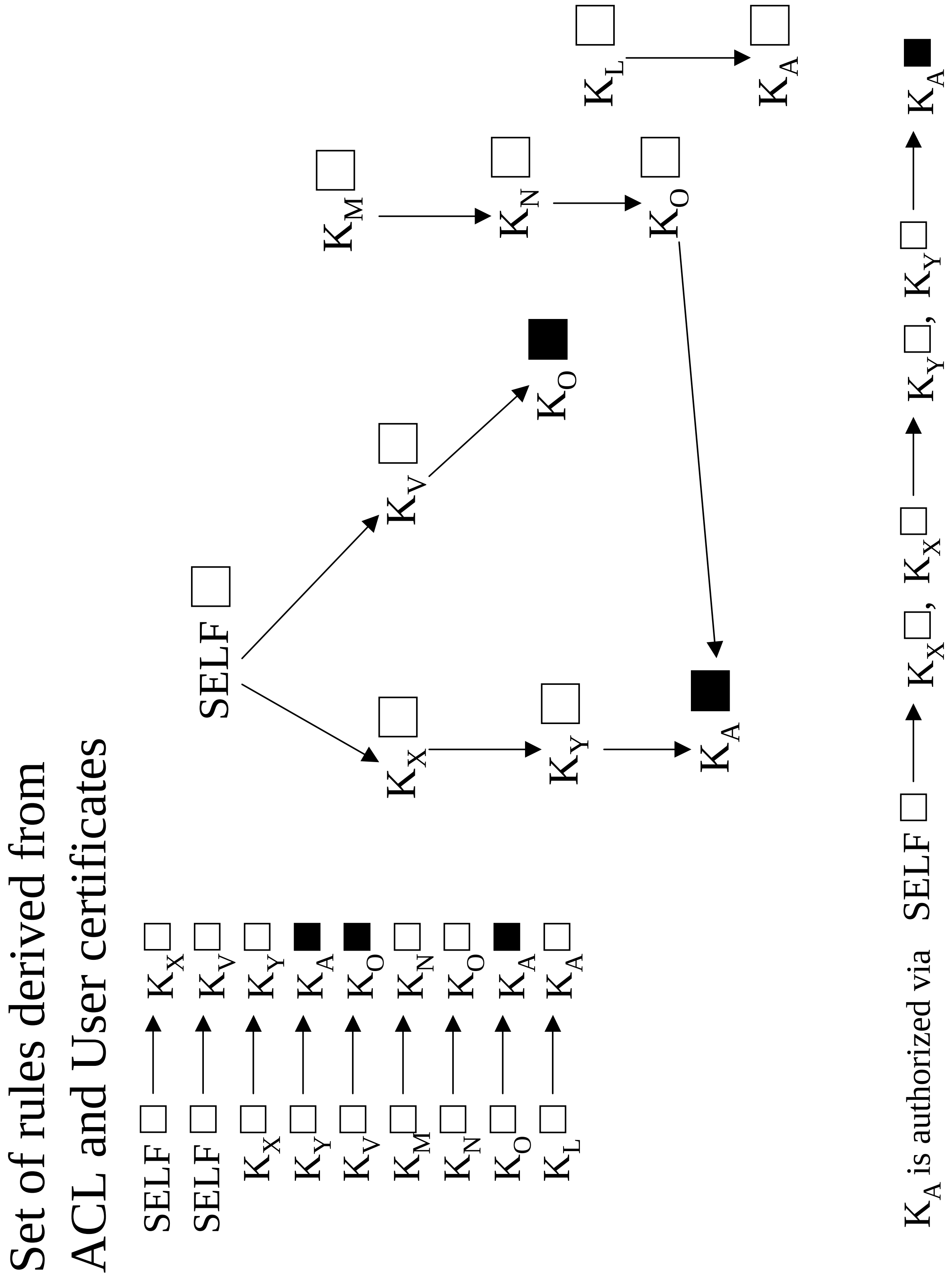


Figure 6-1: Just keys: user is authorized via user certificates

6.2.1 How live vs. dead tickets enforce delegation control

We can see how the distinction between a live ticket "$\square$" and a dead ticket "$\blacksquare$" represents and supports the SPKI/SDSI rules of delegation. An authorization rewrite rule having a dead ticket on the right cannot have another edge leading from it: the right-hand side of this rule must be a leaf vertex. A rule having a live ticket on the right may have another edge leading from it signifying authorization being propagated from the issuer in the first edge to the subject in the second edge. To illustrate the point, $\texttt{SELF} \ \square \longrightarrow K_X \ \square$ and $K_X \ \square \longrightarrow K_Y \ \square$ can be joined together in a graph to propagate authorization from $\texttt{SELF}$ to K_Y. $K_V \ \square \longrightarrow K_O \ \blacksquare$ and $K_O \ \square \longrightarrow K_A \ \blacksquare$ could not be joined together in the graph.

There is a second, very important, function of tickets. This function is described in Section 6.3.1.

6.3 Names and Keys

Now consider the general case, in which terms may be either keys or SPKI/SDSI names. We may now have authorization rules and name rules. Strings can be either a key, a name, a key followed by a "$\square$" ticket, a key followed by a "$\blacksquare$" ticket, a name followed by a "$\square$" ticket, or a name followed by "$\blacksquare$" ticket.

Definition: ***Composition of Rewrite Rules*** *Suppose C_1 is a rule of the form*

$$L_1 \longrightarrow R_1,$$

and suppose C_2 is a rule of the form

$$L_2 \longrightarrow R_2,$$

where L_2 is a prefix of R_1. That is, $R_1 = L_2X$ for some (possibly empty) string X. Then we define the composition of rewrite rules[3] $C_3 = C_1 \circ C_2$ as

$$\begin{aligned} C_3 &= C_1 \circ C_2 \\ &= L_1 \longrightarrow (R_1 \circ C_2) \\ &= L_1 \longrightarrow R_2X. \end{aligned}$$

We say that we have rewritten C_1 (using C_2) to obtain C_3. If L_2 is not a prefix of R_1, then $C_1 \circ C_2$ is undefined.

An an example, we can compose the following rules:

$$\begin{aligned} K_A\ \texttt{friends} &\longrightarrow K_A\ \texttt{Bob my-friends} \\ K_A\ \texttt{Bob} &\longrightarrow K_B \end{aligned}$$

to obtain the rewrite rule:

$$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{my-friends}.$$

That is, if K_A says that one definition of her name “`friends`” is the name “K_A `Bob my-friends`”, and K_A says that one possible definition of her name “`Bob`” is K_B, then one definition of K_A’s name “`friends`” is “K_B `my-friends`”.

Definition: ***Compatible*** *We say that two rules $C_1 = (L_1 \longrightarrow R_1)$ and $C_2 = (L_2 \longrightarrow R_2)$ are compatible if their composition $C_3 = C_1 \circ C_2$ is defined, that is, if L_2 is a prefix of R_1.(More precisely, if $C_1 \circ C_2$ is defined, we say that C_1 is left-compatible with C_2, and that C_2 is right-compatible with C_1.)*

Note that the definition of compatibility really applies to the *ordered pair* (C_1, C_2) since $C_1 \circ C_2$ may be defined (so that C_1 and C_2 are compatible), but $C_2 \circ C_1$ may be undefined (so that C_2 and C_1 are not compatible). Thus, there is the need for the more refined notions of left and right-compatibility.

It should now be clear why we have been describing the rules as "rewrite rules", and defined a "string" in Section 6.1.1. The rules can be used to rewrite strings to form new rules.

Thus, in the general case, we cannot simply form a graph and do a depth-first search through it[5]. Rules can be composed to derive new rules, with new names not previously appearing in any certificate. The new rules can be further composed to derive more rules, with more new names. There exist sets of rules in which composition can take place ad infinitum, producing new rules. Consider, for instance, the set consisting of the single rule $K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}$. This rule can be composed with itself to form new rules indefinitely:

1. $K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A} \circ K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A} = K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}$

2. $K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A} \circ K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A} = K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}$

3. $K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A} \circ K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A} = K_1\ \mathtt{A} \longrightarrow K_1\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}\ \mathtt{A}$

$\vdots$

It was not obvious that an efficient SPKI/SDSI certificate chain discovery algorithm could be developed. We have developed such an algorithm, and a tight bound for its running time. An earlier version of this algorithm appears in Jean-Emile Elien's Master's thesis[7].

6.3.1 Why authorization rewrite rules have tickets

We can see why authorization rules are represented as rewrite rules with tickets. The presence of the tickets prevents an authorization rule from being inappropriately used in a composition as a name rule. For example, it is *not* correct, according to

[5] The Certificate Chain Discovery paper considers the case in which all of the certificates are user name certificates, and all of the subjects of those certificates are either local names or keys; similar to the case described in Section 6.2, a graph can also be created and a depth-first search performed to find the values (described in Section 6.4.1) of the names.

the SPKI/SDSI composition rules, to compose the following name and authorization rules:

$$K_A\ \mathtt{C} \longrightarrow K_B\ \mathtt{C}$$
$$K_B\ \square \longrightarrow K_B\ \mathtt{D}\ \square$$

to obtain

$$K_A\ \mathtt{C} \longrightarrow K_B\ \mathtt{D}\ \mathtt{C}.$$

Were the tickets not used, this might erroneously be considered a legal composition. With tickets, the two rules are not compatible. This restriction is consistent with the viewpoint that the purpose of an authorization certificate is to grant permission, and not to rewrite names. Only name rules can be used to rewrite names. This is a crucial point. Tickets maintain the *def/auth split*[7] in SPKI/SDSI i.e. a single certificate cannot both define a name and grant an authorization: each certificate is either strictly a name certificate or an authorization certificate.

We observe the following properties of the composition $C_3 = C_1 \circ C_2$:

1. The type of C_3, as an authorization or name rule, is the same as the type of C_1. (Rewriting cannot create or destroy tickets.)

2. If C_2 is an authorization rule, then $L_2 = R_1$.
   To illustrate, if C_2 is $K_B\ \square \longrightarrow K_C\ \square$, and C_3 is $K_A\ \square \longrightarrow K_C\ \square$, then C_1 must be $K_A\ \square \longrightarrow K_B\ \square$.

3. If C_1 is a name rule, then so is C_2. (Equivalently, if C_2 is an authorization rule, then so is C_1.)

4. If R_1 contains a dead ticket, then C_2 must be a name cert.

To give more examples:

- the rules $K_B \; \square \longrightarrow K_B \; \texttt{C} \; \square$ and $K_B \; \texttt{C} \longrightarrow K_E$ can be composed to form $K_B \; \square \longrightarrow K_E \; \square$.

- the rules $K_B \; \square \longrightarrow K_E \; \square$ and $K_E \; \square \longrightarrow K_H \; \texttt{I} \; \texttt{J} \; \blacksquare$ can be composed to form $K_B \; \square \longrightarrow K_H \; \texttt{I} \; \texttt{J} \; \blacksquare$.

- the rules $K_B \; \square \longrightarrow K_B \; \texttt{C} \; \blacksquare$ and $K_B \; \texttt{C} \longrightarrow K_E$ can be composed to form $K_B \; \square \longrightarrow K_E \; \blacksquare$.

- the rules $K_B \; \square \longrightarrow K_E \; \blacksquare$ and $K_E \; \square \longrightarrow K_H \; \blacksquare$ cannot be composed.

- the rules $K_B \; \texttt{D} \longrightarrow K_B \; \texttt{C}$ and $K_B \; \texttt{C} \longrightarrow K_E$ can be composed to form $K_B \; \texttt{D} \longrightarrow K_E$.

6.4 Basic Algorithm

Recall, the certificate chain discovery algorithm takes as input an ACL, a tag, a public key, a set of user certificates, and a timestamp. If it exists, the algorithm returns a certificate chain, consisting of user certificates, which provides proof that the public key (principal) is authorized to perform the operation(s) specified in the tag on the object protected by the ACL, at the time specified in the timestamp. In this section, the initial set of input ACL "certificates" and user certificates will be denoted by $\mathcal{C}$; again, the input public key (the prover's key) will be denoted by K_*.

The basic certificate chain discovery algorithm is as follows:

Basic certificate chain discovery algorithm:

1. Remove from $\mathcal{C}$, all the useless certificates. This step is described in Section 6.1. Convert the remaining certificates into rewrite rules.

2. Compute the *name-reduction closure* of $\mathcal{C}$. This step is described in Section 6.4.1. Let's denote the result of computing the name-reduction closure of $\mathcal{C}$ as $\mathcal{C}^{\#}$.

3. From $\mathcal{C}^{\#}$, extract all rules of the form:

$$K_i \,\square \longrightarrow K_j \,\boxed{\text{z}}$$

(where K_i may be "`SELF`"). Let's denote this set of rules as $\mathcal{C}'$.

4. We are now precisely in the case described in Section 6.2 "Just Keys". Now, run the algorithm described in that section on $\mathcal{C}'$. Each of the strings in $\mathcal{C}'$ consists of a key followed by a ticket. To recap, the steps of that algorithm are:

   (a) Set up a graph with one vertex for each distinct string. There is an edge from the vertex representing string S_i to the vertex representing string S_j if there is a rewrite rule of the form $S_i \longrightarrow S_j$.

   (b) Use a depth-first search[5] to determine if there is a path from "`SELF` □" to "$K_* \,\boxed{\text{z}}$". If there is no path, terminate with failure.

5. From the information computed in the previous steps, reconstruct and output the desired certificate chain, consisting only of certificates from the input set of user certificates.

This algorithm is guaranteed to find a certificate chain if one exists. If a certificate chain exists, the chain returned can be used to deterministically prove, to any entity, that K_* is authorized to perform the operation(s) specified in the input tag on the object protected by the ACL, at the time specified in the timestamp. Referring to the protocol in Section 6.1, the prover sends this certificate chain as its "proof of authorization".

The name-reduction closure in step 2 is described in the following section.

6.4.1 Name-Reduction Closure

Valuation function

We shall be concerned with the *value* of various terms. (Recall that a term is a key or a name.) In SPKI/SDSI, *a value is a set of public keys* (possibly the empty set).

The value of a term T is defined relative to a set $\mathcal{C}$ of certificates.

Notation: ***value of a term*** *We let $\mathcal{V}_{\mathcal{C}}(T)$ denote the value of a term T with respect to a set $\mathcal{C}$ of certificates. When $\mathcal{C}$ may be understood from context, we may use the simpler notation $\mathcal{V}(T)$. The value of a term is a set of public keys, possibly empty.*

Value of a key A public key is the simplest kind of a SPKI/SDSI term - it is constant expression evaluating to itself (as a singleton set). $\mathcal{V}_{\mathcal{C}}(K) = \{K\}$ for any public key K and any set $\mathcal{C}$ of certificates. For example, $\mathcal{V}_{\mathcal{C}_1}(K_A) = \{K_A\}$ and $\mathcal{V}_{\mathcal{C}_2}(K_B) = \{K_B\}$, irrespective of $\mathcal{C}_1$ and $\mathcal{C}_2$.

Value of a name A name has a value that is a set of public keys; this value may be the empty set, a set containing a single key, or a set containing multiple keys. This value is determined by one or more name certificates (name rewrite rules); recall that authorization certificates cannot be used to rewrite names, because they have tickets (the def/auth split!).

Intuitively, a name certificate "K `A` $\longrightarrow$ `S`", defining local name "K `A`" in terms of subject `S`, should be understood as a signed statement by the issuer asserting that

$$\mathcal{V}(K\ \mathtt{A}) \supseteq \mathcal{V}(\mathtt{S});$$

that is, every key in the value $\mathcal{V}(\mathtt{S})$ of subject `S` is also a key in the value $\mathcal{V}(K\ \mathtt{A})$ of local name "K `A`". SPKI/SDSI does not have "negative" name certificates, i.e. you cannot issue a certificate to remove some key from a group[6]. Each additional name certificate for "K `A`" can only add zero or more new principals to $\mathcal{V}(K\ \mathtt{A})$. Thus, the above equation says $\mathcal{V}(K\ \mathtt{A}) \supseteq \mathcal{V}(\mathtt{S})$ and not $\mathcal{V}(K\ \mathtt{A}) = \mathcal{V}(\mathtt{S})$.

A local name, such as "K `Alice`", need not have the same meaning as the local name "K' `Alice`" when $K \neq K'$; the owner of key K may define "K `Alice`"

[6] SPKI/SDSI also does not have "negative" authorization certificates: a permission granted by an authorization certificate is good until the certificate expires or becomes invalid.

however he wishes, while the owner of key K' may similarly but independently define "K' `Alice`" in an arbitrary manner.

The preceding definition gives the value of local names. The value of a name with 2 or more identifiers, "$K\ A_1\ A_2\ ...\ A_n$", can be defined recursively as:

$$\mathcal{V}(K\ A_1\ A_2\ ...\ A_n)\ =\ \bigcup_{K' \in \mathcal{V}(KA_1)} \mathcal{V}(K'\ A_2\ ...\ A_n).$$

Thus, any name in SPKI/SDSI represents a set of public keys. Figure 6-2 gives an illustration; it presents a set $\mathcal{C}$ of name rules and gives $\mathcal{V}_{\mathcal{C}}(T)$ for various terms T.

Closure of a set of rules

The notion of the closure of a set of rules is fundamental; the closure contains all rules that can be derived by composition from the given set of rules.

Definition: ***Closure*** *If $\mathcal{C}$ is a set of rules, we define the set $\mathcal{C}^+$, called the (transitive) closure of $\mathcal{C}$, as the smallest set of rules that includes $\mathcal{C}$ as a subset and that is closed under composition of rules.*

Informally, the closure $\mathcal{C}^+$ contains all rules that can be inferred from $\mathcal{C}$ using any finite number of compositions.

It seems, therefore, that in step 2 in the Basic Algorithm in Section 6.4 on page 128, we just need to calculate the closure, $\mathcal{C}^+$, of $\mathcal{C}$, and extract those rules in which the terms on the left and right sides are just keys. However, the closure $\mathcal{C}^+$ need not be a finite set, even if $\mathcal{C}$ is finite. For example, as we have seen, if $\mathcal{C}$ were to contain the rule K_1 `A` $\longrightarrow K_1$ `A A`, $\mathcal{C}^+$ will be an infinite set. Each rule in $\mathcal{C}^+$ has a finite-length derivation from the rules in $\mathcal{C}$, but the set $\mathcal{C}^+$ may or may not be finite.

Thus, we define a finite subset of the closure, called the "name-reduction closure", a finite subset of $\mathcal{C}^+$ that is easy and efficient to compute, and just as useful as the closure in deriving certificate chains.

Name certificates issued by Alice:

$$K_A\ \texttt{Bob} \longrightarrow K_B \tag{6.1}$$

$$K_A\ \texttt{Carol} \longrightarrow K_B\ \texttt{Carol\_Jones} \tag{6.2}$$

$$K_A\ \texttt{Ted} \longrightarrow K_B\ \texttt{Carol\_Jones}\ \texttt{Ted} \tag{6.3}$$

$$K_A\ \texttt{friends} \longrightarrow K_A\ \texttt{Bob} \tag{6.4}$$

$$K_A\ \texttt{friends} \longrightarrow K_A\ \texttt{Carol} \tag{6.5}$$

$$K_A\ \texttt{friends} \longrightarrow K_A\ \texttt{Ted} \tag{6.6}$$

$$K_A\ \texttt{friends} \longrightarrow K_A\ \texttt{Bob}\ \texttt{my-friends} \tag{6.7}$$

Name certificates issued by Bob:

$$K_B\ \texttt{Alice} \longrightarrow K_A \tag{6.8}$$

$$K_B\ \texttt{Carol\_Jones} \longrightarrow K_C \tag{6.9}$$

$$K_B\ \texttt{Frank} \longrightarrow K_F \tag{6.10}$$

$$K_B\ \texttt{my-friends} \longrightarrow K_B\ \texttt{Alice} \tag{6.11}$$

$$K_B\ \texttt{my-friends} \longrightarrow K_B\ \texttt{Frank} \tag{6.12}$$

Name certificates issued by Carol:

$$K_C\ \texttt{Ted} \longrightarrow K_T \tag{6.13}$$

If follows that the following local names have the values:

$$\begin{aligned}
\mathcal{V}(K_A\ \texttt{Bob}) &= \{K_B\} \\
\mathcal{V}(K_A\ \texttt{Carol}) &= \{K_C\} \\
\mathcal{V}(K_A\ \texttt{Ted}) &= \{K_T\} \\
\mathcal{V}(K_A\ \texttt{friends}) &= \{K_B,\ K_C,\ K_T,\ K_A,\ K_F\} \\
\mathcal{V}(K_B\ \texttt{Alice}) &= \{K_A\} \\
\mathcal{V}(K_B\ \texttt{Carol\_Jones}) &= \{K_C\} \\
\mathcal{V}(K_B\ \texttt{Frank}) &= \{K_F\} \\
\mathcal{V}(K_B\ \texttt{my-friends}) &= \{K_A,\ K_F\} \\
\mathcal{V}(K_C\ \texttt{Ted}) &= \{K_T\}
\end{aligned}$$

Figure 6-2: Example illustrating the values of some local names

Name-reduction closure of a set of rules

Definition: ***Name-reduction closure*** *If $\mathcal{C}$ is a set of rules, we define the set $\mathcal{C}^{\#}$, called the name-reduction closure of $\mathcal{C}$, as the smallest set of rules that includes $\mathcal{C}$ as a subset and that is closed under composition of "reducing" rules. That is, if $\mathcal{C}^{\#}$ contains a rule $\mathcal{C}_1$ and it also contains a (right-)compatible* *<u>reducing</u> rule $\mathcal{C}_2$, then $\mathcal{C}^{\#}$ must also contain $C_1 \circ C_2$.*

Definition: ***Reducing rule*** *We say that a rule $C = (L \longrightarrow R)$ is reducing if $|L| > |R|$, where $|X|$ denotes the length of sequence X. A reducing rule can only be a name rule of the form: K $\mathtt{A} \longrightarrow K'$ (K' may or may not be the same as K).*

If $C_1 = (L_1 \longrightarrow R_1)$ is an arbitrary rule, and $C_2 = (L_2 \longrightarrow R_2)$ is a (right-)compatible reducing rule, then $C_3 = C_1 \circ C_2 = (L_1 \longrightarrow R_3)$ satisfies $|R_1| > |R_3|$. That is, rewriting C_1 with a reducing rule C_2 gives a new rule with a strictly shorter right-hand side. For example,
$K_A\ \mathtt{D} \longrightarrow K_B\ \mathtt{E\ F\ G} \circ K_B\ \mathtt{E} \longrightarrow K_C = K_A\ \mathtt{D} \longrightarrow K_C\ \mathtt{F\ G}$.

Thus, to compute the name-reduction closure, we only perform rewritings that cause a reduction in the length of the right-hand side, until no more such rewritings can be done. This is clearly a finite process. More precisely, our algorithm for computing the name-reduction closure is the following:

Name-reduction closure algorithm:

1. Initialize $\mathcal{C}_{temp}$ to be the input set $\mathcal{C}$ of rules.

2. As long as $\mathcal{C}_{temp}$ contains two compatible rules C_1 and C_2 such that C_2 is a reducing rule and $C_1 \circ C_2$ is not yet in $\mathcal{C}_{temp}$, add $C_1 \circ C_2$ to $\mathcal{C}_{temp}$.

3. Return $\mathcal{C}_{temp}$ as the computed value of $\mathcal{C}^{\#}$.

Intuitively, computing the name-reduction closure $\mathcal{C}^{\#}$ of a set of rules $\mathcal{C}$ only performs compositions that are useful in computing the value $\mathcal{V}_{\mathcal{C}}(S)$ of each subject S in $\mathcal{C}$. The importance of the name-reduction closure of a set of rules is given by the following theorems, which show that the name-reduction closure explicitly computes the values of terms appearing on the right-hand side of the input rules.

Theorem 1 *Suppose that $\mathcal{C}$ is a set of rules.*
If

$$C \;=\; (L \;\longrightarrow\; R)$$

is a name rule in $\mathcal{C}$. Then, for each key $K \in \mathcal{V}_{\mathcal{C}}(R)$,

$$L \;\longrightarrow\; K$$

is a rule in $\mathcal{C}^{\#}$.
Similarly, if

$$C \;=\; (K' \,\Box \;\longrightarrow\; R \,\Box)$$

is an authorization rule in $\mathcal{C}$. Then, for each key $K \in \mathcal{V}_{\mathcal{C}}(R)$,

$$K' \,\Box \;\longrightarrow\; K \,\Box$$

is a rule in $\mathcal{C}^{\#}$.
Similarly, if

$$C \;=\; (K' \,\Box \;\longrightarrow\; R \,\blacksquare)$$

is an authorization rule in $\mathcal{C}$. Then, for each key $K \in \mathcal{V}_{\mathcal{C}}(R)$,

$$K' \,\Box \;\longrightarrow\; K \,\blacksquare$$

is a rule in $\mathcal{C}^{\#}$.

Theorem 2 *Suppose that $\mathcal{C}$ is a set of rules.*
If

$$C \;=\; (L \;\longrightarrow\; K)$$

is a name rule in $\mathcal{C}^{\#}$, then there exists a rule

$$C' \;=\; (L \;\longrightarrow\; R)$$

in $\mathcal{C}$ such that $K \in \mathcal{V}_{\mathcal{C}}(R)$.
Similarly, if

$$C \;=\; (K' \,\square \;\longrightarrow\; K \,\square)$$

is an authorization rule in $\mathcal{C}^{\#}$, then there exists a rule

$$C' \;=\; (K' \,\square \;\longrightarrow\; R \,\square)$$

in $\mathcal{C}$ such that $K \in \mathcal{V}_{\mathcal{C}}(R)$.
Similarly, if

$$C \;=\; (K' \,\square \;\longrightarrow\; K \,\blacksquare)$$

is an authorization rule in $\mathcal{C}^{\#}$, then there exists a rule

$$C' \;=\; (K' \,\square \;\longrightarrow\; R \,\blacksquare)$$

in $\mathcal{C}$ such that $K \in \mathcal{V}_{\mathcal{C}}(R)$.

These two theorems are the ‘heart-and-soul’ of the certificate chain discovery algorithm. Their proofs are in the “Certificate Chain Discovery in SPKI/SDSI”[3] paper. (This thesis just focuses on giving the intuition behind the algorithm, and is not mathematically rigorous enough to detail the proofs.) Figure 6-3 shows the name-reduction closure, and closure, of the certificates from Figure 6-2 on page 132.

The name-reduction closure $\mathcal{C}^{\#}$ of the certificates in Figure 6-2 includes those certificates/rules as well as the following (the derivation of each rule is given on the right; for example, rule (6.14) is obtained by composing rules (6.2) and (6.9)):

(6.14)	$K_A\ \texttt{Carol} \longrightarrow K_C$	(6.2) $\circ$ (6.9)
(6.15)	$K_A\ \texttt{Ted} \longrightarrow K_C\ \texttt{Ted}$	(6.3) $\circ$ (6.9)
(6.16)	$K_A\ \texttt{Ted} \longrightarrow K_T$	(6.15) $\circ$ (6.13)
(6.17)	$K_A\ \texttt{friends} \longrightarrow K_B$	(6.4) $\circ$ (6.1)
(6.18)	$K_A\ \texttt{friends} \longrightarrow K_C$	(6.5) $\circ$ (6.14)
(6.19)	$K_A\ \texttt{friends} \longrightarrow K_T$	(6.6) $\circ$ (6.16)
(6.20)	$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{my-friends}$	(6.7) $\circ$ (6.1)
(6.21)	$K_B\ \texttt{my-friends} \longrightarrow K_A$	(6.11) $\circ$ (6.8)
(6.22)	$K_B\ \texttt{my-friends} \longrightarrow K_F$	(6.12) $\circ$ (6.10)
(6.23)	$K_A\ \texttt{friends} \longrightarrow K_A$	(6.20) $\circ$ (6.21)
(6.24)	$K_A\ \texttt{friends} \longrightarrow K_F$	(6.20) $\circ$ (6.22)

All of the preceding rules are also in the closure $\mathcal{C}^{+}$. The following rules are in $\mathcal{C}^{+}$ but not in the name-reduction closure $\mathcal{C}^{\#}$, since (6.11) and (6.12) are not reducing rules:

(6.25)	$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{Alice}$	(6.20) $\circ$ (6.11)
(6.26)	$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{Frank}$	(6.20) $\circ$ (6.12)

Figure 6-3: The name-reduction closure, and closure, of the certificates in Figure 6-2

6.5 Running Time

Determining the running time of the certificate chain discovery algorithm reduces to determining the running time of calculating the name-reduction closure.

Theorem 3: Running time of calculating the name-reduction closure *The running time of name reduction closure on an input set of n certificates, where l is the length of the longest subject in any input certificate, is $O(n^3 l)$.*

Again, the proof of this theorem is detailed in the "Certificate Chain Discovery in SPKI/SDSI"[3] paper. There exists an actual pathological input set of certificates in which the name-reduction closure will take $O(n^3 l)$ time to run.

We believe that the certificate chain discovery algorithm is very practical, and that it will be exceptionally effective in practice. In practice, we feel that, as pathological cases will be uncommon, it will often be the case that $|\mathcal{C}^{\#}|$ is proportional to $|\mathcal{C}|$, so that the running time of our algorithm will be linear.

6.6 Full Example

Thus, the certificate chain discovery algorithm first removes useless authorization certificates; converts the remaining name and authorization certificates into rewrite rules; computes the name-reduction closure on these rules so that, with respect to the rules, the names in the subjects of authorization rules are dereferenced to keys; extracts all rules of the form: $K_i \; \square \longrightarrow K_j \; \boxed{\text{z}}$ (where K_i may be "SELF"); forms a graph of these rules; does a depth-first search to determine if there is a path from "SELF □" to "$K_* \; \boxed{\text{z}}$", where K_* is the input public key; if there is a path reconstructs and outputs the desired certificate chain, consisting only of certificates from the input set of user certificates, and otherwise, terminates with failure. Essentially, the proof of authorization that the algorithm returns consists of a chain of certificates that allows one to derive "$K_* \; \boxed{\text{z}}$" from "SELF □"; i.e. *the proof of authorization consists of the derivation of the rule SELF* $\square \longrightarrow K_* \; \boxed{z}$.

Consider the following example:

ACL entries:

(6.27) $(\texttt{SELF},\ K_0\ \texttt{engineering},\ T_1,\ \texttt{true},\ (\texttt{07/28/01},\ \texttt{07/30/01}))$

(6.28) $(\texttt{SELF},\ K_0\ \texttt{finance},\ T_1,\ \texttt{true},\ (\texttt{07/28/01},\ \texttt{07/30/01}))$

(6.29) $(\texttt{SELF},\ K_0\ \texttt{human\_resources},\ T_1,\ \texttt{false},\ (\texttt{10/09/01},\ \texttt{10/11/01}))$

User certificates:

(6.30) $K_0\ \texttt{finance} \longrightarrow K_1\ \texttt{accounting}$

(6.31) $K_1\ \texttt{accounting} \longrightarrow K_1\ \texttt{Bob}$

(6.32) $K_1\ \texttt{Bob} \longrightarrow K_2$

(6.33) $(K_2,\ K_3\ \texttt{Alice},\ T_1,\ \texttt{false},\ (\texttt{07/28/01},\ \texttt{07/30/01}))$

(6.34) $K_3\ \texttt{Alice} \longrightarrow K_A$

(6.35) $K_5\ \texttt{Alice\_Brown} \longrightarrow K_A$

(6.36) $(K_6,\ K_3\ \texttt{Alice},\ T_2,\ \texttt{false},\ (\texttt{07/28/01},\ \texttt{07/30/01}))$

The tags T_1 and T_2 are completely disjoint.

At time 07/29/01, Alice (K_A), runs the certificate chain discovery algorithm with tag T_1 (assume that all of the name certificates are valid on 07/29/01).

1. *Remove all the useless certificates and convert the remaining certificates into rewrite rules:*

At the conclusion of this step, we have:

$$\mathtt{SELF}\ \square \longrightarrow K_0\ \mathtt{engineering}\ \square \qquad \text{(from certificate (6.27))} \tag{6.37}$$

$$\mathtt{SELF}\ \square \longrightarrow K_0\ \mathtt{finance}\ \square \qquad \text{(from certificate (6.28))} \tag{6.38}$$

$$K_0\ \mathtt{finance} \longrightarrow K_1\ \mathtt{accounting} \qquad \text{(from certificate (6.30))} \tag{6.39}$$

$$K_1\ \mathtt{accounting} \longrightarrow K_1\ \mathtt{Bob} \qquad \text{(from certificate (6.31))} \tag{6.40}$$

$$K_1\ \mathtt{Bob} \longrightarrow K_2 \qquad \text{(from certificate (6.32))} \tag{6.41}$$

$$K_2\ \square \longrightarrow K_3\ \mathtt{Alice}\ \blacksquare \qquad \text{(from certificate (6.33))} \tag{6.42}$$

$$K_3\ \mathtt{Alice} \longrightarrow K_A \qquad \text{(from certificate (6.34))} \tag{6.43}$$

$$K_5\ \mathtt{Alice\_Brown} \longrightarrow K_A \qquad \text{(from certificate (6.35))} \tag{6.44}$$

2. *Compute the name-reduction closure.*

   At the conclusion of this step, we have:

$$\mathtt{SELF}\ \square \longrightarrow K_0\ \mathtt{engineering}\ \square \qquad \text{(rule (6.37))} \tag{6.45}$$

$$\mathtt{SELF}\ \square \longrightarrow K_0\ \mathtt{finance}\ \square \qquad \text{(rule (6.38))} \tag{6.46}$$

$$K_0\ \mathtt{finance} \longrightarrow K_1\ \mathtt{accounting} \qquad \text{(rule (6.39))} \tag{6.47}$$

$$K_1\ \mathtt{accounting} \longrightarrow K_1\ \mathtt{Bob} \qquad \text{(rule (6.40))} \tag{6.48}$$

$$K_1\ \mathtt{Bob} \longrightarrow K_2 \qquad \text{(rule (6.41))} \tag{6.49}$$

$$K_2\ \square \longrightarrow K_3\ \mathtt{Alice}\ \blacksquare \qquad \text{(rule (6.42))} \tag{6.50}$$

$$K_3\ \mathtt{Alice} \longrightarrow K_A \qquad \text{(rule (6.43))} \tag{6.51}$$

$$K_5\ \mathtt{Alice\_Brown} \longrightarrow K_A \qquad \text{(rule (6.44))} \tag{6.52}$$

$$K_1\ \mathtt{accounting} \longrightarrow K_2 \qquad (6.48) \circ (6.49) \tag{6.53}$$

$$K_2\ \square \longrightarrow K_A\ \blacksquare \qquad (6.50) \circ (6.51) \tag{6.54}$$

$$K_0\ \mathtt{finance} \longrightarrow K_2 \qquad (6.47) \circ (6.53) \tag{6.55}$$

$$\mathtt{SELF}\ \square \longrightarrow K_2\ \square \qquad (6.46) \circ (6.55) \tag{6.56}$$

3. *Extract all rules of the form:*

$$K_i \, \square \;\longrightarrow\; K_j \, \boxed{z}$$

(where K_i may be "`SELF`").

At the conclusion of this step, we have:

$$K_2 \, \square \;\longrightarrow\; K_A \, \blacksquare \qquad \text{(rule 6.54)} \tag{6.57}$$

$$\texttt{SELF} \, \square \;\longrightarrow\; K_2 \, \square \qquad \text{(rule 6.56)} \tag{6.58}$$

4. *Set up a graph and do a depth-first search to determine if there is a path from "`SELF` □" to "K_A $\boxed{z}$". If there is no path, terminate with failure.*

This graph simply looks like:

$$\texttt{SELF} \, \square \longrightarrow K_2 \, \square \longrightarrow K_A \, \blacksquare$$

The DFS returns the path

`SELF` □ to K_2 □ to K_A ■

showing that a certificate chain exists.

5. *From the information computed in the previous steps, reconstruct and output the desired certificate chain, consisting only of certificates from the input set of user certificates.*

The resulting certificate chain is, represented as an ordered list:

$$(K_0\ \texttt{finance} \ \longrightarrow\ K_1\ \texttt{accounting},$$

$$K_1\ \texttt{accounting} \ \longrightarrow\ K_1\ \texttt{Bob},$$

$$K_1\ \texttt{Bob} \ \longrightarrow\ K_2,$$

$$(K_2,\ K_3\ \texttt{Alice},\ T_1,\ \texttt{false},\ (\texttt{07/28/01},\ \texttt{07/30/01})),$$

$$K_3\ \texttt{Alice} \ \longrightarrow\ K_A)$$

If Alice (K_A), as a prover, were to sign a request that was a subset of T_1 and send this request accompanied with this certificate chain to a verifier at time `07/29/01`, the verifier would be able to derive the rule $\texttt{SELF}\ \square \ \longrightarrow\ K_A\ \blacksquare$. If the verifier was a server[7], and the other parts of the server's verification process were successful, Alice's request would be honored.

Production of Certificate Chains The format of the output certificate chain in this example in the last step is described in the Certificate Chain Discovery paper[3] as the *linear* format. As the paper notes, the linear format may produce an output which is exponential in the size of the input certificate set. The paper also describes the *compressed* format, which is just as easy to create, but produces an output which is polynomial in the size of the input.

6.7 Threshold Subjects

Threshold subjects are described in Section 3.3.3. The scenario for certificate chain discovery is, essentially, the same as before: a set of parties $Alice_1$ (K_{Alice_1}), $Alice_2$ (K_{Alice_2}), ..., $Alice_n$ (K_{Alice_n}) attempt to determine if they are authorized if they collectively sign an access request, based on a set of (user and ACL) certificates that may contain authorization certificates with threshold subjects. To derive certificate chains, the

[7] described in Section 5.1

certificate chain discovery algorithm could, essentially, make n copies of the certificate/rule with the threshold subject, use a separate copy to handle each different subject that is specified in the threshold, and recursively call itself to derive an authorization chain from each copy to any one of keys K_{Alice_1}, K_{Alice_2}, ..., K_{Alice_n}. If there are chains from k copies, the algorithm continues until it has derived a chain originating from `SELF`; otherwise, the algorithm terminates with failure.

Definition: ***Threshold subject*** *A threshold subject[3] is an expression of the form*

$$\Theta_k(S_1,\ S_2,\ ...,\ S_n)\ \boxed{z}$$

where $1\ \leq\ k\ \leq\ n$ *and where each* S_i *is a term or another threshold subject.*

As an example, consider the following authorization rule derived from a certificate with a threshold subject:

$$\texttt{SELF}\ \Box\ \longrightarrow\ \Theta_2(K_0\ \mathsf{mit\ faculty},\ K_0\ \mathsf{intel\ researcher},\ K_0\ \mathsf{Alice})\ \Box \qquad (6.59)$$

This rule requires that keys representing at least two of the three names sign an access request; equivalently with two of the three groups (MIT faculty, Intel researcher, or Alice) represented. (If Alice is an MIT faculty member, then her signature alone is good enough; otherwise two keys must be used to sign the request.)

The basic certificate chain discovery algorithm described in Section 6.4 is extended to handle threshold subjects. As noted, there is now not just a single signer K_* on the request, but a set $\mathcal{K}_*$ of signers; we want to determine if this set of signers is authorized.

The extended algorithm is as follows:

Basic certificate chain discovery algorithm extended to handle threshold

subjects:

1. Apply Step 1 of the original certificate chain discovery algorithm, i.e. remove useless certificates and convert the remaining ones into rewrite rules.

2. Make n copies of the rule with the threshold subject, use a separate copy to handle each different subject that is specified in the threshold. Basically,

   (a) introduce new dummy placeholder keys at each position in the threshold subject. For example, the rule derived from the authorization certificate (6.59) is rewritten as

   $$\texttt{SELF}\ \square \longrightarrow \Theta_2(K_{dummy_1},\ K_{dummy_2},\ K_{dummy_3})\ \square \tag{6.60}$$

   where K_{dummy_1}, K_{dummy_2}, K_{dummy_3} represent new public keys that do not appear elsewhere in the set of rules.

   (b) add additional authorization rules to preserve the semantics of the original (now rewritten) authorization rule. Continuing with the same example, we would add the rules:

   $$\begin{aligned} K_{dummy_1}\ \square &\longrightarrow K_0\ \textsf{mit faculty}\ \square \\ K_{dummy_2}\ \square &\longrightarrow K_0\ \textsf{intel researcher}\ \square \\ K_{dummy_3}\ \square &\longrightarrow K_0\ \textsf{Alice}\ \square \end{aligned}$$

   (If the original authorization rule had a dead ticket instead of a live one on the right-hand side, then these rules would also have dead tickets on their right-hand sides.)

3. Temporarily set aside the rewritten authorization rules of the form (6.60), so that we have a set of rules containing no threshold subjects whatsoever. We now

apply Step 2 of the original certificate chain discovery algorithm, the name reduction closure. Again, let's denote the result of computing the name reduction closure as $\mathcal{C}^{\#}$.

4. Extract all rules of the form:

$$K_i \, \square \;\longrightarrow\; K_j \, \boxed{\text{z}}$$

(where K_i may be "`SELF`"). This step is the same as the Step 3 of the original algorithm. Again, let's denote this set of rules as $\mathcal{C}'$.

5. Add the threshold rules set aside in Step 3 of this algorithm to $\mathcal{C}'$. We now have just a set of authorization rules, each of which has as a subject either a key or a threshold subject on a list of keys. This is analogous to the case in Step 4 of the original algorithm. At this stage, using the set $\mathcal{C}'$,

   (a) Remove any authorization rule whose right-hand side is "K_j ■" where "K_j" is not a member of the set $\mathcal{K}_*$ of keys that may participate in this access request.

   (b) Label each key in $\mathcal{K}_*$ as "marked"; label all others as "unmarked".

   (c) Until no more progress can be made, iterate the following:

   - If the key K_j is marked, and there is an authorization rule $K_i \, \square \;\longrightarrow\; K_j \, \boxed{\text{z}}$, then mark K_i.
   - If there is an authorization rule of the form

$$K_i \, \square \;\longrightarrow\; \Theta_k(K_{l_1},\; K_{l_2},\; ...,\; K_{l_n}) \, \boxed{\text{z}}$$

   where at least k of the keys K_{l_1}, K_{l_2}, ..., K_{l_n} are marked, then mark K_i.

If `SELF` is now marked, there exists a certificate chain. Otherwise, terminate with failure.

6. From the information computed in the previous steps, reconstruct and output the desired certificate chain, consisting only of certificates from the input set of user certificates. This step is analogous to step 5 of the original algorithm.

6.8 Certificate Chain Discovery Conclusion

We have developed an efficient algorithm for computing certificate chains for SPKI/SDSI. Thus, SPKI/SDSI has an efficient procedure for answering the fundamental question, "Is A authorized to do X?" While SPKI/SDSI is very expressive, its expressiveness does not come at the price of intractability; sets of SPKI/SDSI certificates are easy to work with.

Chapter 7

Conclusion

SPKI/SDSI's most interesting features include the ability to define groups, the ability to delegate authorizations, and the facilitation of the development of scalable, secure, distributed computing systems. This thesis demonstrates the viability of using the infrastructure to provide access control in distributed environments. It demonstrated that, with SPKI/SDSI, security can be integrated into systems in a manner which embodies the end-to-end argument: using certificates and digital signatures, security decisions can be made at the 'ends', on the clients and servers.

The research presented in this thesis is currently playing a fundamental role in integrating security into MIT's Project Oxygen[34], and one system[2] is already using this research as the basis of its security. Two interesting directions for future research are the integration of SPKI/SDSI into a Peer-to-Peer system, and the development of a distributed certificate chain discovery algorithm, in which all the certificates may not be present on one computer, but may be present on several computers over a network.

Appendix A

SPKI/SDSI Demo Objects

The appendix shows the objects used in the demo outlined in Section 5.3 on page 114. The format of the Sexps is described in the SPKI/SDSI IETF Draft[11].

```
SPKI/SDSI on
AclFile demo/spki-sdsi-acls/ABC_financial_acl
ErrorFile demo/ABC/public/error.html
```

Figure A-1: demo object: .htaccess file protecting the financial directory

```
(acl
 (entry
  (subject
   (name
    (public-key
     (rsa-pkcs1-md5
      (e #23#)
      (n
       |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3C
       MBkkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMg
       bR4Fk+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH+
       +Txw9NAHq4r|)))
    ABC_executive_committee))
  (tag
  (http
    (* set GET)
    (*
     prefix
      http://ostrich.lcs.mit.edu:8081/demo/ABC/financial/))))
 (entry
  (subject
   (name
    (public-key
     (rsa-pkcs1-md5
      (e #23#)
      (n
       |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3C
       MBkkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMg
       bR4Fk+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH+
       +Txw9NAHq4r|)))
    ABC_auditors))
  (tag
   (http
    (* set GET)
    (*
     prefix
     http://ostrich.lcs.mit.edu:8081/demo/ABC/financial/))))
  )
```

Figure A-2: demo object: the ACL ABC_financial_acl

```
(tag
 (http
  GET
   http://ostrich.lcs.mit.edu:8081/demo/ABC/financial/budget2000.html))
```

Figure A-3: demo object: tag created from client's HTTP request

```
(sequence
 (tag
  (http
   GET
    http://ostrich.lcs.mit.edu:8081/demo/ABC/financial/budget2000.html))
 (timestamp
  2001-08-02_23:15:10))
```

Figure A-4: demo object: tag-timestamp sequence

```
(signature
 (hash  md5  |NaIIcSv3MQJwdeG2F9sEWg==|)
 (public-key
  (rsa-pkcs1-md5
   (e  #23#)
   (n
    |AKg3tOzoJ5PGQ5q9jzxzwxE8o6bIZ6/cE8gEL+1xJa23viE3bz68ruh
    pD5muqJ+uyDCNxgAZ0JVXJazmX1QjiGudj9kEmuni8gJRLZRu0T5E3K7
    OU2dodu0kdDg32kym7+ooZNe/F0zWGekfESeezyQ25kvNO3XQvMHXafW
    cYjRw|)))
 (rsa-pkcs1-md5
  |Z85SJP0CygufKofBZXcL6ISXFeOYfyoGnCwh0vX07RgemHWIJJTRsHzx4
  7NfnkoSKcpDy+cG8NnhdULFw0Ymnc5sxPzPJUxIYdrQZToFk52VUxgR3tb
  ibzeM9CrvL5qZ19lpFiBXDb1KfWsAEYKL9PhW6D6oHEAof0Q50bbIIMc=|)
  )
```

Figure A-5: demo object: signature

```
(cert
 (issuer
  (name
   (public-key
    (rsa-pkcs1-md5
     (e #23#)
     (n
      |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3C
      MBkkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMg
      bR4Fk+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH+
      +Txw9NAHq4r|)))
   Alice))
 (subject
  (public-key
   (rsa-pkcs1-md5
    (e #23#)
    (n
     |AKg3tOzoJ5PGQ5q9jzxzwxE8o6bIZ6/cE8gEL+1xJa23viE3bz68ru
     hpD5muqJ+uyDCNxgAZ0JVXJazmX1QjiGudj9kEmuni8gJRLZRu0T5E3
     K7OU2dodu0kdDg32kym7+ooZNe/F0zWGekfESeezyQ25kvNO3XQvMHX
     afWcYjRw|)))))
(signature
 (hash md5 |J1F6I3PH8B7cx9nHb5XXIA==|)
 (public-key
  (rsa-pkcs1-md5
   (e #23#)
   (n
    |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3CMB
    kkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMgbR4F
    k+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH++Txw9N
    AHq4r|)))
 (rsa-pkcs1-md5
  |aekwGvKshxzWP9AI9ViKq4AKzIB/wb4Ub4I1CUh3Z0p2Nqa0/4J/qL4dW
  4DBIQfGFNhCazjn3DIQJbjQan9TsLh7G2lfysrcozVpPCwqnLfrJUwdyTS
  M8yU6795T4pg4RhLY33MDsc85MSHX6qGYbAP26GzXmBY66JPAqZJIjAk=|
  ))
```

Figure A-6: demo object: Alice's name certificate

```
(cert
 (issuer
  (name
   (public-key
    (rsa-pkcs1-md5
     (e #23#)
     (n
      |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3C
      MBkkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMg
      bR4Fk+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH+
      +Txw9NAHq4r|)))
   ABC_auditors))
 (subject
  (name
   (public-key
    (rsa-pkcs1-md5
     (e #23#)
     (n
      |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3C
      MBkkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMg
      bR4Fk+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH+
      +Txw9NAHq4r|)))
   Alice)))
(signature
 (hash md5 |4/quwr4uJ8a0tOBaNCKrHQ==|)
 (public-key
  (rsa-pkcs1-md5
   (e #23#)
   (n
    |AMMgMuKpqK13pHMhC8kuxaSeCo+yt8TadcgnG8bEo+erdrSBveY3CMB
    kkZqrM0St4KkmMuHMXhsp5FX71XBiVW1+JGCBLfI7hxWDZCxGTMgbR4F
    k+ctyUxIv3CQ93uYVkg9ca6awCxtS0EI7sLuEB+HKuOLjzTsH++Txw9N
    AHq4r|)))
 (rsa-pkcs1-md5
  |JxvEayYstAs2aypi422iJ/x0/CKzvwVT0WheIqKJhopkuY4SAyre53tKj
  cqw1wy91EnGgdoLvLTTmUn/yPRVNLsiffAo96w87tN2KP1rnOp06rrQVf8
  +4cBKNKQ/b4vFddKIBZ4dK6UqGDRaafUeQsAXPi2M3yFuDCeKU9220Pg=|
  ))
```

Figure A-7: demo object: Alice's name (group) certificate

Bibliography

[1] Daniel J. Barrett & Richard E. Silverman. *SSH The Secure Shell. The Definitive Guide.* O'Reilly & Associates, Inc., 2001.

[2] Matt Burnside, Dwaine Clarke, Srinivas Devadas, and Ronald Rivest. Integrating Resource Discovery and Communication with SPKI/SDSI Security. In preparation for submission to a conference.

[3] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, Ronald L. Rivest. Certificate Chain Discovery in SPKI/SDSI. To appear in *Journal of Computer Security.* See `http://theory.lcs.mit.edu/~rivest/ClarkeElElFrMoRi-CertficateChainDiscoveryInSPKISDSI2.ps` Last visited 08/07/2001.

[4] Roger Clarke. Conventional Public Key Infrastructure: An Artefact Ill-Fitted to the Needs of the Information Society. Prepared for submission to the 'IS in the Information Society' Track of the *European Conference on Information Systems (ECIS 2001).* See `http://www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html`. Last visited 08/07/2001.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[6] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6), Nov. 1976, pages 644-654.

[7] Jean-Emile Elien. Certificate Discovery Using SPKI/SDSI 2.0 Certificates. Master's thesis, EECS Dept., Massachusetts Institute of Technology, May, 1998. See http://theory.lcs.mit.edu/~cis/theses/elien-masters.pdf. Last visited 08/07/2001.

[8] Carl Ellison. Cryptography Timeline. http://world.std.com/~cme/html/timeline.html. Last visited 08/07/2001.

[9] Carl Ellison. Establishing Identity Without Certification Authorities. *6th USENIX Security Symposium*, July 1996. See http://world.std.com/~cme/usenix.html. Last visited 08/07/2001.

[10] Carl Ellison. SPKI/SDSI and the Web of Trust. See http://world.std.com/~cme/html/web.html. Last visited 08/07/2001.

[11] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *Simple Public Key Certificate.* The Internet Society. July 1999. See http://world.std.com/~cme/spki.txt. Last visited 08/07/2001.

[12] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *SPKI Examples.* The Internet Society. July 1999. See http://world.std.com/~cme/examples.txt. Last visited 08/07/2001.

[13] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *RFC 2693: SPKI Certificate Theory.* The Internet Society. September 1999. See ftp://ftp.isi.edu/in-notes/rfc2693.txt. Last visited 08/07/2001.

[14] Carl Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *RFC 2693: SPKI Requirements.* The Internet Society. September 1999. See ftp://ftp.isi.edu/in-notes/rfc2692.txt. Last visited 08/07/2001.

[15] What do you need to know about the person with whom you are doing business? House Science and Technology Subcommittee. Hearing of 28 October 1997: Signatures in a Digital Age. Written testimony of Carl M. Ellison. See http://world.std.com/~cme/html/congress1.html. Last visited 08/07/2001.

[16] C. Ellison and B. Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. See `http://www.counterpane.com/pki-risks.pdf`. Last visited 08/07/2001.

[17] C. Ellison and B. Schneier. Risks of PKI: Electronic Commerce. *Communications of the ACM*, 43(2), Feb. 2000. See `http://www.counterpane.com/insiderisks5.html`. Last visited 08/07/2001.

[18] Warwick Ford, and Michael S. Baum. *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption.* Prentice Hall PTR, 1997.

[19] Matthew H. Fredette. An implementation of SDSI - the Simple Distributed Security Infrastructure. Master's thesis, EECS Dept., Massachusetts Institute of Technology, May, 1997. See `http://theory.lcs.mit.edu/~cis/theses/fredette-masters.ps`. Last visited 08/07/2001.

[20] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Do's and Don'ts of Client Authentication on the Web. In the Proceedings of the *10th USENIX Security Symposium*, August 2001.

[21] R. Fielding, J.Gettys, J. Mogul, H.Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.* The Internet Society. 1999. See `ftp://ftp.isi.edu/in-notes/rfc2616.txt.` Last visited 08/07/2001.

[22] Simson Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, Inc., 1995.

[23] Simson Garfinkel with Gene Spafford. *Web Security & Commerce.* O'Reilly & Associates, Inc., 1997.

[24] Ed Gerck. Overview of Certification Systems: X.509, PKIX, CA, PGP & SKIP. *The Bell*, ISSN 1530-048X. See `http://www.thebell.net/papers/certover.pdf`. Last visited 08/07/2001.

[25] Shafi Goldwasser, Silvio Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2), 1984, pages 270-299.

[26] ITU-T X.509v3. `http://www.mcg.org.br/mirrors/97x509final.doc`. Last visited 08/07/2001.

[27] Loren M Kohnfelder. Towards a Practical Public-key Cryptosystem. Bachelor's thesis, EECS Dept., Massachusetts Institute of Technology, May, 1978.

[28] Charlie Kaufman, Radia Perlman, Mike Speciner. *Network Security, Private Communication in a Public World.* Prentice Hall PTR, 1995.

[29] Butler W. Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4), Nov. 1992, pages 265-310. See `http://research.microsoft.com/lampson/45-AuthenticationTheoryandPractice/Abstract.html`. Last visited 08/07/2001.

[30] Butler W. Lampson. Computer Security in the Real World. See `http://research.microsoft.com/lampson/Slides/SecurityAbstract.htm`. Last visited 08/07/2001.

[31] Butler W. Lampson. Hints for Computer System Design. *ACM Operating Systems Review*, 15(5), Oct. 1983, pages 33-48. See `http://research.microsoft.com/lampson/33-Hints/Abstract.html`. Last visited 08/07/2001.

[32] Ben Laurie & Peter Laurie. *Apache: The Definitive Guide.* O'Reilly & Associates, Inc., 1997.

[33] Andrew Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, EECS Dept., Massachusetts Institute of Technology, June, 2000. See `http://theory.lcs.mit.edu/~cis/theses/maywah-masters.ps`. Last visited 08/07/2001.

[34] MIT LCS and AI Labs. MIT Project Oxygen. http://oxygen.lcs.mit.edu/. Last visited 08/07/2001.

[35] MIT PGP Team. PGP Freeware. http://web.mit.edu/network/pgp.html. Last visited 08/07/2001.

[36] Network Associates, Inc. and its Affiliated Companies. How PGP works. See http://www.pgpi.org/doc/pgpintro/. Last visited 08/07/2001.

[37] J. Paajarvi. XML Encoding of SPKI Certificates. http://world.std.com/~cme/draft-paajarvi-xml-spki-cert-00.txt. Last visited 08/07/2001.

[38] Jerome H. Saltzer, M. Frans Kaashoek. Topics in the Engineering of Computer Systems. M.I.T. 6.033 class notes, draft release 1.10. 6 February 2001, pages 6-5 to 6-109.

[39] J.H. Saltzer, D.P. Reed and D.D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), Nov. 1984, pages 277-288. See http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf. Last visited 08/07/2001.

[40] Lincoln D. Stein. Do “Cookies” Pose any Security Risks?. See http://www.w3.org/Security/Faq/wwwsf7.html#Q66. Last visited 08/07/2001.

[41] Lincoln Stein & Doug MacEachern. *Writing Apache Module with Perl and C.* O’Reilly & Associates, Inc., 1999.

[42] Eric Rescola. *SSL and TLS: Designing and Building Secure Systems.* Addison-Wesley, 2001.

[43] R. L. Rivest, A. Shamir, and L. Adleman. A Method For Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2), Feb. 1978, pages 120-126. See

http://theory.lcs.mit.edu/~rivest/rsapaper.pdf. Last visited 08/07/2001.

[44] Ronald L. Rivest. Can We Eliminate Certificate Revocation Lists? Proceedings of *Financial Cryptography '98*; Springer Lecture Notes in Computer Science No. 1464 (Rafael Hirschfeld, ed.), February 1998. See http://theory.lcs.mit.edu/~rivest/Rivest-CanWeEliminateCertificateRevocationLists.ps. Last visited 08/07/2001.

[45] Ronald L. Rivest. SEXP—(S-expressions). See http://theory.lcs.mit.edu/~rivest/sexp.html. Last visited 08/07/2001.

[46] Ronald L. Rivest and Butler Lampson. SDSI - A Simple Distributed Security Infrastructure. See http://theory.lcs.mit.edu/~rivest/sdsi10.ps. Last visited 08/07/2001.

[47] SSH Communications Security Corp. Choosing the Best Solution for Network Security: Secure Shell, TLS or IPSec. White paper. Version 1.0, January 2001. Available on request from SSH Communications Security Corp.

[48] The Stanford SRP Authentication Project. The Secure Remote Password Protocol. See http://srp.stanford.edu/. Last visited 08/07/2001.

[49] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. The *Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press, November 1996, pages 29-40. See http://www.counterpane.com/ssl.html. Last visited 08/07/2001.